Performance Oriented Partial Checkpoint and Migration of LAM/MPI Applications

A thesis presented

by

Rajendra Singh

to

The Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Computer Science

The University of Manitoba

Winnipeg, Manitoba

January 2011

Thesis advisor

Dr. Peter Graham

Author

Rajendra Singh

Performance Oriented Partial Checkpoint and Migration of LAM/MPI Applications

Abstract

In the parallel computing community MPI is ubiquitous due to the ability it offers to its users to harness relatively cheap distributed memory computing resources while coding in the widely accepted Single Program Multiple Data (SPMD) style. In shared parallel computing environments, however, participating nodes can become unexpectedly overloaded. If only a few nodes out of all the nodes comprising the MPI execution environment are affected by the overload it would result in overall application slowdown. Thus, it is very desirable to relocate the affected processes in the running application by partially checkpointing and migrating only the processes on the overloaded nodes to lightly loaded nodes. In many MPI applications, subsets of the processes communicate frequently with one another. We therefore have to be able to predict such groups of communicating processes so they can be checkpointed and migrated together for communication efficiency reasons. Such partial checkpoint and migration will allow long running applications to finish faster than if a subset of their processes was left running on overloaded node(s).

I have built a prototype, using LAM/MPI, that allows partial checkpoint, migration and restart of LAM/MPI applications by checkpointing and migrating selected

processes that are affected directly or indirectly (due to inter-process communication) by node overloading. To group processes that communicate with process(es) on the overloaded node(s), I gather and use communication information from the running LAM/MPI application. I adapted TEIRESIAS (an algorithm for pattern discovery from bio-informatics) to discover the frequent, recurring patterns of communication in my collected communications information. My predictors then analyze the discovered communication patterns and make predictions about groups of communicating processes that should be checkpointed and migrated together.

I assessed the effectiveness and usefulness of my prototype and technique using synthetic as well as real communication data to show that my predictors can accurately predict the groups of communicating processes that should be checkpointed and migrated together. Additionally, I created a simulation system building building on my techniques and collected data to allow me to explore scenarios related to network characteristics and various overload conditions under which my system might provide speedup.

The results that I obtained from my experiments and simulations indicate that my technique should be successful for a useful range of application types, network characteristics and overload conditions.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my supervisor, Dr. Peter Graham, for all the motivation, support for my ideas and encouragement throughout this research. He was a constant source of support, without which this thesis could not have been completed. I really appreciate the way he helped me organize the research work, which not only ended in the completion of my thesis but also in my learning much about how to do research and present the results.

I am grateful to my parents (and relatives) for all their encouragement and for extending their support always, without which I would not be what I am today. I would also like to extend my gratitude to my sister and brother-in-law for having put the insight into me of obtaining higher education in Canada. I love my nieces Deeksha and Divya.

My wife Neelam was a constant source of support and motivation throughout my PhD and was always there to share my sorrows and happy times, which kept me focused on my research leading to completion of the work. I am also grateful to her for bringing my son Aaditya to our life, who is such a sweet heart that his sweet activities kept me joyful and energetic during the critical phases of my research. I love you Adi.

I would also like to extend my thanks to all my committee members, Dr. Deborah Stacey, Dr. Robert McLeod and Dr. Parimala Thulasiraman for providing excellent reviews of my research, which helped me compile a quality thesis. Thanks also to all my friends for extending their support throughout.

Above all, I would like to thank the God Almighty for this wonderful life.

This thesis is dedicated to several important people in my life:

My Parents

Father Shri S.N. Singh and Mother Smt. Saroj Singh

My Wife and Son

Neelam Singh and Aaditya Raj Singh

My Sister and her family

Aruna Adil and Brother-in-law Dr. G.K. Adil and my nieces Deeksha Adil and Divya Adil

Almighty God

Above all, I would like to dedicate my Thesis to Shri Adi Shakti Mata Pita

Chapter 1

Introduction

The types of "computing" used have evolved over the last few decades. Only a little more than 60 years ago, when the first serial program was written, it was a great achievement and lead to huge changes in our lives. With the advent of parallel computing and development of tools such as PVM [Sun90] and MPI [GLDS96], computing has stepped forward by allowing multiple tasks to be done concurrently using a single program. This offers the promise of solving larger and/or more difficult problems and potentially deriving more accurate answers to problems that would otherwise take too long.

High Performance Computing (HPC)[1], as the name suggests, focuses on the overall performance of the computation when solving a time constrained problem, such as weather forecasting, using parallel programming. Such computational problems require huge computing power, which in general is not available in one single machine.

[1]In this thesis the term HPC is not intended to confer any special meaning with respect to the type of computing environment used. Thus, HPC could be done on a range of systems including special purpose parallel machines, dedicated or shared clusters of computers or combinations thereof.

Parallel or distributed computing environments having multiple processing engines and large collective memories are better suited for such problems.

HPC systems use either shared or distributed memory architectures. The former, as the name suggests, have all processors/cores sharing the same memory. This makes them relatively simple to program and efficient but expensive and limited in size. Distributed memory machines associate separate memories with different processors. These machines require messaging between processors, but are cheaper and more scalable.

Almost since the beginning of High Performance Computing (HPC), researchers have strived to utilize existing shared computing resources (such as idle desktop machines) to minimize the cost of HPC. This has met with some success for a limited range of problems. As well as minimizing costs, users of HPC need to perform their tasks fault free and in a timely fashion. Such users come from many fields including chemistry, astronomy, weather forecasting, biology, etc., many without access to large dedicated HPC systems, making the use of shared resources an important goal. With the growing deployment of high speed networks, distributed computing based HPC systems used to harness available resources across even relatively wide geographic areas have started to develop (e.g. computational grids [FKT01]). The research described in this thesis addresses one aspect of making the use of shared resources effective.

HPC users have high performance and reliability demands due to large size and relatively long runtimes of their jobs. If their work is to be done using a shared distributed computing environment, there are several issues that need to be addressed.

These issues include, finding suitable resources for performing jobs, managing the execution of the jobs on the resources allotted, and ensuring the performance and reliability requirements of the users for execution of their jobs. In this thesis I deal with the issue of meeting the performance requirements of users when executing their jobs in a distributed HPC environment. Specifically, I focus on the design of an efficient *partial* checkpoint/migration mechanism for use when application performance degrades in some part(s) of a distributed parallel application. The ability to move one or more poorly performing processes to lesser loaded compute nodes without stopping the entire HPC job offers the promise of an effective way of dealing with load variation on shared computing resources.[2]

Much work has been done trying to exploit the unused cycles available in many networked computing environments. Some success has been achieved but, typically, only in limited scenarios. Notably, systems such as Condor [LLM88] have been successful in exploiting such resources for "high throughput" style computations(those with little/no communication between running processes). I am interested in exploring how we can broaden the type of applications that can benefit from the availability of such resources. Due to the extremely widespread use of MPI [GLDS96] it seemed logical to focus on what would be required to make at least some reasonable subset of MPI applications run effectively in an environment of shared machines. This thesis considers one such requirement – a practical partial checkpoint-migrate-restart capability that involves only those processes impacted by overloaded compute hosts/nodes.

[2]It is important to distinguish the work in this thesis from load balancing. The purpose of the research presented in this thesis is to help ensure the efficient execution of a given HPC application. While partially checkpointing and migrating some MPI application processes **may** have the effect of balancing load across some set of machines, there is no algorithmic or system level intent to accomplish load balance.

Condor supports checkpoint, migration and restarting of individual processes and this has helped it be successful. However, such facilities are very rare for general MPI implementations and are not available for use in shared computing environments. This is, in part, probably due to the fact that checkpointing processes which are not independent of one another is a much more challenging problem, especially if other processes are to remain in execution.

One might argue that "computational grids"[FKT01; FK97] now provide enormous compute power to solve large scale computational problems, so what is the sense of creating another system for providing high performance computing. Grids, obviously provide access to enormous compute power but the problem space is often limited to loosely coupled distributed applications (those with constrained inter-process communication requirements). Further, it can be difficult to port MPI applications and expertise to grid environments where a different set of parallel programming tools is common. MPI is the pre-eminent parallel programming environment and is widely used in virtually all HPC research communities and has been used for many years. Researchers from outside Computer Science/Computer Engineering (the bulk of HPC users) are usually hesitant to learn new technologies and want to use what they have used previously in a familiar programming environment (such as MPI). I thus chose MPI as my target environment to leverage its already widespread use in the research community and its familiarity to users. The fact that MPI programs can be run on dedicated as well as shared compute clusters also makes it convenient for researchers to construct their own smaller-scale clusters using compute resources available at their disposal to develop and run their initial MPI programs and later

use larger, potentially shared resources to run long jobs using the same code.

In dedicated clusters the machines are used explicitly so that computing can be performed in parallel. Individuals typically do not own the machines that are a part of the cluster. Whenever an individual wants to run a job, he/she submits the job to the cluster of machines (typically via a host node) and the job is done in a parallel fashion using as many dedicated processors as necessary and the result is returned to the individual once the job is finished. With non-dedicated (i.e. "shared") clusters (currently typically seen in LAN environments), machines are owned by individuals but made available for use by others when not fully utilized. The main idea behind such non-dedicated cluster computing (using shared resources) is to "steal" the idle CPU cycles that a machine has without violating the rights of the owner of the system. A serious concern in non-dedicated clusters is that when the machine is used as a part of the cluster, the individual who owns that machine might not be able to use the computing power of the machine. To use non-dedicated clusters for long-running parallel jobs there are mechanisms(e.g. [Sin03]) to predict the likely availability of machines so they can be used when they are available without negatively affecting the access of the individuals who actually own them.

Just using LAM/MPI to try to achieve high performance in a shared environment is not sufficient due to the inevitable variation in load of the cluster nodes involved in running the MPI program. To make LAM/MPI programming and its runtime environment more performance oriented in such a cluster environment we need to support partial checkpoint and migration while, at the same time, keeping the behaviour of the MPI programming and runtime environment intact for the researchers.

There are several tools that support the construction and use of Grids to provide on-demand computational power that would also be needed for the LAM/MPI runtime in a shared computing environment. While implementation of such tools is outside of the scope of this thesis, they need to be enumerated. It is likely that we could safely "borrow" such tools from Grids. These tools would include a Resource Management System (RMS) consisting of resource discovery tools that can find the compute resources to build a runtime execution environment (distributed cluster) and an application manager that can monitor the performance of the compute resources as well as the MPI application processes running on them.

Another important practical reason for choosing LAM/MPI in my thesis is that LAM/MPI already supports total checkpoint/restart. This saved me the work of creating support for checkpoint/restart and provided code to which I could make changes to support partial checkpoint, migrate and restart of just those process(es) of an MPI application that are running on overloaded node(s).

In environments where machines are shared, load guarantees cannot be made. If one or more machines become overloaded it will decrease application performance. This provides a strong motivation to be able to checkpoint and migrate the affected processes to new machines. Such *performance-driven* migration should not involve the entire set of application processes as this would be wasteful both in terms of lost progress and overhead (from migrating processes).

My hypothesis is that the use of partial checkpoint and migrate in an environment of shared computing resources will provide a performance benefit by checkpointing/migrating only selected poorly perfroming processes as opposed to doing a

complete checkpoint and migration of the entire application. Therefore, my ultimate research goal is to provide a *partial* checkpoint-restart facility that can be integrated with a pre-existing or separately created resource management system (RMS) so that when the RMS detects overload on one or more nodes associated with a running MPI application, it can signal the MPI application to checkpoint those processes that are impacted by the overloaded nodes so the processes can be migrated to a new set of lightly loaded nodes selected by the RMS and then subsequently restarted. By migrating only a subset of the processes, the overhead of checkpoint and migration is reduced and application progress continues to be made during migration by those processes "left behind". A fundamental challenge in doing this is to select the correct set of MPI processes to migrate. Just selecting the processes that are running on the overloaded nodes is insufficient since they may be tightly communicating with other processes that should be migrated as well for efficiency reasons. Naturally, such a partial checkpoint facility will not be useful for all MPI applications (e.g. those where all processes frequently communicate with all other processes) but it should be useful for large, long-running applications that exhibit predictable repeating inter-process communication patterns between specific sub-groups of processes.

In this thesis, I extend a general MPI implementation (LAM/MPI) to support the partial checkpoint and restart of an MPI application. For such partial checkpoint and migration to be effective, however, inter-process communication patterns must be considered which is also addressed in this thesis. My prototype implementation instruments the LAM/MPI Request Progression Interface (RPI) (which provides low-level support for inter-process communication) to efficiently gather timestamped

pair-wise inter-process communication events. This data forms the basis on which pattern discovery is done to determine inter-process communication patterns. These patterns are then used to predict the communication patterns expected in the near future, following a checkpoint event. The predicted patterns, in turn, provide the information needed to determine groups of processes that are expected to communicate frequently with one another. If one process in such a group needs to be migrated then all the processes in that group are checkpointed and co-migrated to a new set of lightly-loaded nodes. Thus, I ensure that these processes will be located near[3] each other and thus continue to run effectively. Finally, using a combination of experimentation (with synthetic and real, gathered communication data) and simulation, I assess the effectiveness of my partial checkpoint and migration scheme to show that for some widely used application types and for a range of useful loading and network characteristics, the approach is useful (i.e. should result in shorter application execution times).

The rest of this thesis is organized as follows. Chapter 2 provides general background and review work related to this thesis. Chapter 3 provides a detailed description of LAM/MPI itself. (A brief overview of programming in MPI is provided in Appendix A for those unfamiliar with MPI.) Chapter 4 provides motivation for my work and gives a problem definition. Chapter 5 discusses some factors influencing partial checkpoint and migration in LAM/MPI while Chapter 6 describes my implementation. Chapter 7 discusses how processes can be grouped for partial checkpoint and migration and how I chose to do this. Chapter 8 presents my experimental re-

[3]By "near" or "nearby" I refer to network proximity. The issue here is that processes running on nearby compute nodes will be able to communicate with one another efficiently.

sults. The thesis concludes in Chapter 9 with a summary of contributions made and a brief discussion of some possible areas for future work.

Chapter 2

Background and Related Work

2.1 Parallel Computing

Parallel Computing, in simple words, is the use of multiple compute resources simultaneously to solve a computational problem [Bar10]. Traditionally, programs were written for serial computation. The instructions were executed in a serial fashion to be executed one after the other. Serial programs are good for solving relatively small problems. When it comes to solving bigger, more computationally complex problems, however, we can not always afford to wait for months to get results using serial computing. Some of these bigger problems that can be broken down into relatively independent parts can be solved using parallel computing to get faster results. Researchers now use parallel scientific modelling and simulation to solve otherwise infeasible problems such as understanding nuclear bomb blasts, the study of galactic evolution, etc. This is particularly important where safety, scalability and/or cost make physical study impossible.

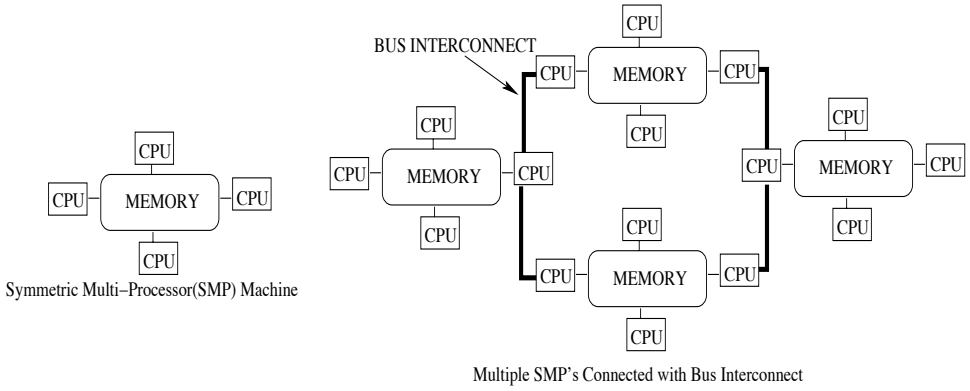

Figure 2.1: Shared Memory Parallel Machine Organization

Parallel computing systems are commonly divided into different types depending, primarily, on the memory organization. These include shared memory parallel computing systems, distributed memory systems and various hybrids of shared and distributed memory systems. In a shared memory parallel computing system parallel programs are written to share a common large memory. A typical example of such a system is a symmetrical multi-processor machine (SMP) with one large memory or several SMP's sharing memory with each other all connected with a high-speed interconnect, as shown in Figure 2.1. The advantages of this type of parallel machine are high performance due to locally shared memory and programming convenience. The primary disadvantage of such a system is poor scalability due to limitations on memory capacity upon addition of more processors. When more processors are added the load/traffic on the shared memory-processor path increases geometrically[Bar10] limiting the size of systems that can be built. The cost involved in building such a system is also very high.

Distributed memory systems (as shown in Figure 2.2), which are of particular

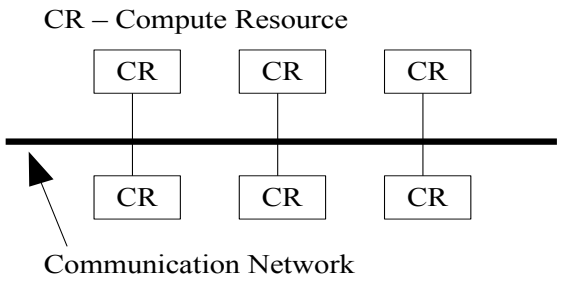

Figure 2.2: Distributed Memory Parallel Machine Organization

interest in this thesis, offer lower cost and better scalability of memory with increases number of processors. Adding a new processor adds new memory in proportion and the processors can access their associated memories without any interference[Bar10]. Of course, these memories are not shared between processors so explicit communication is required. Message passing parallel programming models have made these systems extremely successful by harnessing high-speed network connections available between the compute nodes to permit the solution of a number of large parallel programming problems, cost effectively.

Due to the differences in parallel computing organizations, several different parallel programming models have evolved. Custom thread based and Open MP[CJP07] programs can be written for use in shared memory parallel computing environments. Multi-threaded applications can be visualized as a single program with several concurrent execution activities in it [Bar10], each executing a subroutine to solve part of a single problem using a shared global memory to cooperate. In thread based parallel programs, parallelism is achieved by using multi-threading libraries (i.e. writing a POSIX threads [But97] program).

Open MP is a compiler directive based approach to parallel programming in shared

memory parallel computing environments that uses user specified *pragmas* (compiler directives) to direct the parallel execution of a given program based on serial code. An example Open MP program for matrix-vector multiplication is shown in Figure 2.3, that illustrates how the code is naturally shared between threads. Using OpenMP, one can define the regions of a program to be run in parallel by different threads by using, for example, the directive *#pragma omp parallel shared(...)* as shown in Figure 2.3.

The availability of low cost wide-spread clusters has given rise to a very popular parallel programming tool for distributed memory parallel computing environments known as MPI (the Message Passing Interface), a standard for message passing parallel programming. MPI based parallel programs spawn MPI processes on several machines, which use their own local memory for local computations and send and receive messages to other processes of the same MPI application to interact with them. These messages are sent and received by using special MPI library calls (E.g. *MPI_Send(...) and MPI_Recv(...)*). MPI based parallel programming can utilize both dedicated clusters as well as clusters built using several (co-located) distributed networked machines. An example of an MPI program for matrix-vector multiplication is shown in Figure 2.4. The code in Figures 2.3 and 2.4 both solve the same problem and collectively highlight the general distinction between the shared memory (e.g. Open MP) based parallel programming and distributed memory (e.g. MPI) based parallel programming styles.

In the case of shared memory parallel applications the data resides in the common shared memory among processors working together to solve the problem, as shown

```
#include <omp.h>
...
int main (int argc, char *argv[]){
     ... declarations  ...

     chunk = 10;   /* set loop iteration chunk size */
     /*** Spawn a parallel region explicitly scoping all variables ***/
     #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k){
          tid = omp_get_thread_num();
          if (tid == 0){
               nthreads = omp_get_num_threads();
          }
          /* Initialize matrices */
          #pragma omp for schedule (static, chunk)
          for (i=0; i<NRA; i++)
               for (j=0; j<NCA; j++)
                    a[i][j]= i+j;
          #pragma omp for schedule (static, chunk)
          for (i=0; i<NCA; i++)
               for (j=0; j<NCB; j++)
                    b[i][j]= i*j;
          #pragma omp for schedule (static, chunk)
          for (i=0; i<NRA; i++)
               for (j=0; j<NCB; j++)
                    c[i][j]= 0;

          /* Do matrix multiply sharing iterations on outer loop */
          /* Display who does which iterations for demonstration purposes */
          printf("Thread %d starting matrix multiply...\n",tid);
          #pragma omp for schedule (static, chunk)
          for (i=0; i<NRA; i++){
               printf("Thread=%d did row=%d\n",tid,i);
               for(j=0; j<NCB; j++)
                    for (k=0; k<NCA; k++)
                         c[i][j] += a[i][k] * b[k][j];
          }
     }   /*** End of parallel region ***/
     ... Print results ...
}
```

Figure 2.3: Matrix-Vector Multiplication in Open MP (adopted from [Bar10])

in Figure 2.5. In the shared memory part of Figure 2.5 the lines show the activity of accessing the memory by obtaining a lock on the part of the data structure accessed by a process. If a processor, $P2$, needs to access the same data that processor $P1$ is

```
#include "mpi.h"
int main (int argc, char *argv[]){
     ... declarations ...
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
/***** master task *****/
   if (taskid == MASTER){
      ...    Initializing arrays   ...
      /* Send matrix data to the worker tasks */
      averow = NRA/numworkers;
      extra = NRA%numworkers;
      offset = 0;
      mtype = FROM_MASTER;
      for (dest=1; dest<=numworkers; dest++){
         rows = (dest <= extra) ? averow+1 : averow;
         printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
         MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
         offset = offset + rows;
      }
      /* Receive results from worker tasks */
      mtype = FROM_WORKER;
      for (i=1; i<=numworkers; i++){
         source = i;
         MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
         MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
         MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
         printf("Received results from task %d\n",source);
      }
      ... Print results ...
   }
/***** worker task *****/
   if (taskid > MASTER){
      mtype = FROM_MASTER;/* receive matrix from master */
      MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
      for (k=0; k<NCB; k++)
         for (i=0; i<rows; i++){
            c[i][k] = 0.0;
            for (j=0; j<NCA; j++)
               c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }
      mtype = FROM_WORKER;/* send result to master */
      MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
      MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
   }
   MPI_Finalize();
}
```

Figure 2.4: Matrix-Vector Multiplication in MPI (adopted from [Bar10])

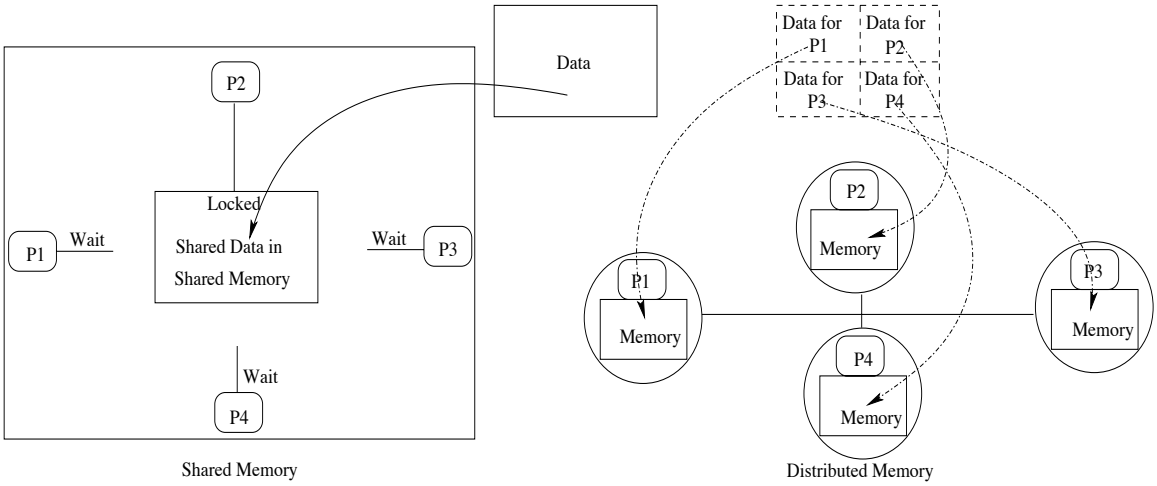

Figure 2.5: Shared vs. Distributed Data Access

working on then processor $P2$ will have to wait for the lock to be released by processor $P1$ on that data. Such synchronization is necessary in shared memory architectures.

In the case of distributed memory applications, the data is divided and distributed among the memories associated with each processor. The processors then work on solving their part of the problem independently and when ever they need some data from other processors they must send a request using message passing to the processor that has the data. Figure 2.5 (right hand side) shows how data is localized to specific processors. The focus of this thesis is on message passing MPI based parallel applications. A description of the relevant aspects of MPI is provided in AppendixA for readers who may be unfamiliar with MPI.

Shared memory based parallel programming tools, such as Open MP, while convenient can only be used effectively on costly shared memory machines and this limits their use by most users to the smaller scale (100s of processes). On the other hand, message based parallel programming with MPI for applications having mod-

erate communications demands can cost effectively utilize more machines and hence offer greater practical scalability. Further, improvements in network latency and bandwidth have enabled message passing in distributed memory machines to become more efficient than before, leading to an increased use of clusters and MPI in high performance parallel computing. It is now possible to effectively use MPI across machines connected using dedicated networks in relatively local (e.g. organization-wide) environments. The "closeness" of nodes in the network used, however, naturally affects the range of problems that can be effectively solved.

2.2 mpC: parallel programming language

mpC [Las02] is a parallel programming language, based on C, developed for programming high performance computations on a local area network of computers. It uses MPI libraries for message passing but of different code blocks within an application can be mapped on to different sets of processors (described by a "network type").

In mpC, like MPI, jobs can run on different numbers of machines in SPMD fashion but unlike MPI, the number is not pre-determined by the execution environment. The program itself can construct virtual networks of machines from an existing pool at runtime. mpC also allows non-SPMD (heterogenous code) applications to run and facilities for load balancing are provided to such applications.

mpC does not support full dynamic migration of processes in response to loading conditions but in a local area context it does allow the programmer to specify the volume of computation to be done by heterogenous processes to allow for appropriate

load balancing. Further, the computational ability of nodes can be taken into account as well by running a test (RECON) computation to determine the ability of each node before assigning application processes to available nodes. A serial execution of the code to be run in parallel is commonly used for this purpose. Naturally, communications overhead is not assessed when doing this. RECON computations can be done multiple times for different computational blocks within a program.

A typical (simple) mpC code would have declarations followed by a RECON call. The network for computation is then built and uses the information from the RECON to do the load assignments. The actual computation then follows. This basic use of mpC does not support partial checkpoint/migrate or additional load balancing once the computation starts. However, repetition of the above steps, RECON, reconfiguring the network for computation and computation, multiple times does support load balancing and hence a migration-like ability of processes. Of course, potentially significant extra overhead would be incurred because between RECONs one would have to gather all application level partial results and re-distribute them given the new network for computation. Additionally, the programmer is responsible for inserting the code needed to achieve this load re-balancing behaviour.

mpC is interesting because it provides a basis for adaptation in message-based parallel computation but it relies on the user to write code to support this and although it runs over MPI, it does not use the widely accepted MPI API. Further, due to the overload of gathering and re-distributing all application data between RECONs it is likely limited to use in a local area network environment.

2.3 Grid Computing

Parallel computing tools to use computers across much larger geographic distances have also been developed. To enable the use of shared distributed computing resources in a wide area network setting, Grids[FKT01; FKNT02; FK97] have been proposed. In particular, the *globus*[FK97] toolkit has enabled utilization of distributed resources to provide high-performance computing environments for loosely coupled parallel applications. The term "The Grid" came into existence in the middle 1990's to denote the construction of a nation-wide computing infrastructure in the form of a very large scale distributed computing system built by connecting geographically distributed machines and other network resources available at various institutions and organizations[1]. Foster and Kesselman [FK99] argued that the ongoing need for more compute cycles to solve ever larger compute intensive problems could only be satisfied with the power of a national-scale computational Grid and this viewpoint is now accepted by many researchers.

Such Grids not only provide the huge aggregate power of their available computing resources but also the data and other significant resources that are contributed by the participating machines (e.g. storage, custom instruments, and applications). Grid computing therefore must deal with the problem of the provision and co-ordination of *general* resource sharing across various multi-institutional "virtual organizations" that need to concurrently solve computationally intensive problems [FKT01; FKNT02].

Each such virtual organization consists of individuals, departments and other

[1]The name is by analogy with national *power* grids.

groups who are willing to contribute their resources and who are also in need of more resources themselves for solving their own compute intensive problems. By sharing resources, each participant may, at certain times, have access to far greater resources than they would otherwise have access to. Sharing rules in the Grid strictly define what can be shared, when, and with whom. The Grid must provide services and tools for managing such virtual organizations.

The Grid Protocol Architecture [FK99] is layered like the Internet Protocol (IP) Architecture [Pos81] and consists of: "Fabric", "Connectivity", "Resource", "Collective" and "Application" layers. The fabric layer provides raw resources such as computational capacity, storage, network links, etc. which users have agreed to share using grid sharing protocols. The connectivity layer provides basic communication and authentication. To exchange data between resources provided by the fabric layer, a communication protocol is required. To maintain security in a grid system, authentication is required for the users as well as for the resources offered by the fabric layer. The connectivity layer provides both communication and the needed authentication protocols to support secure network transactions[2]. The resource layer implements protocols for secure negotiation for, monitoring and control of, as well as payment for, the resources used. It is built on the connectivity layer protocols and is meant to deal only with a single resource at a time. Multiple resource co-ordination is handled by the collective layer. Finally, the application layer consists of the applications themselves which actually run in the environment provided by the virtual organizations using the facilities provided by the collective layer.

[2]An example of such a protocol is the *Single Sign-on* [FKT01] protocol, which allows a user to log on once with an authentication check and then use all the resources offered by the fabric layer.

The Globus Metacomputing toolkit [FK97] is a widely used environment for constructing computational grids and serves as a useful example of a grid system. Globus provides a set of services to the user which can be used to build custom grid services. These services are well defined so they can be incorporated into other applications and tools. Globus follows an "hourglass" architecture, defining the neck of the hourglass as a well defined interface to access local services. Higher level services (in "the top of the hour glass") which can be more complex are developed on top of these local services by using the well defined interfaces. The interfaces in Globus are designed such that there is useful visibility of the heterogeneity involved rather than completely hiding it beneath any layer. This allows users to make new higher level services or applications based on these lower level interfaces which understand and leverage the underlying service intricacies. Such a "translucent" environment is stated to often be advantageous in heterogeneous computing environments.

In a resource sharing environment where it is not certain that the resources which are in use will remain available in the future, some services have to be developed which keep information about other resources so that they can be found when required for the execution of a job. Discovery of new resources and configuration of applications to dynamically adapt to the availability of the resources are provided by higher level services that are built on lower level services and are typically focused on specific issues such as scheduling, fault tolerance, etc.

2.4 Checkpoint and Restart

A checkpoint is the saved "state"[3] of a running process (or application[4]) which can be used to restart the execution of the process (or application) at a later time. Several checkpoints can be taken during the execution of a process so that, in case of a failure, the process can be restarted from the most recent checkpoint rather than starting again from scratch. Checkpoints can sometimes also be "migrated" to other machines. This allows work to be done on more lightly loaded machines thereby addressing performance issues as well as reliability. Migration is a mechanism for moving an already running process from one machine to another. The checkpointed data of the process is transferred to a different machine and the process is then restarted there. This process may be complicated by the need to re-establish prior existing communication links (e.g. new IP addresses) as well as by other factors.

Historically, checkpoint and restart was used exclusively for fault tolerance purposes in long-running jobs. In a long-running job where the results are obtained after a long time (days or months or longer) its always a good idea to regularly checkpoint the application to avoid loss of work in case of failure. The use of checkpoint and migrate in distributed systems for performance reasons is somewhat more recent.

Checkpointing can be of two types, Application Level Checkpointing (ALC) and Process/System Level Checkpointing (SLC). ALC checkpoints are generally smaller in size than SLC's because only the important data structures and high-level execution state of the application are saved, and the entire application can be restarted

[3]The "state" of a process contains all the details for initializing and running the process at a particular point in the process' execution.

[4]For brevity, I will hence forth consider process checkpointing only unless otherwise explicitly stated.

from them. In SLC an entire process image is saved including all code and data, etc. This is quite large. ALC is more portable because of less dependence on the system architecture while SLC is system dependent. An SLC type checkpoint taken on one system type can't be restarted on another system type (e.g. a Linux/x86 application can't run on a Solaris/SPARC machine). Apart from this ALC involves programming effort and knowledge from the programmer who must explicitly code checkpoint and restart routines. Using SLC there is no such involvement of the programmer. Historically, few HPC programmers have been willing to code checkpoint routines despite their obvious benefit. So, in this thesis, I consider only SLC type checkpointing.

Message-based parallel computing offers an environment in which a job runs on several different machines (possibly geographically separated) concurrently. The network connectivity between the machines involved supports communication for message passing between the job parts (processes) running on different machines. For the job (as a whole) to complete, it is naturally necessary that the machines chosen for running the job must all complete their parts of the job. Another concern for a parallel application is that jobs must normally be completed within a desired maximum time and job constituent processes are normally expected to make approximately equal progress over a period of time. Certain problems may arise with such applications that can be addressed using checkpointing and migration. These include failure of machines or overloading of machines due to unexpected reasons (such as, resumption of a queued job, submission of some new work, etc.). These factors may make it impossible to complete a parallel application in a timely fashion due to loss in performance at one or more machines. Such activities should not affect the execution

of a job, but if they may, then there must be some way to deal with such situations. One possible way is to checkpoint and migrate a job from the affected machine(s) to some other location(s) where it may execute more efficiently (e.g. [VD03]). This idea sounds simple but involves many issues, such as where to move the job to, how to move the job, how to adapt the job to its new location(s), what the effect on performance will be, etc.

An MPI application involves communication between application processes. Some application processes may frequently communicate with each other while others communicate only infrequently. Frequently communicating processes introduce strong communication dependencies between the machines they run on. Whether the executing processes require frequent communication or infrequent communication will play a major role in checkpointing and migration of the application in parts (as discussed in detail later).

The concept of homogeneous vs. heterogeneous environments in migration also needs to be considered. A migration environment can include either identical (homogeneous) or different (heterogeneous) types of machine platforms. Normally an application running on one OS/machine architecture can't be migrated to a machine with a different OS/machine architecture. A homogeneous environment is therefore much simpler. Considering heterogeneous environments requires conversion of the process state of one architecture to that of another [CCG+95]. In this thesis I deal with process checkpointing and migration resulting from the detection of performance degradation and assuming a homogenous migration environment. I now review checkpointing strategies generally, and several systems that employ checkpointing and

migration mechanisms in existing distributed systems.

2.4.1 Checkpointing in Multi-process Systems

Checkpointing in systems with multiple processes (whether they are distributed or not) is quite complex when compared to single process applications. This is because of the presence of multiple execution streams and the absence of a common global clock. In particular, it is not possible to initiate a checkpoint in all the execution streams at the exact same time[5].

The following fundamental definitions are taken "as is" from [KR00] with slight changes in wording to preserve the correctness of the definitions:

Definition 1: *"happens before":* (1) If a and b are two events occurring in the same process and if a occurs before b, then a "happens before" b. (2) If a is the event of sending a message and b is the event of receiving the same message in another process then, a "happens before" b. The "happens before" relation is transitive.

Definition 2: *"Concurrent Events":* Concurrent events occur if and only if a does not "happen before" b and b does not "happen before" a for events a and b.

Definition 3: *"Local Checkpoint":* Saving the state of a process at a processor at a given time is called a Local Checkpoint.

Definition 4: *"Global Checkpoint":* A collection of "local checkpoint"(s), one from each processor running an application is called a Global Checkpoint.

Definition 5: *"Globally Consistent Checkpoint":* If the collection of local checkpoints is such that each local checkpoint is from a separate processor and if each

[5]In this Subsection we are not considering migration issues.

local checkpoint is concurrent/consistent (consistent here means that the process of exchange of messages between processors has reached a state that avoids any orphan messages being left "in-flight") with every other local checkpoint then, this collection is said to be a globally consistent checkpoint.

Definition 6: *"Synchronous Checkpointing:"* A process of checkpointing in which several processes exchange messages and co-ordinate with each other at runtime to reach a globally consistent checkpoint.

Definition 7: *"Asynchronous Checkpointing:"* A process of checkpointing where processes can initiate checkpoints independently without any co-ordination with other processes.

Synchronous checkpoints take a little more time than asynchronous checkpoints because of the messages exchanged to reach a globally consistent checkpoint, but it is guaranteed that there will be no orphan messages and hence, no need of rollback to get rid of orphan messages. Using synchronous checkpointing, a recovery algorithm is not needed, unlike asynchronous checkpoint, to reach a globally consistent checkpoint. This can be time consuming as well but the cost is only incurred when recovery from a failed process is needed.

2.4.2 Checkpointing Strategies

In message passing systems, three different general types of checkpointing algorithms can be used, as described in Figure 2.6. These are referred to as coordinated algorithms, independent algorithms and quasi-synchronous algorithms and differ on when and how the global consistent checkpoint is made. If a consistent checkpoint

is made in coordination with all other nodes at runtime (by passing some messages in the form of *markers* or *headers*, as described later) then it is called coordinated checkpointing. Using this approach, the global consistent checkpointed state is stored on stable storage and upon restart the state can be easily restored.

Independent algorithms are asynchronous and thus do not coordinate during normal process execution and checkpointing, but rather they coordinate at restart time to form a consistent global state. Since these algorithms do not coordinate on an ongoing basis there is less message passing overhead. Instead messages are simply logged locally, so as to be able to coordinate during restart, based on what messages were passed, to create a consistent global state.

A quasi-synchronous approach to checkpointing is based on taking checkpoints asynchronously during normal process execution but with communication induced co-ordination. This mechanism reduces the number of useless checkpoints, which is common with asynchronous checkpointing, by inducing sufficient communication for co-ordination to reach a "recovery line"[6]. The advantage of having such a recovery line is that all the useless checkpoints taken prior to the latest recovery line can be discarded because the recovery line ensures a global consistent checkpointed state. The Quasi synchronous checkpoint approach is neither fully synchronous nor asynchronous, exploiting the low-cost of asynchronous checkpointing by taking checkpoints without any co-ordination and the advantages of synchronous checkpointing to maintain a consistent recovery line by inducing some communication for checkpoint co-ordination [MS99].

[6]A recovery line is a collection of checkpoints from all processes which is ensured to be in a globally consistent checkpointed state.

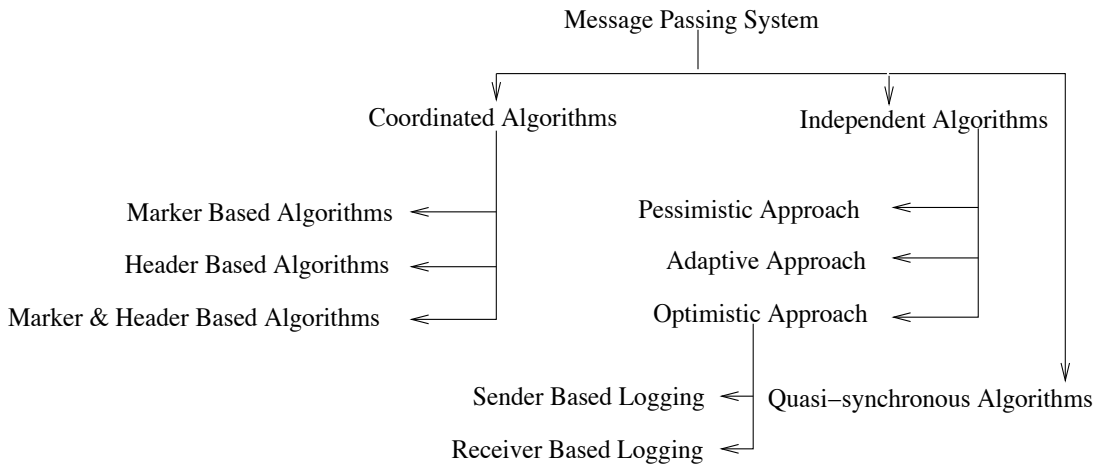

Figure 2.6: Classification of Checkpointing Algorithms for Message Passing Systems (adopted from [KR00])

As shown in Figure 2.6, coordinated algorithms may be based on markers or headers or both. Markers are messages passed between the different processors an application is running on to coordinate a consistent global checkpoint. Markers are used to identify different "checkpoint intervals" [KR00], the time spent by each processor executing between two consecutive checkpoints. Each checkpoint interval is identified by a number called its checkpoint interval number. The main purpose of using markers is to inform the receiving node that a checkpoint has to be taken and to establish a clear difference between messages belonging to different checkpoint intervals.

Headers are an alternative to markers, which contain the checkpoint interval number and are sent piggybacked on regular messages thereby decreasing the message passing overhead. Markers are used to inform the receiving processes to take checkpoints (processes are not capable of initiating checkpoints themselves). With header based algorithms processes have the ability to initiate checkpoints by themselves. Pro-

cesses differentiate between checkpoint interval numbers and then take appropriate checkpoints.

It is difficult to checkpoint when messages are received out-of-sequence. Processes receiving such out-of-sequence messages may start checkpointing or log the message incorrectly assuming it to be an in-sequence message. This problem can be solved by using message sequence numbers. Also, in cases where processes use header based algorithms, processes that have not communicated with each other between consecutive checkpoints may not be able to participate in a consistent global checkpoint (because they may not know the checkpoint interval number and/or message sequence numbers). To allow these processes to participate in consistent global checkpoints, both markers and headers can be used together where markers are used to inform such non-communicating processes to checkpoint and headers take care of the communicating ones.

Independent checkpointing algorithms use message logs to determine a global consistent checkpoint at restart time. The *Pessimistic* approach to message logging (Figure 2.6) involves logging messages as soon as they are received. The *optimistic* approach logs messages only periodically. Using the pessimistic approach, recovery becomes fast and effective at the cost of frequent stable storage accesses to add to the log. Recovery is fast and effective because all the messages are logged, which helps in establishing the most accurate consistent global checkpoint. In comparison, with the optimistic approach stable storage accesses are reduced but the chances of multiple rollback's are increased because of the possibility of incomplete message logs.

Optimistic message logging approaches can be classified into sender-based logging

and receiver-based logging (Figure 2.6). Receiver-based logging refers to logging of messages at the receiver's end. Using this approach it may happen that a message is lost in transit before it can be logged. This may result in a situation where the state cannot be re-created and the processor has to rollback to a previous saved state resulting in the sender having to roll back to the previous state too. In the case of sender based message logging, the messages are logged at the sender's end. In this case, if a failure happens, the receiver has to request any lost messages from the sender. Sender-based schemes may increase stable storage overhead for the sender and recovery may become more time consuming but rollbacks are not required.

2.4.3 Example Checkpointing Systems

A number of checkpointing systems for various environments and applications have been implemented. A representative sample is now reviewed.

Libckpt

Libckpt [PBKL95] is a checkpointing tool for Unix processes which facilitates user-directed checkpointing. Libckpt does incremental checkpointing. In incremental checkpointing [EJZ92], only that portion of the checkpoint data is saved which is newer than the previously saved checkpointed data. This means only the updated information is stored so saving state information is faster.

Libckpt is not user-transparent. It requires modification to the code before running under the Libckpt checkpointing environment. Libckpt handles the problem of the increase in the cumulative size of the checkpointing files (due to its use of in-

cremental checkpointing) by periodically combining the files to create a single new checkpoint file and getting rid of the old ones. To perform its incremental checkpointing, Libckpt uses a page protection mechanism to implement copy on write. By marking the existing pages in memory as read-only, any subsequent write causes an access violation which allows for the copying of the pages and setting the copy of the faulted page to allow subsequent read-write access. All read-write pages are then included in the next Libckpt checkpoint.

Libckpt also uses "memory exclusion" and synchronous checkpointing to increase performance. Memory exclusion is a technique used to exclude certain memory locations depending on data's expected life span while synchronous checkpointing allows the user to specify directives for checkpointing that indicate at what point in the application's execution it is beneficial to take checkpoints. This reduces the number of (useless) checkpoints taken. Plank, et al. [PBKL95] describe two situations when memory exclusion can be useful: first, when the data is dead (i.e. the data will never subsequently be read/written from/to a location), and second, when the data is clean, which means that the data that is to be checkpointed has already been written in a previous checkpoint. When the size of excluded memory is very small compared to the data that is being checkpointed, memory exclusion may not be beneficial. Such cases are dealt with using the synchronous checkpointing approach, where the checkpointing locations are chosen to be the frequently reached locations in a program.

Checkpointing in Condor

The Condor scheduling system [LLM88; BLL92] was originally designed to work in a network of workstations environment. It schedules jobs onto workstations depending on the activities of the people who own the workstations. Idle workstations are detected and jobs are scheduled onto them. The Condor system removes jobs and re-schedules them to other workstations when the owner resumes using their workstation (with the guarantee that such jobs will eventually be completed). The purpose of the Condor system is to maximize the use of a workstation environment's computing resources. It is not focused on supporting parallel programming (as in cluster computing).

The Condor system provides three fundamental functions to make efficient use of the computational capacity in a network of workstations: the analysis of workstation usage patterns, the design of remote capacity allocation algorithms, and the development of remote execution facilities. Condor uses the Up-Down scheduling algorithm [ML87] for capacity allocation and the Remote Unix (RU) [Lit87] facility is used for executing jobs remotely. A critical feature of RU is check-pointing. RU creates checkpoints in case, at some point in time, the job fails to execute or has to be removed from the current workstation. Each checkpoint saves the most recent state of the job (a single process in Condor) so that, once relocated, the job can be re-started from the saved state rather than from scratch.

Condor works with a centralized, static coordinator but is otherwise highly distributed. It is the central coordinator that locates the idle workstations and allocates jobs to them. Every user job is registered with the central coordinator. Worksta-

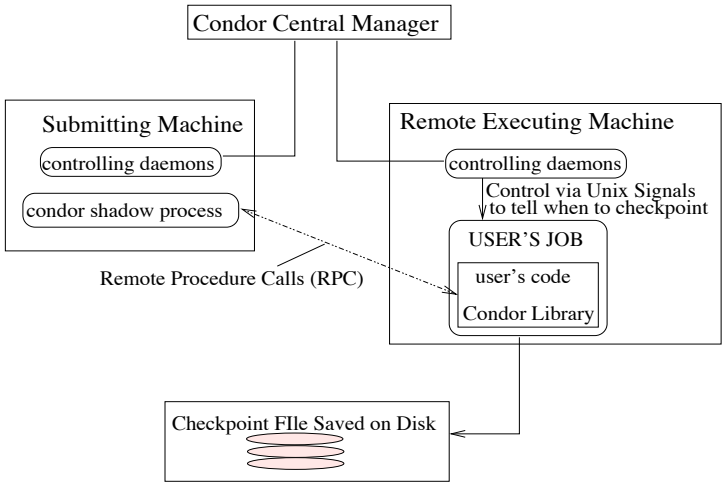

Figure 2.7: Condor Checkpointing (adopted from [LTBL97]

tions taking part in the system have a local scheduler and their own job queue. It is the responsibility of the local scheduler to schedule the jobs on that workstation, but which workstation is to receive a Condor job and how much capacity is to be allocated is decided by the central coordinator. The central coordinator itself runs on some workstation and uses the Up-Down algorithm to priorize the workstations participating in the system. It manages the available remote processing capacity in such a way that it can be fairly allocated among users, without being biased to users trying to use all the free cycles.

The checkpointing and migration mechanism in Condor is illustrated in Figure 2.7. When the owner of a machine resumes activity at the machine the Condor controlling daemon on the machine detects this and stops any Condor jobs that are running by sending a checkpoint signal to the job, which then invokes a checkpointing library to checkpoint its current state to a local disk. This checkpointed file is then sent to the originating machine (which submitted the job) [LTBL97]. The central coordina-

tor then finds a suitable new machine to run the job on and moves the checkpoint file to the selected machine where it is restarted and resumes execution from the checkpointed location.

Checkpointing Systems for PVM

MIST (Migration and Integrated Scheduling Tools) [CCG+95] is an enhancement of the PVM (Parallel Virtual Machine)[7] [Sun90] system which addresses issues such as scheduling, resource management, task migration and checkpointing. To have PVM applications perform better in a shared network environment despite unpredictable factors such as overloading of the machines on which a PVM application is running, unanticipated shutdown of machines, etc., a mechanism which supports task migration was developed. This task migration mechanism works in conjunction with a scheduler and a load monitor. The scheduler is responsible for selection of machines on which the job is to be started initially, checkpointing of a job when any unpredicted performance limiting event occurs, finding machines where the job should be migrated to in such cases and restarting the task at the newly found location(s). The scheduler gets information about the loads on machines from load monitors, which help the scheduler in making a decision concerning which machines to migrate to.

The checkpointing approach used with PVM is a simple one which involves stopping all the processes once they have all agreed to checkpoint. The process state is saved on disk and then transferred to the migration location(s) through socket connections. While checkpointing data to the disk, the application is allowed to run

[7]PVM is an earlier system that provides support for message based (distributed memory) parallel computing, similar to what MPI does.

(using cloned processes created via the fork() system call). This allows processes to continue to interact with other processes that have not yet begun their checkpoints. (e.g. to support open communication sockets, etc.)

Migratable Parallel Virtual Machine (MPVM) [CCK+95] offers another approach to enhancing PVM with a checkpointing and migration framework. It provides a transparent migration mechanism by saving the process state of a PVM process and restarting the process by reconstructing it at the migrated location. A PVM process consists of Unix processes which are linked with the run-time library *pvmlib*. Each such Unix process is called a task in PVM and they use *pvmlib* to communicate with each other. MPVM uses a migration protocol to perform migration of jobs from one location to another depending on certain control messages (passed between several components of the system to co-ordinate task migration and checkpointing) that are specific to managing task migration.

The migration process in MPVM (as shown in Figure 2.8) has 4 stages: 1. migration event, 2. migration initialization, 3. state transfer and 4. restart. The global scheduler determines where a job is to be migrated to when triggered by a migration event. To migrate a task, the global scheduler notifies the source PVM daemon (Spvmd) running on the host machine where the task is currently running. The notification message is accompanied by a tid (task id) and hid (host id) for the destination host, which are then verified to be valid by the host machine. The initialization process for migration is then started, which is itself a two step process performed at both the source (where the task is running) and the destination (where the job is to be migrated to). At the source machine, a message sent by the daemon (Spvmd) to the

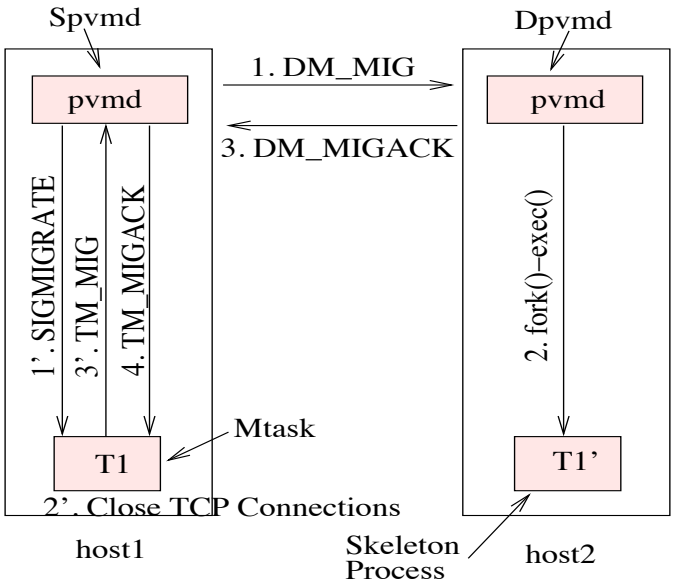

Figure 2.8: Task Migration Protocol in MPVM (adopted from [CCK+95]

task initiates the migration process which captures the process state. After the process state is captured it needs to be transferred to the destination machine, for which a TCP socket connection is established between the source and destination machines. Spvmd receives back a message from the task that the checkpoint is complete and that the process can be migrated. At the same time as the local checkpoint is completing, a "skeleton process" is being set up at the destination machine which receives the state of the task being migrated. The Spvmd sends the necessary information to the Dpvmd (destination pvmd) to setup the skeleton process. Messages are sent between the Spvmd and Dpvmd to start the actual transfer. Once an acknowledgment from the Dpvmd has been received, the Spvmd sends a message to the local task that the skeleton process is setup and is ready to receive the process state at a specific port on the target machine. On receiving this message, the local task migrates to the destination machine after stopping all its activities thereby freeing the local machine. The

skeleton process, after having received all the state information, reconstructs the task as its own and starts it by placing the relevant data in its address space and registers itself with the local pvmd to become a part of the application again. Registering with the PVM system makes it an MPVM task again.

DynamicPVM [DLVS94] is yet another extension of PVM which deals with problems related to the dynamic scheduling of tasks on parallel machines. It uses Condor's job scheduling mechanism to support dynamic scheduling and also incorporates a checkpointing and migration mechanism. The condor checkpointing code is used for creating checkpoints and a core-dump is created in a shared NFS file system. When checkpointing, the system checks whether the task is going through a "critical section" (In this context critical sections are code sequences where communication is taking place between processes). If so, the checkpointing is delayed until the task comes out of the critical section. This means that during checkpointing, all the tasks which are communicating will effectively agree to the checkpointing and "stop" subsequent communication. The migration protocol used is oriented towards ensuring transparency of the migration process so that the migration of one task will not affect the execution of other related tasks. When a task initiates a checkpoint it sends a message to all other tasks it is communicating with. Upon receiving this message other tasks stop sending any messages to the checkpointing task and wait for an "accept" from the task (which initiated the checkpoint) to make sure that it has migrated and restarted again before receiving additional messages from them. For checkpointing to begin, a new task context is created at the new location by sending messages to the PVM daemon (which is assumed to be already established at the new location). Next the

"routing tables" at the new location and the old location (from where the task is to be migrated) are updated so that all messages are now received by the new location. Finally, the actual checkpoint is taken and written to the disk by stopping the task.

Diskless Checkpointing

Diskless checkpointing [PLP98] is a checkpointing technique that uses memory and checkpointing processors (m spare processors used to store *in-memory* checkpoints taken by the processors running the job, and which can help a system tolerate the failure up to a maximum of m processors) for checkpointing system state and does not depend on the use of stable storage. The main goal of diskless checkpointing is to remove the use of stable storage devices in checkpointing. This eliminates the overhead of writing to stable storage which is a significant source of delay.

Diskless checkpointing [PLP98] is based on co-ordinated checkpointing and assumes that message logs contain the checkpoints of each processor. The two major steps in diskless checkpointing are, first, each processor's state is checkpointed in memory and second, these checkpoints are encoded and sent to *checkpointing processors*. When a failure occurs, the processes which are still running rollback to the checkpoints stored in their own memory. Some *replacement processors* are then selected to replace the failed processors by combining the checkpointed state from the non-failed processors with the encoded checkpoints from the checkpointing processors so the application can start execution again on the replacement processors.

Disk based and diskless checkpointing are compared in Figure 2.9. As shown in Figure 2.9 (a), in disk based checkpointing, the process state, address space and reg-

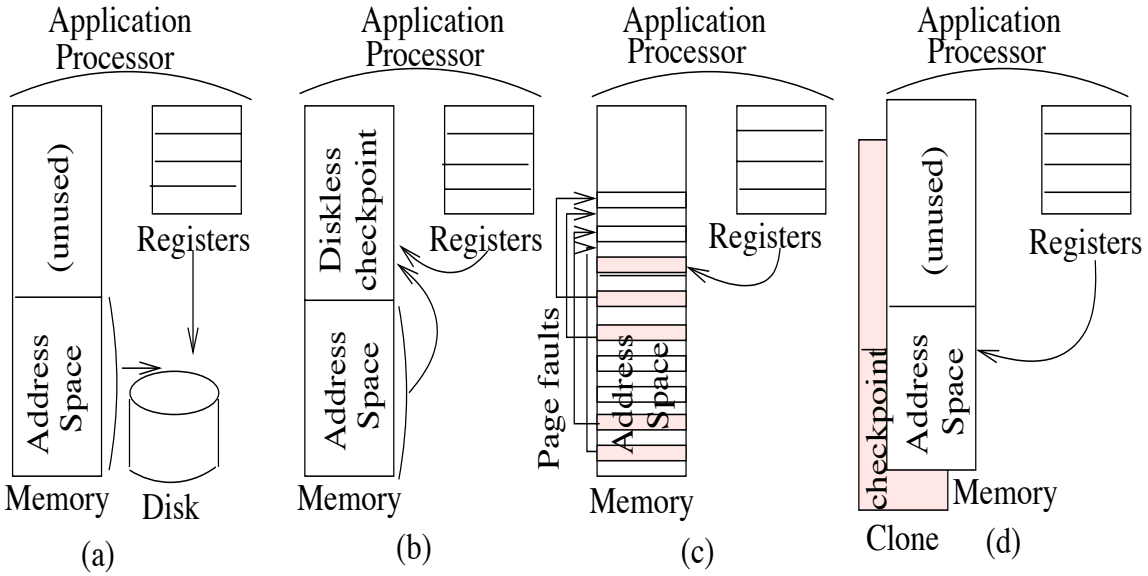

Figure 2.9: Different Checkpointing Schemes (adopted from [PLP98]

isters are stored on a disk and upon failure the processors rollback to the checkpoints stored on the disk. Figure 2.9 (b) shows a simple diskless checkpointing scheme. The address space and registers are stored in the memory itself rather than on disk. When a processor needs to be rolled back, the checkpoints are read much more quickly from the memory itself. The major cost is memory usage and the problem that prevails is that such a system can't withstand failure of the processor itself. Whereas, on the failure of other processors, it can easily rollback to the in-memory checkpoint. Figure 2.9 (c) shows an incremental checkpointing scheme [FB89; WM89], in which application processors mark all the virtual memory protection bits of all the pages as *read-only* in their address space. As described earlier ("copy on write"), an access violation occurs whenever an application tries to write to these pages and the checkpointing system makes a copy of these page contents in memory and marks them as *read-write*. The read-only pages in the address space and the copies of all

the read-write pages constitutes the processor's checkpoint. Whenever a processor has to rollback to a checkpoint the processor just copies the checkpointed copies of all the *read-write* pages from the memory back to the application's address space again marking them as *read-only*. Another scheme which is shown in Figure 2.9 (d) is called forked checkpointing [EJZ92]. This approach is similar to the incremental approach except that the modified pages' identification and page copying is done by the operating system. In this scheme each process makes a clone of itself (via fork) and whenever a failure occurs the address space is replaced by the clone.

Checkpointing in NetSolve

Starfish [AF99] and IBP [PBB+01] are used in combination for providing a checkpointing framework for NetSolve [CD96; AP00]. NetSolve is a system that attempts to use widely distributed computational resources for solving computational problems at a high level of programming abstraction. Unlike many other systems, it is oriented towards the use of existing applications remotely, not the creation of new distributed ones.

Starfish [AF99] is a checkpointing library used for migration and fault-tolerance in Message Passing Interface (MPI) [GLDS96] applications based on the use of core dumps. Starfish assumes that both high speed communication technology and a checkpoint/restart recovery mechanism in a cluster based MPI environment are available to build on. The checkpoint/restart mechanism is very flexible as Starfish provides both coordinated as well as uncoordinated checkpointing depending on system or application functionality. Starfish supports the delivery of an event (which signifies

the failure or crash of an application process on some node) to all surviving processes related to the crashed application process or node. Such an event is a notification provided by the system to inform all the participating nodes about the change in the cluster. Upon learning of the failure of a node, surviving application processes re-partition the dataset on which the processes are executing and all processes subsequently execute on the re-partitioned dataset without any further interruption. Thus, when a process on a node fails, Starfish may either restart the process on a different node by using its checkpoint/restart mechanism or it can dynamically restructure the computation.

The Internet Backplane Protocol (IBP) [PBB+01] provides "Logistical Networking" allowing applications to control the data path and storage of data in the network on that path for large scale distributed applications. The Starfish library is used by

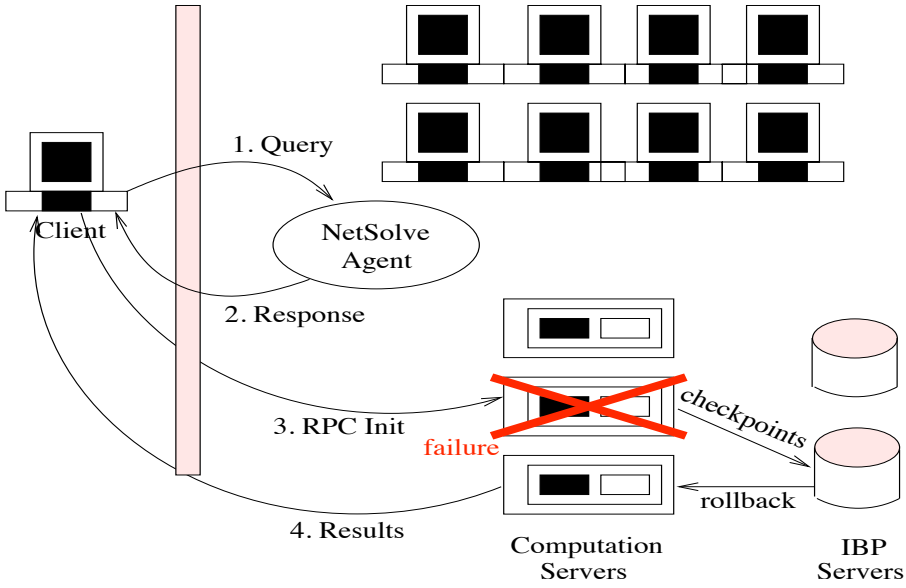

Figure 2.10: NetSolve with Checkpointing (adopted from [AP00])

the NetSolve servers to checkpoint their state in IBP buffers [AP00]. As shown in Fig-

ure 2.10, the NetSolve client contacts a NetSolve agent which has all the information about the available computational servers. The client sends a query for computational resources and gets a response consisting of a list of suitable computational resources. The computation is started on the computation servers and regular checkpoints are stored in the IBP servers. If there is a failure then another computational server takes over the failed computational server's position and starts execution using the checkpointed data in the IBP servers.

CoCheck

CoCheck [Ste96] is a transparent parallel application checkpointer for a Network of Workstations (NOW). CoCheck is implemented as a part of tuMPI [Ste96] (which is a native MPI implementation). CoCheck sits over the MPI library and uses special "ready" messages to clear inflight communication messages when a checkpoint is initiated. A tuMPI application consists of application processes and a daemon process. The daemon process provides a starter which starts the application processes, a locator which tracks the addresses of all the application processes and a grouper which maintains MPI groups and contexts for the application. CoCheck creates a consistent state for process checkpoint and migration before checkpointing. Once the migration notification is initiated the migrating processes send a ready message to all the processes which are communicating with them. The information about the communicating processes is fetched from a local mapping table (Each application process maintains such a table). All the application processes that receive the ready message from the migrating process(es), coordinate with the migrating process(es) to reach a

consistent state and upon reaching such a state, the migrating processes use Condor's checkpoint and migration system to checkpoint and migrate to a new node.

Job Pause in LAM/MPI+BLCR for Fault Tolerance

Job Pause [WMES07], provides a fault tolerance service for LAM/MPI using the Berkeley Lab Checkpoint/Restart (BLCR) system [DHR02] (described in more detail in Section 2.4.3). Job Pause provides transparent checkpoint and migration of processes for just those processes that are on failed nodes. The main behind the work is to avoid complete restart of long running applications in a large execution environment and to facilitate the replacement of just those nodes that have failed. All the MPI processes that reside on live nodes are paused using the utility *cr_pause* and rolled back to the most recent checkpoint. The failed node is then replaced with a spare node and the process(es) on the failed node are restarted by rolling back to the most recent checkpoint as well.

This work is closely related to the work presented in this thesis except that it is focused only on fault-tolerance in a dedicated cluster environment and checkpoint and migration is facilitated at the cost of rolling back to the most recent checkpoint and losing all execution progress between the checkpoint and the failure point. Also, when the checkpoint and migrate is happening, job pause stops the entire application and no progress is made during recovery process. The work that I present in this thesis is focused on increasing the performance of MPI applications by partially checkpointing and migrating those MPI processes from slower running nodes to faster running nodes without stopping the unaffected processes.

Checkpoint/Migrate in Grid

There are two general approaches used in grids to checkpoint/restart applications, namely Application Level Checkpointing (ALC) and System Level Checkpointing (SLC). ALC requires modification of applications to include code to checkpoint the application state (critical program variables and data structures). SLC is independent of such application modifications, since checkpoint/restart is done external to the application. In SLC, processor registers, stack, memory, etc. are saved as part of the checkpoint. One representative example of each type is discussed from among several such systems. GridCPR WG is an ALC type checkpoint/restart facility for use in grids and XtreemGCP is an example of SLC type checkpoint/restart in grids.

The GridCPR WG Checkpoint/Restart System

GridCPR WG [SSK05] (Grid Checkpoint and Restart Working Group) provides a GridCPR (Grid Checkpoint Restart) API to be used by programmers to write applications with checkpoint/restart support for grids and acts as an interface between the application processes and the GridCPR system. Once an application is written using the GridCPR API, the application can checkpoint at suitable points in its execution by making a GridCPR library call to store and retrieve checkpoint information. GridCPR provides APIs for application state writing and reading, checkpoint data management, failure/event notification, and checkpoint data transport, among others. These APIs can be used both by the user application and the administrative tools of GridCPR system for execution of an application. The state management (SM), checkpoint transfer (CT) and event handling (EH) services provide support for

storing, querying, and managing information about the whereabouts of checkpoint data, transfer of checkpoint data from one point to another and registration of failure and notification services, respectively.

Once job execution starts an application takes checkpoints using embedded GridCPR library calls to checkpoint application state at appropriate points in execution. This checkpoint data is then managed by the SM services. Any failure is detected by the EH services and notification is sent to the appropriate GridCPR component. A failed job can then be terminated and re-queued for automatic recovery and restart. The most recent checkpoint locations are queried from the SM and read and, if required, transferred to the new execution location(s) and re-launched from the checkpointed state. This provides fault tolerance which is important for long-running applications. GridCPR is not used for performance management via migration.

The XtreemGCP Checkpoint/Restart System

XtreemGCP [MSRSM09] (grid checkpoint/migrate service) is an integrated grid checkpoint/migrate service in XtreemOS used for migration and fault-tolerance of grid applications. XtreemOS is an EU-funded Linux-based open source grid operating system (OS) that supports distributed heterogenous resources. In such an environment, checkpointing and restarting of applications must be system-specific due to variation in checkpointers available on the variety of platforms supported as grid nodes. XtreemGCP provides application checkpoint and restart using system-specific checkpointer support. XtreemGCP uses a common kernel checkpointer API which is implemented by including customized translation libraries for different system-

specific checkpointers. Currently XtreemOS includes translation libraries for the BLCR [DHR02] and LinuxSSI [MSSMM07] checkpointers.

The layered architecture of XtreemGCP has a job checkpointer (JCP) at the grid level which has a broad view of an application (job) that crosses several grid nodes and which manages checkpoint/restart of the entire application. JCP uses the services provided by job-unit checkpointers (JUCP) that reside on each grid node. The JUCPs control the local checkpointers available on the node they reside on. JUCP uses the common kernel checkpointer API and its customized translation libraries to translate between the grid semantics and the checkpointer semantics to achieve checkpoint and restart of individual application processes (job-units).

CRAK

CRAK [ZN02] is a system initiated checkpoint and restart facility for Linux processes. CRAK is implemented as a Linux kernel module to provides a checkpoint/restart facility for linux processes. CRAK operates in both user-space and kernel-space. The user-space processing is used to identify the processes that need to be checkpointed and all their related information (accessible from user-space) such as file descriptors and pipes. Once the processes to be checkpointed are identified they are sent a SIGSTOP signal to suspend them. The kernel-space processing then picks up and invokes the necessary code to checkpoint the state (which is consistent). After the checkpoint is complete, the user processes may be continued or terminated depending on need.

During restart, CRAK opens each checkpoint file and re-establishes the process by

establishing the access modes, files, structures, etc. It restores sockets in three phases, by creating them in user-space, by re-establishing the local socket data structures to reflect the original endpoints in kernel-space and then by re-establishing the remote socket data structures to reflect the restarting address.

Berkeley Lab Checkpoint and Restart (BLCR)

The Berkeley Lab Checkpoint and Restart (BLCR) facility [DHR02] is a Linux kernel-level process Checkpoint/Restart System. BLCR is implemented as a Linux kernel module and a shared user library. BLCR is capable of checkpointing as a stand-alone system for single-node applications. It can also be used by a parallel communication library to checkpoint and restart parallel applications running on multiple computers [SSB+03]. In LAM/MPI, a *blcr* plugin can harness the knowledge of *mpirun* process of an MPI application, which also acts as the startup point for checkpointing. The *blcr* plugin is the only currently supported checkpointer in LAM/MPI and the BLCR module gets initialized by *mpirun* within LAM/MPI, BLCR accepts the process id of the *mpirun* process to start the checkpoint of an MPI application. The process topology of the entire MPI application is fetched from the *mpirun* process's address space and then the relevant LAM daemons on each remote machine are instructed to initiate the checkpointing of their MPI processes which are identified to be part of the MPI application to be checkpointed. All MPI processes then coordinate with *mpirun* to reach a consistent state and then the checkpoint is written to a single file.

During restart, BLCR reads the saved checkpoint file and rebuilds the MPI ap-

plication by starting a new *mpirun*, which in turn builds *all* MPI processes from the application schema (describing the MPI-application processes) saved in the checkpoint file. The topology of the processes must be the same as before but the machines involved in the topology may be different (i.e. a different "mpihost" file may be used). When the MPI processes are re-started, they re-establish their TCP Socket connections for communication and then the MPI application begins its normal execution again.

2.5 Limitations of Existing Checkpoint and Migrate Schemes

Several checkpoint and migration schemes have been presented. It is evident that application level checkpointing puts undue burden on the users of the system. One example of such a system is mpC, which requires explicit coding in the application programs at the places where checkpoint and migration has to happen. Further, MPI is the predominant message-based parallel programming environment, so efforts towards improving the performance of applications in as MPI environment will be most valuable and have greater impact than similar systems for less widely deployed parallel programming environment.

The checkpoint and migrate schemes presented in this chapter are mostly focused on the reliability of the system rather than on performance. Particularly when seeking to make use of shared distributed computing resources, care must be taken to ensure the performance of the execution environment, which may vary due to variation in

the load on the participating nodes. All the systems described help facilitate execution of long running application in a distributed environment by, at least, providing some measure of fault tolerance. With shared resources with unpredictable load, partial checkpoint and migration is needed to ensure good performance by dynamically adjusting the virtual machine to provide ongoing load balance.

When several processes are involved in solving one large problem and if the performance of the entire application is degrading due to a single slow node then it is wasteful to checkpoint and migrate the entire application to a new location or even checkpointing the entire application to just replace the slow node. A partial checkpoint and migrate mechanism is needed since not all processes are affected and others should continue to execute and make progress.

Chapter 3

Overview of LAM/MPI - a basis for parallel computing

Distributed Memory High Performance Computing became popular among researchers needing to solve compute intensive problems because of the strong support of the Message Passing Interface (MPI) standard, which has now become the *de facto* standard for message passing parallel programming, particularly on distributed memory parallel systems [SL03].

3.1 LAM/MPI Overview

LAM/MPI is an open source implementation of the MPI Standard [SL03] for message passing based parallel programming consisting of a layered component architecture. As shown in Figure 3.1 the LAM layer provides a Run Time Environment (RTE) and the MPI layer which is presented to the programmer supports communi-

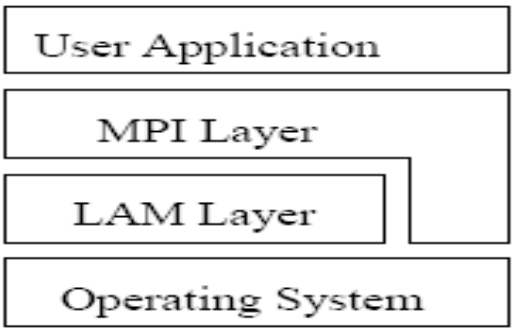

Figure 3.1: LAM/MPI Architecture (adopted from [SL03])

cation between the processes. The LAM layer sits between the Operating System and the MPI layer, but both LAM and MPI layers are able to interact with the Operating System for the purpose of performance gain. The LAM RTE can be used independently without the existence of the MPI layer but, MPI can not be used without LAM. MPI uses LAM functionality for setting up the communication environment during initialization. LAM, apart from providing the RTE also exports some basic API's in the form of a library to allow interaction with the RTE, which can be used by programmers for "out-of-band" communication, process control, I/O control, etc. The out-of-band communication channels are also used for setting up the initial communication environment as well as for coordination of the processes at the lower level. In contrast the MPI communication layer exports all its APIs for use by the application programmer in the form of library. As both the LAM and MPI layers are structured as component architectures, it is relatively easy to add plug-ins to enhance functionality, as shown in Figure 3.2.

The component architectures of LAM and MPI makes it fairly easy for programmers to develop their own component types and plug-ins for the LAM and MPI en-

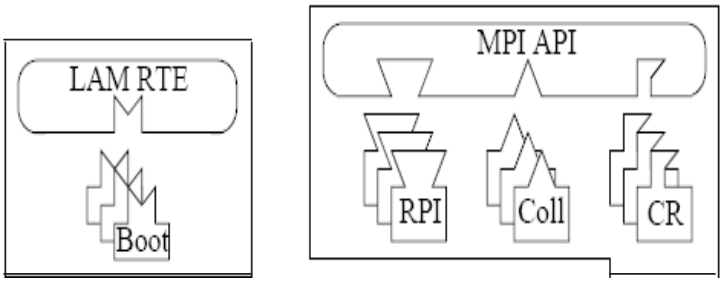

Figure 3.2: Plug-ins in LAM/MPI (adopted from [SL03])

vironments. The currently existing components, formally known as System Software Interfaces (SSIs), are listed in Table 3.1.

Table 3.1: System Software Interface in LAM/MPI (adopted from [SL03])

SSI Name	**Available in**
boot	LAM
rpi	MPI
coll	MPI
cr	both LAM and MPI

3.1.1 The System Software Interface (SSI)

The LAM/MPI System Software Interface (SSI) is a collection of system interfaces designed as a two-tiered modular framework, where tier one represents the SSI type itself and tier two consists of several modules or instances of that SSI type. For example, *boot* SSI is the type and *bproc, globus, rsh,* and *tm* are the existing instances, or modules, of the type *boot* available in LAM. These available instances can be selected during initialization of LAM and MPI.

LAM provides the run-time backbone to MPI, the functionality of the MPI com-

munication layer depends on the proper function of the LAM RTE. Naturally, LAM SSI types can only use services that are exported by the LAM layer, but MPI SSI types can use services exported by both the LAM layer and the MPI Layer.

The *boot* [SBL03a] SSI type is the back-end SSI used to boot the LAM RTE on remote nodes prior to the existence of *lamds*[1] . Using the *boot* SSI modules, LAM first setup its RTE and subsequently starts the *lamds* on the nodes included in the RTE. LAM must use OS provided facilities such as *rsh* and *ssh* to launch its *lamds* on remote nodes. The *tm boot* module was developed to boot LAM on the PBS Batch Queue Systems [Sys10], the *globus* boot module was developed to start a LAM RTE on Globus [FK97] enabled systems and the *bproc* module is used to establish a LAM RTE on BProc [Hen02] systems.

The Request Progression Interface (*rpi*) [SBL03d] SSI is used by the MPI layer for Point-to-Point communication. It is actually the lowest level of message passing supported between any two MPI processes and manages how the messages are passed in terms of bytes sent over the network. TCP and shared memory modules were the initially created RPI SSI modules, followed by *gm*, which provides support for Myrinet [BCF+95] networks and *crtcp*, which helps provide support for checkpointing and restart of a process (discussed in further detail later).

The MPI Collective Communication Component (*coll*) [SBL03b] SSI is the MPI layer level SSI used to provide MPI collective communications (supporting operations such as broadcast, scatter, gather and all-gather). A key concept behind the

[1] *lamds* are LAM daemons that reside on each virtual cluster node and help in managing the MPI execution environment by monitoring and performing housekeeping jobs such as cleaning-up an MPI application if a user aborts execution, etc.

development of this SSI was to allow different MPI "communicators"[2] to use different collective communication operations algorithms. In addition to the *lam-basic* collective communications algorithm modules, there are now *smp*-based algorithm modules.

The Checkpoint/Restart Component (*cr*) [SBL03c] SSI type is a recent addition to the functionality of LAM/MPI and is of direct importance to this thesis. It allows the parallel MPI applications to co-ordinate and checkpoint any time between MPI_INIT and MPI_FINALIZE calls, and to later restart the same parallel MPI application on the same or a different LAM RTE (i.e. set of machines) using the checkpointed data without having to start the application from scratch. The *cr* SSI basically provides two functions: checkpointing of the parallel application using, currently, the only existing back-end checkpointer *blcr*, and co-ordination with all other SSI modules to agree on checkpointing themselves to allow later restart of the checkpointed parallel MPI application.

3.2 The LAM/MPI Execution Environment

To fully understand the implementation component of research described in this thesis, the functionality of LAM/MPI must be considered. A programmer who writes a parallel application uses the libraries that are provided by LAM/MPI to develop an MPI application that runs under the LAM/MPI Run Time Environment (RTE). The underlying functionality, how LAM/MPI launches the application and how the execution environment is created, etc. is transparent to the users. Dealing with and

[2]For more details on communicators please refer to Appendix A.

modifying LAM/MPI itself however, requires knowledge of its internal operations. The following sections contain an overview of the LAM/MPI RTE as well as the functionality used when checkpoint requests and restart requests arrive.

3.2.1 The LAM/MPI Run Time Environment (RTE)

The initial setup for the creation of an MPI application is facilitated by the LAM RTE. The *lamds* play the major role in the creation of the MPI execution environment. The *lamds* are the LAM daemons that reside on the set of nodes on which the MPI application will be executed. These *lamds* actually create the virtual cluster machine on which the MPI application is then distributed when the *mpirun* command is invoked (with possible parameters). The process of creating an execution environment for the execution of an MPI application is shown in Figure 3.3.

As shown, in Figure 3.3, when an *mpirun* command is issued the application execution environment is built jointly by the LAM and MPI Layers. The *mpirun* command parses any command line arguments as the first step in building the application execution environment. The arguments are then passed to the *build_app()* function which, upon completion, returns a LIST of the processes that need to be executed depending on what the command line arguments were. This LIST is then passed to the *asc_run()* function which does most of the process creation (on both local and remote nodes). This function passes control to the *lamds* which create the processes on the desired nodes. Once the processes have been successfully created by the *lamds* they each send their *PID* (Process ID of the MPI process on the local node) and *IDX* (IDX is the index of the MPI process in the local LAM daemon's process ta-

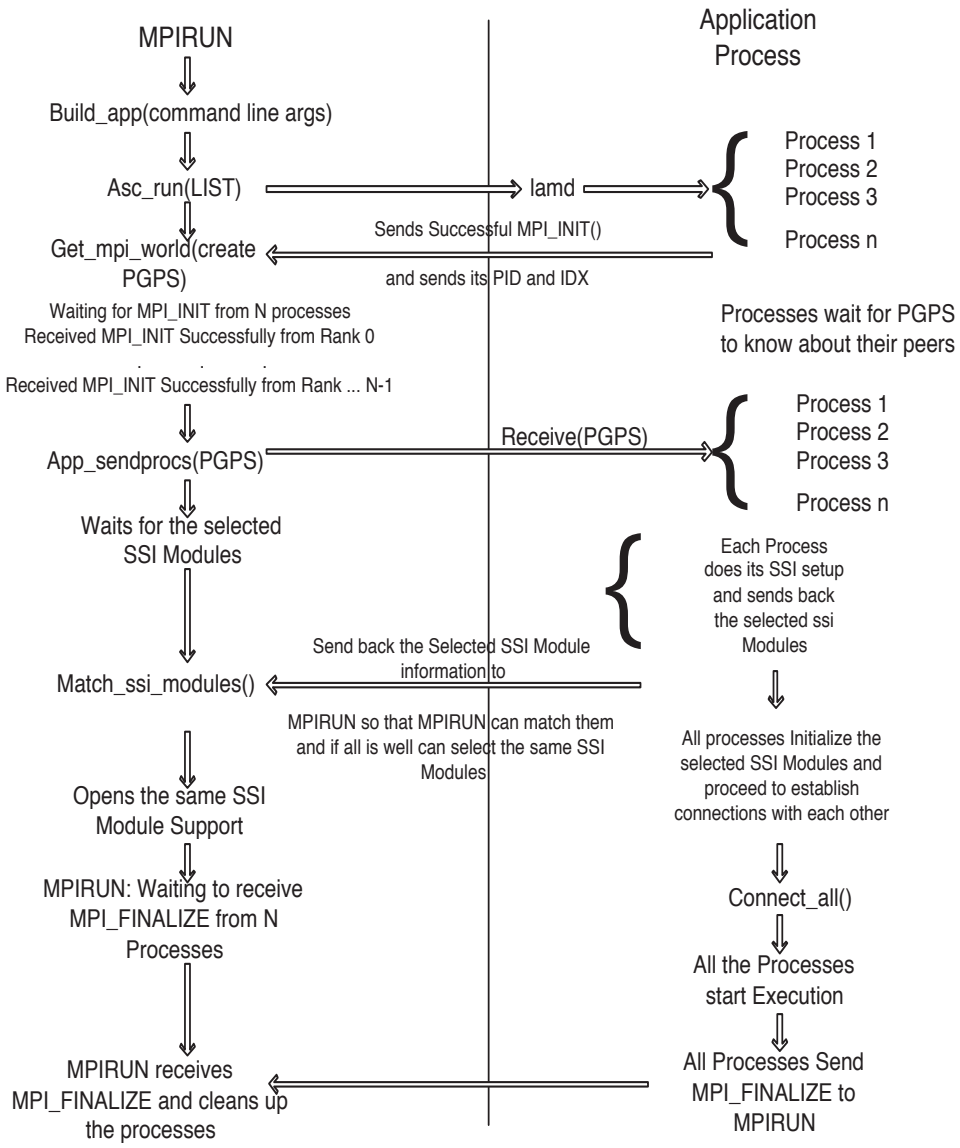

Figure 3.3: The MPI Process Creation Steps

ble) upon successful executions of *MPI_Init()* to the *mpirun* process which is waiting to receive them by calling *get_mpi_world()*. This is used to construct the Processes GPS (Global Positioning System) (PGPS) information, which is then stored in memory in the *_gps struct*. Having built the PGPS, *mpirun* then propagates the PGPS to

each process created using the *app_send()* function, so that each process gets to know about every other process' location. Once each process receives the PGPS sent by *mpirun*, each selects the System Software Interfaces (SSI) modules to be loaded for supporting execution, and then sends the selected SSI information back to *mpirun*. The *mpirun* process which is waiting to receive the selected SSI's from each of the processes, receives them and matches them to check that all of the processes have selected the same SSIs. The *cr* SSI *blcr* plugin is selected at this time and all the SSI's are made checkpoint/restart capable (what is called *cr-aware*). Once the SSI's are checked, a go-ahead is given to the processes and *mpirun* itself opens the selected SSI's to support the application execution environment. The MPI application processes, after receiving the go-ahead, connect to each other, using Connect_all(), creating the necessary communication channels and then proceed with the execution of the application. During execution, there is a single mpirun process and one MPI application process per MPI rank. These processes are distributed over the nodes in the virtual machine cluster. After initialization, the *mpirun* process waits to receive messages sent by *MPI_Finalize()* from each of the application processes. Once all such messages have been received the application execution environment can be shutdown.

3.2.2 Application Checkpointing in LAM/MPI

Checkpointing in LAM/MPI is only possible if BLCR (Berkeley Lab Checkpoint/Restart) has been installed and LAM has been configured to make the *cr* SSI available as a selectable module. The *blcr* plug in, which is available in LAM/MPI in

the form of a *cr* SSI module, is the only *cr* SSI implementation currently available. An application that runs in a LAM/MPI environment is thus made checkpointable when it selects blcr as its *cr* SSI module. When the application execution environment is being built (refer to Figure 3.3) the application selects the specified SSI and registers two callbacks with blcr during the SSI's initialization. These two callbacks are the "thread based" callback and the "signal based" callback. The thread based callback is used for checkpointing the application which can then continue or terminate, as determined by the user. The signal based callback is used to restart the application from a previously created checkpoint. A signal based callback routine/handler is needed for the restart of the application from the saved checkpoint information, because only in signal based context[3] can processes regain their original PID's. Figure 3.4 shows the overall process of checkpointing a LAM/MPI application.

As shown in Figure 3.4, when there is a checkpoint request by a user, which is of the form *cr_checkpoint <PID>*, where PID is the PID of an *mpirun* process, the blcr thread based callback, *crlam_thread_callback()*, is invoked and the checkpointing of the entire MPI application is started. The thread callback function calls the base checkpoint function which, in turn, calls the actual checkpoint function *lam_ssi_crlam_checkpoint()*, which prepares all the processes to checkpoint. The *lam_ssi_crlam_checkpoint()* function forks a child process and waits for the child to return a success indication to proceed with the creation of the "application schema" (which is an ASCII file explicitly describing which programs constituting an application are to run at which locations in a virtual machine[4]). The child process

[3]Unix supports signal based programing, which allows signals to be sent to processes to notify them of events. Received signals are processed by a signal handler routine.

[4]Application schemas permit explicit control over the placement of MPI application processes

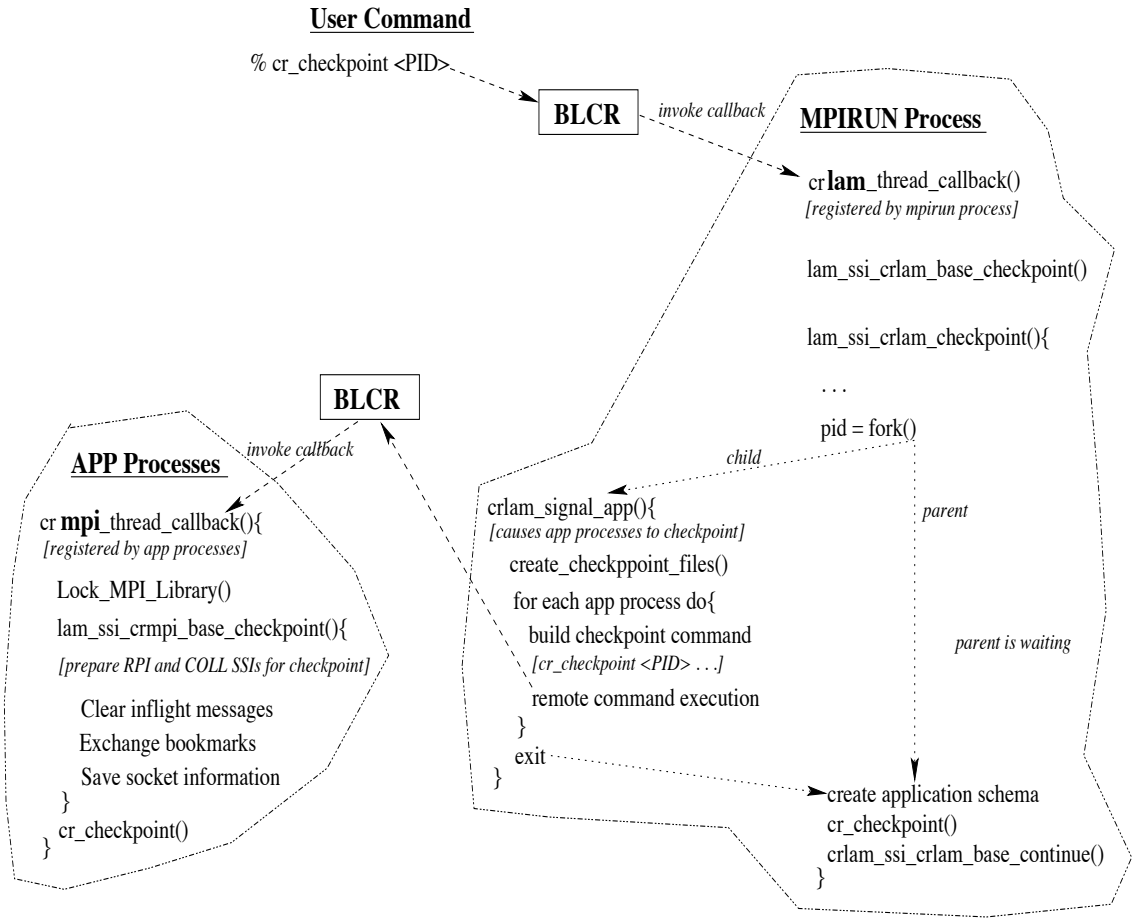

Figure 3.4: Application Checkpointing Steps in LAM/MPI

which is a clone of its parent and thus has all the information that the parent has, signals all the processes described in the PGPS to checkpoint. Note that in Figure 3.4 there are two separate thread callback routines, *cr**lam**_thread_callback()* (for mpirun) and *cr**mpi**_thread_callback()* (for MPI application processes). The *crlam* here refers to the LAM layer callback and the *crmpi* refers to the MPI layer callback. Each layer registers a callback with blcr for checkpoint and restart. Before the *cr**mpi**_thread_callback()* is invoked, the child process prepares the files in which the

and also enable non-SPMD jobs. For more information see the manpage for appschema(5).

checkpoints will be written and the command that is to be invoked on the remote node where the processes are running to accomplish checkpointing of the MPI application processes. (This command is shown in Figure 3.4). The *rploadgov()* function actually invokes this command on the remote nodes and the *cr**mpi**_thread_callback()* picks up from there to perform the MPI application process checkpoint procedure. The child then waits for the *crmpi* thread callback to return success.

*cr**mpi**_thread_callback()* acquires a lock on the MPI library so that no process can make an MPI library call during checkpointing. It then signals the RPI and COLL SSI's to prepare for checkpoint. This is done to clear any in-flight messages. Each process sends a "bookmark" to every other process and receives a bookmark from each and every process except itself. These bookmarks represent the amount of data sent between two processes in bytes and this information is exchanged using out-of-band communication channels provided by the lamds. Once the bookmarks are exchanged, the processes then figure out how much data each has to read from the in-band communication channels to clear all in-flight messages and they proceed with reading the necessary data. After the in-flight messages are cleared the RPI and COLL modules declare that they are ready for the checkpoint and save the socket information associated with each process. The *crmpi* thread then proceeds with actually writing the MPI application process checkpoint files by calling *cr_checkpoint()*. Control is then returned to the waiting child process of mpirun which then renames the files and unlinks them[5] and returns control to the parent. The parent updates the application schema by incorporating the process checkpoint filenames into the

[5]This assumes a shared file system such as NFS across cluster nodes. A system such as GPFS [SH02] could also be used to provide wide area network sharing of files. In the absence of any shared file system, a mechanism for the explicit transfer of files can be envisioned.

in-memory schema and then takes a checkpoint of the *mpirun* process itself. Once the checkpoint of *mpirun* succeeds the callback returns control to the *mpirun* process by calling *lam_ssi_crlam_base_continue()*, which can then continue as if nothing has happened or be terminated by the user for future restart.

An MPI application which is checkpointed in one LAM RTE (the set of nodes used to create virtual machine) can be successfully (re)started in a different LAM RTE (completely different set of nodes) from the checkpointed data [SBL03d]. This is because, the LAM RTE is not a part of the checkpoint. When a checkpoint of an MPI application is taken, only the MPI universe (which includes all the application processes, the *mpirun* process and the selected SSI's) is checkpointed.

3.2.3 Application Restart in LAM/MPI

An MPI application can be (re)started in two ways: using the actual executable file for the MPI program (which is how a user typically starts an MPI application; the "normal" way) or by using an application schema, which is what is done when an MPI application is restarted from its checkpointed state (stored in the checkpoint files).

Restart of an MPI application from the checkpointed files is achieved by issuing the command *cr_restart context.<mpirunPID>*, where context.mpirunPID is the checkpoint file (containing the application schema within the process image) of the *mpirun* process. BLCR loads the process image into memory. Once the image is loaded the signal based callback is invoked by the *blcr* plugin by calling *crlam_signal_callback()* which, in turn, calls the base restart function and passes the name of the executable

and the application schema to the function. In this case, the executable name is *mpirun* and the application schema is the schema that was written as part of the file context.mpirunPID. The *create_restart_argv()* function is then invoked which creates the restart command to restart the *mpirun* process from its checkpointed state. The *do_exec()* function then restarts the *mpirun* process using the parameters built (as shown in Figure 3.5).

Once the *mpirun* process is restarted from its checkpointed state, it then builds restart commands for all the application processes using information from its state and stores the necessary commands in a temporary file named *schema.xxxxxx*. Each line in this file is a *cr_restart* command meant to be invoked on remote nodes by the *mpirun* process to restart each MPI application process on that remote node from its checkpointed state. An example of such a line in the schema file is n# <path>/cr_restart <path>/context.mpirunPID-n#-processPID where, n# is the node on which the process is to be restarted, <path>/cr_restart is the fully qualified BLCR *cr_restart* command pathname, and <path>/context.mpirunPID-n#-processPID refers to the checkpoint file (containing the MPI application process state) for the process with process ID, *processPID*. The checkpoint file name context.mpirunPID-n#-processPID signifies that PID, *processPID*, is the PID of the process belonging to an MPI application managed by the *mpirun* process with PID, *mpirunPID*, that was running on node *n#* before the checkpoint. After building the *schema.xxxxxx* file, the *mpirun* process uses this file as an argument (in place of the pathname of the MPI program executable) to remotely invoke restart commands for each process to restart them on the remote nodes from its checkpointed state.

As shown in Figure 3.5, the only difference between a normal *mpirun* execution and a restart of an application from a checkpoint is that starting the application is done by naming the application executable instead of an application schema. In the first case the command that *mpirun* invokes on remote nodes is, for example, *mpirun application-executable* and in the latter case it is *cr_restart context.mpirunPID-n#-processPID*.

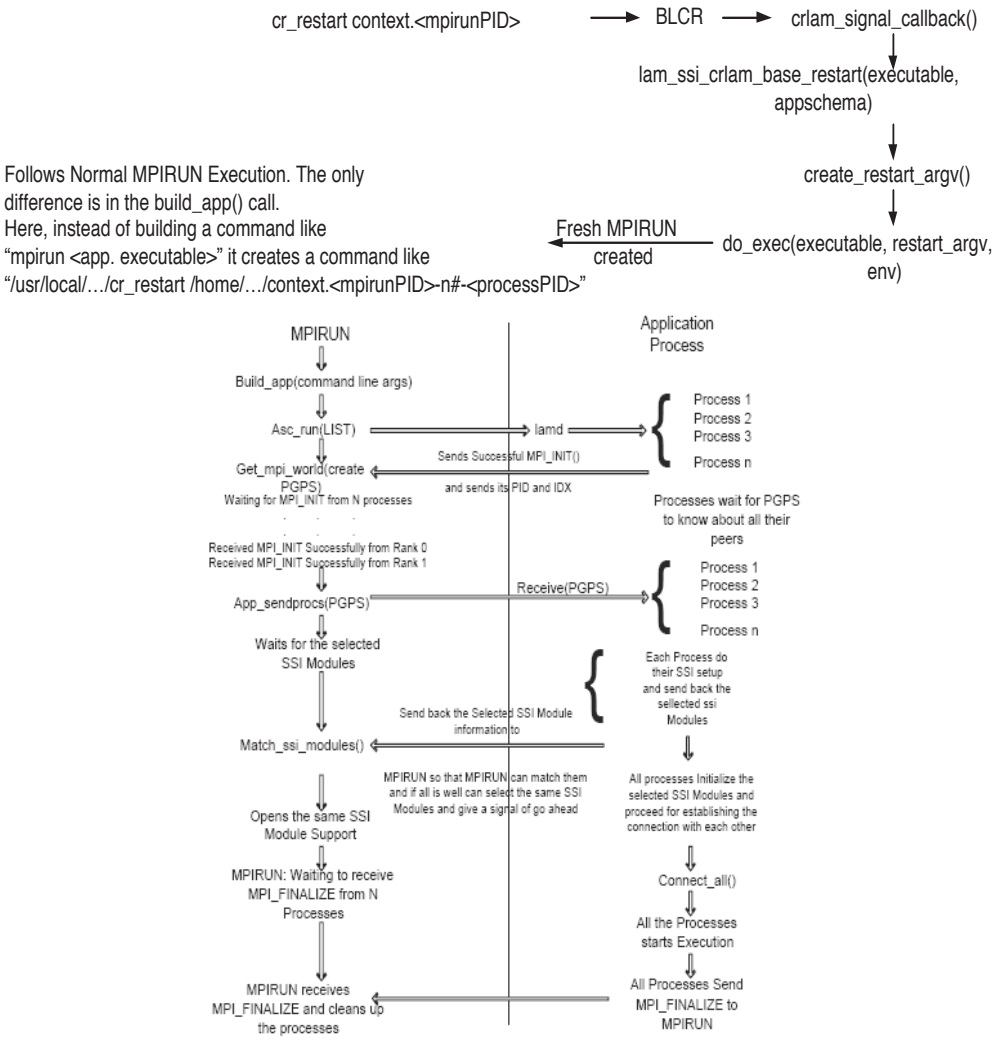

Figure 3.5: Application Restart Steps in LAM/MPI

3.3 Limitations of Current Checkpoint/Restart in LAM/MPI

LAM/MPI uses the *blcr* plugin as the back-end checkpointer. At present, this allows it only to take a "complete checkpoint" of the MPI application. To do this, at the time of taking a checkpoint, the entire MPI application (all the processes that are part of the MPI application) is stopped and the checkpoint is written to stable storage. During this time, no progress is made towards application completion. After the checkpoint is taken the application can resume execution or be terminated. This approach to checkpointing is designed for fault tolerance, so a checkpoint of an MPI application is always available to be rolled-back to, in case of a fatal malfunction (hardware or software). In the case where the performance of an entire MPI application is being affected due to a slow running node, this approach of complete checkpointing is inappropriate. This is because, to avoid performance loss due to a single node running slowly we would have to take a complete checkpoint and restart the MPI application on a completely different set of nodes (a new virtual machine). During this checkpoint and restart there is no progress made towards the application completion even by unaffected nodes because the entire application is stopped.

Consider the example shown in Figure 3.6 which shows an MPI application with three processes running on a LAM RTE consisting of three host machines {H1, H2, H3}. If the node $H2$ is overloaded and running slowly, then it will affect the process (represented by the circle) running on that node and subsequently other processes running on nodes $H1$ and $H3$ if they are communicating with the process on node

$H2$. Currently, LAM/MPI would take a complete checkpoint of the MPI application (see Figure 3.6), which will require stopping all the processes until a new LAM RTE is created and the entire MPI application can be restarted from the checkpointed state on that LAM RTE. This entire process results in no progress being made towards application completion. On the other hand, if we can detect that it is only $H2$ and subsequently only the process on $H2$ that is running slowly, then we could just checkpoint the process on node $H2$, remove node $H2$ from the LAM RTE, add a new node $N1$ and restart the checkpointed process on $N1$, as shown in Figure 3.6. During the time the affected process is being checkpointed and restarted, other processes running on other nodes ($H1$ and $H3$) would still be executing and making progress towards application completion (assuming that there is no communication between those processes and the one being checkpointed and restarted).

LAM provides two utility functions *lamgrow* and *lamshink*. The routine *lamgrow* can start a *lamd* on a single node at a time and make it a part of an existing LAM RTE. The routine *lamshrink* can remove a node from an existing LAM RTE. These routines could be used to alter the LAM RTE in a way that would support partial checkpoint/restart (e.g. of node $H2$ in the previous example).

One key goal of my thesis research is to be able to checkpoint a single process from a running parallel MPI application environment and restart it from the saved checkpoint on a different node, as shown in Figure 3.6. All MPI application processes perform some kind of message passing during their execution. Broadly, MPI applications can be divided into categories depending on their message passing (communication) behaviour. One category, which consists of compute intensive processes

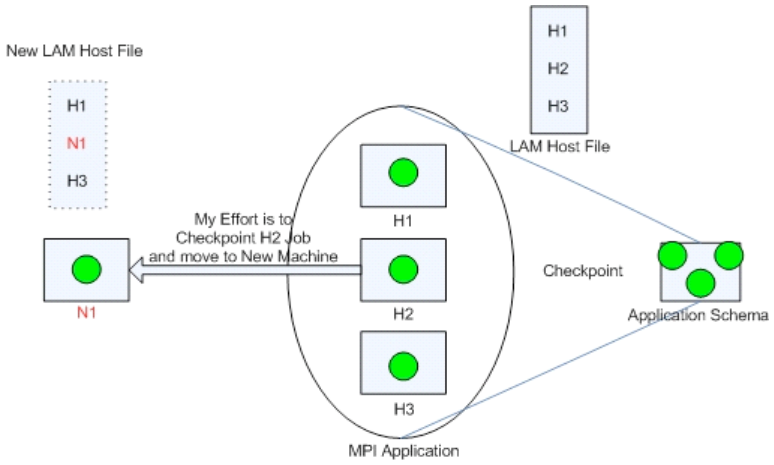

Figure 3.6: Checkpoint and Restart Single Process using *blcr* in LAM RTE

with little communication between processes and another which consists of communication intensive processes that require frequent communication with other processes represent the end points of a spectrum of possibilities.

To checkpoint and restart a single process of an MPI application that is less communication intensive is relatively easy because of the independence of processes from each other. On the other hand, it is quite tricky (and possibly, unproductive) when the processes are communication intensive. In Figure 3.6 if the process on node $H2$ was communicating with the process on node $H3$ at the time the process on node $H2$ was getting checkpointed (node $H2$ replaced by $N1$), the process on node $H3$ would have to wait for the process on $H2$ to reply. Currently, in LAM/MPI, this situation would lead to a malfunction and *mpirun* would terminate the entire MPI application thinking that there was a broken communication channel between processes. Certainly, if we want to be able to checkpoint a single process from an MPI application we need to avoid the situation where temporary lack of communication

leads to application termination.

Further, we must know that the process on node $H3$ was communicating with the process on node $H2$. LAM/MPI does not provide any built-in facility to determine who is communicating with whom but it does maintain some information about how much a process has communicated with other processes at the RPI level. This is used to clear the in-flight messages at the time of checkpoint. Collection of such communication statistics between processes is required, to be able to determine who is communicating with whom at the time of single process checkpointing (in general, partial checkpointing), so that frequently communicating processes can be grouped together and checkpointed, migrated and restarted together. Coming back to the same example as above, if we can determine that the process on $H2$ and the process on $H3$ communicate with each other at the time that the process on node $H2$ is being checkpointed, then we can group and move the processes on both nodes $H2$ and $H3$ together to maintain communication efficiency.

Chapter 4

Motivation and Problem Definition

Using LAM/MPI in a conventional (dedicated cluster) environment with a well designed parallel algorithm, good performance can normally be achieved. When using LAM/MPI in a shared distributed computing environment, however, performance can be affected due to slow running machines that are part of the MPI virtual machine. This slow behavior can be caused by unexpected overload on the machines due to, for example, a long queued job beginning to execute or some other additional unforeseen computation activities starting on the machine. The network that connects the machines taking part in the shared distributed computing environment may also get overloaded due to unanticipated data transfer by other applications consuming significant bandwidth, forcing the MPI processes to wait for messages they send/receive. Checkpointing and migrating an affected application to a more lightly loaded execution environment is a viable option to deal with this kind of performance loss. As described in Chapter 3, LAM/MPI allows checkpoint/migrate of an entire MPI application (i.e. a complete checkpoint) but this is not always a practical approach. Shared

distributed high performance applications can be large, involving 100's or even 1000's of processes distributed over similarly large numbers of machines and working on large data (ranging from gigabytes to possibly terabytes in the near future) to solve a single parallel problem. A complete checkpoint of such an application would be very large and also very expensive, in terms of time taken to migrate the checkpointed data. Further such a complete checkpoint could actually result in lost performance since during checkpoint/migrate of such an application there is no progress made in execution. If only a part of an MPI application is experiencing performance loss then it makes no sense to checkpoint/migrate the entire application. It must be possible to checkpoint only those processes that are performing poorly and migrate them to faster machines without affecting the other processes' execution. That is, we need to be able to do *partial* checkpoint/migrate of an MPI application.

One might argue that conventional dedicated clusters better meet the performance requirements of an MPI application, but, unfortunately, they are not always a cost effective choice for many researchers. The availability of shared distributed machines over wider area networks and the improving network connections between them make them an attractive alternative for some types of high performance computing because it is extremely cheap to build a virtual machine using such resources. As long as they can be programmed with existing widely-used cluster software (e.g. MPI) they offer a potentially useful platform on which to run coarser-grained parallel applications. Unexpected load on such shared distributed resources introduces challenges however, that can potentially be met using partial checkpoint/migrate of MPI applications.

To achieve partial checkpoint/migrate the following seven steps are necessary. In

this thesis I address steps 2, 3, 5 and 7. The remaining steps are the responsibility of the execution environment and their existence is assumed in my research.

1. Identifying poorly performing compute nodes and the MPI application processes running on them.

2. Identifying groups of tightly communicating MPI application processes.

3. Selecting groups of MPI application processes to migrate, consisting of the poorly performing processes together with those they communicate with frequently (since leaving such processes where they are may lead to poor communication performance after migration).

4. Identifying, lightly loaded, new target compute nodes to migrate the processes to.

5. Checkpointing only the selected processes in such a way that other processes may continue to run and make progress.

6. Migrating the selected processes.

7. Restarting the migrated processes and re-integrating them into the LAM/MPI virtual machine.

4.1 Problem Description

High performance distributed computing facilitates the execution of parallel, message passing computational jobs in a distributed environment ranging from a dedi-

cated local cluster to, potentially, grid-scale systems. An end user (scientist, researcher, organization, etc.) who wishes to solve a computationally intensive problem will typically use a high performance distributed computing environment by submitting their application for execution and then waiting for the results. It is the responsibility of the computing environment to execute the job submitted and return the results fault free and as quickly as possible. As discussed, a possible problem that may arise during the course of execution on shared systems is the poor performance of compute resources chosen for execution due to variation in load on those resources. One way to deal with this problem is to detect poor performance and migrate computations (i.e. processes or other parts of an application) away from overloaded nodes onto lesser loaded ones. A challenge in doing this is to ensure that the migration is effective (i.e. that the correct application processes are migrated together) and to minimize disruption to the ongoing computation. This is the focus of my thesis research, as applied to LAM/MPI.

Process migration can be provided only if the execution environment supports a mechanism for discovering suitable sets of machines and a mechanism to move jobs to those new machines. Assuming that resource discovery is supported by the system and that a set of suitable machines are available at some location, the challenging part of the problem that remains is identifying, checkpointing and migrating the right parts of an already running application (those that are performing poorly and others that are tightly coupled to them) to the new set of machines such that the time and overhead of migration is low and hence the performance of the overall job execution is preserved as much as possible. The main focus of my research is to correctly

identify the *parts* of an application to move and to permit this while the other parts continue to execute effectively (using LAM/MPI). Continued application progress during migration and reduced overhead relative to a complete checkpoint/migrate are the benefits of this approach.

To be able to checkpoint and migrate poorly performing processes from a running MPI application, we must be able to identify the poor performers. A simple performance monitor can be used for this purpose. Once poorly performing processes have been identified they can be grouped for checkpoint and migrate with those processes they are known to communicate with frequently.

The most challenging part of partial migration in a distributed environment is to decide what parts of an MPI application should be checkpointed and migrated together. Every parallel job must somehow be divided into several pieces and allocated to a group of machines for execution. Each piece (an application process in MPI) may or may not communicate with other processes at various times. The communication among these processes determines which processes should be migrated together. Those that have frequent inter-communication should migrate as group. If there are two processes that frequently communicate with each other and the system decides to move one of the processes because it was on a machine that was performing poorly then subsequent communication between the processes will be negatively impacted if the two processes end up running on relatively distant machines. Further, leaving one process running is unlikely to provide benefit since it will likely block quickly waiting for communication with the other process which is being checkpointed and migrated. Thus, depending on the communication patterns between processes, one must decide

how to group processes for migration together.

Generally, parallel programs work on large data and often have regular repeated patterns of communication corresponding to repeated patterns in computation. This results in alternating, relatively long periods of computation and communication. In an application that will take days to finish execution, it is reasonable to expect that for, perhaps, 10's of minutes the processes will have consistent computation and communication patterns at a time. This determines the time frame for the collection of communication information used to form groups of frequently communicating processes. Thus, if communications data can be collected at appropriate intervals, changing application communication patterns should be able to be tolerated. To analyze the collected communication information and come up with meaningful patterns of communication some pattern recognition algorithms (discussed in later chapters) might be used.

For determining the communicating groups of processes we need to know the communication behavior of the MPI application processes. This can be achieved by collecting communication statistics for each process relative to every other process from the start of application execution to the time when we need to determine groups of communicating processes. Given this information (history of communication between processes), we can search for patterns of communicating processes that have repeated over time and that are expected to re-occur. To avoid considering old patterns of communication that no longer apply, the collected data can be divided into sampling periods and patterns of communicating processes can be found in the recent sampling periods. Also summarized pattern information could be retained and the

raw communication information discarded to avoid unnecessary memory overhead. The collected communication information provides the basis upon which expected future patterns may be predicted. This prediction can then be used to group processes for checkpoint and migration. A natural question arising from the need to collect communication information is that of accuracy of the information collected which determines whether the detection of groups of processes that communicate with each other at checkpoint time, will be accurate enough. Determining such groups consisting of processes expected to be communicating frequently with each other is critical to a successful partial checkpoint/migrate strategy.

Predicted groups of processes need to be checkpointed/migrated and restarted on their new nodes without affecting other processes' execution. Currently, when a process dies or stops responding the LAM daemon *lamd* residing on the node of the stalled process informs *mpirum* about the malfunction and *mpirun* terminates the entire MPI application. When we checkpoint/migrate *part* of an MPI application, *lamd* must not cause the application to be aborted since we know the migrated processes will restart in the future. This will allow the other MPI application processes to continue execution.

It will also be necessary to extend LAM/MPI to allow the migrated processes to be re-integrated into the ongoing computation. This will involve establishing new (socket) communication channels and then unblocking processes waiting for communication with checkpointed processes.

Finally, an assessment of the conditions (application and network) under which partial checkpointing will be effective, needs to be undertaken.

Chapter 5

Factors Influencing Partial Checkpoint/Migration

In this chapter, I temporarily set aside the focus on LAM/MPI to consider the general issues associated with doing partial checkpoint/migrate. There are three main factors influencing the design of any partial checkpoint/migration system, as shown in Figure 5.1. These are the computation model, unit of program modularity and the independence model, as described shortly. Though, all three factors (and axes in Figure 5.1) are in some ways dependent on each other, my goal is to consider them as independently as possible to see how they might affect partial checkpoint/migrate.

I begin this discussion with a few fundamental definitions:

Process: A process is a program executing in a self-contained environment (provided by an operating system) which includes all the resources that are required by the process. Generally, processes are considered to be "heavyweight" units of execution. Because they need their own address spaces, context switching between processes is

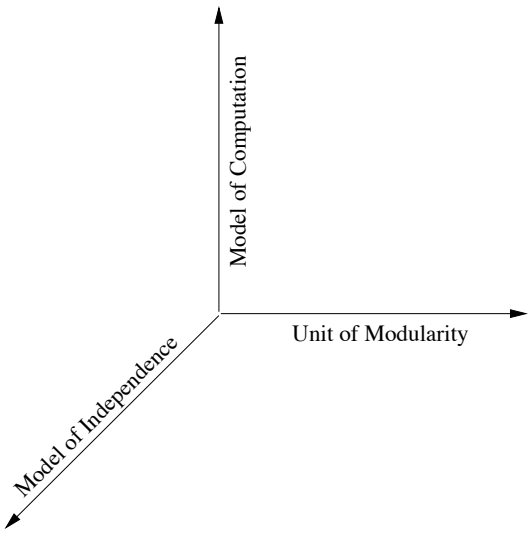

Figure 5.1: Factors affecting Partial Checkpoint/Restart Design

expensive. Further, if processes need to communicate with each other, it is relatively expensive because of the need to cross address space boundaries. In a process based environment, a process is the smallest unit of execution. If multiple tasks are to be performed then either they must be performed sequentially (if only a single process is executing) or a separate process is required for each task.

Thread: Threads are lightweight execution units. Related threads share the same address space which makes context switching between them inexpensive. Inter-thread communication is also inexpensive due to the fact that they share memory. If multiple tasks are to be performed, a program can often be written in such a way that multiple threads can work on the different tasks simultaneously. When accessing shared data, threads can be made to work in a synchronized way such that if one thread is reading a data item the other threads will never modify it until the previous thread has done its reading completely and vice versa. In a thread based environment, a thread is the

smallest unit of code dispatched by the scheduler.

Data structures: A data structure represents some data of interest in a computation and provides a way of representing and storing of data in memory. The efficiency of a program, to a great extent, depends on the algorithm(s) used which, in turn, depend on how well the program manages its data using data structures. It is the responsibility of a programmer to write the code (process based or thread based) in such a way that access and updates to the data structures are done without conflict while obtaining as much efficiency as possible.

Objects: An object (in terms of Object Oriented programming [CN93]) is a unit of encapsulation of data as well as functionality. An object, in comparison to a data structure, is different because of the associated behaviors/methods stored with the data. Objects are self contained units and their encapsulation of data with the code to manipulate the data facilitates the isolation of the data from the outer world thereby improving code reliability and understandability.

5.1 Computation Model

At the highest level, there are two main approaches to solving a parallel programming problem, namely, the data driven or "data decomposition" approach and the compute driven or "functional decomposition" approach. A compute driven approach focuses on the computational aspects of an application, while a data driven approach focuses on the data being used by the application. This affects partial checkpoint/migrate as it determines the computational entities (processes, threads or both) that must be checkpointed and migrated.

5.1.1 Data Driven vs Compute Driven Approach

Functional decomposition, partitions a problem into smaller tasks without focusing on the data being accessed. These tasks are made as independent as possible (i.e., are kept disjoint). Once a successful task division is obtained, the data requirements of these tasks are examined. If a successful, largely disjoint, data access pattern is obtained, the partitioning is complete. Otherwise, if there is significant overlap in data accesses by sub-tasks then communication would be required to avoid data replication which might be inefficient so an alternate decomposition may be needed. Functional decomposition often results in pipelined communication.

Data decomposition examines the data (commonly the largest and/or most frequently accessed data first) and partitions it into smaller, approximately equal pieces to provide a basis for parallel execution and, hopefully, load balance. Based on this data partitioning, the computational functions are written which will perform some set of operations on each of the data partitions. This decomposition will typically not result in completely disjoint partitions, but will often be better than functional decomposition in terms of available parallelism and overall communication requirements.

A typical application that is running in a parallel or distributed environment runs in the form of chunks of code and data (i.e. small processes or threads) on different processors/cores accessing data that may or may not be shared. Accordingly, these processes may or may not communicate with each other. Apart from interprocess communication there may also exist some intra-process communication, where threads within a process communicate with each other (if processes are multithreaded). In

such hybrid applications the threads of a process may access the same data structures since they share memory. This effectively results in "communication" and the need for synchronization. Such in-memory sharing between processes on *different* machines is not possible, but if the code is written carefully, the sharing of data, and with it the requirement for communication, can be minimized by grouping threads into processes depending on what data they will be working on.

The computational model chosen affects the patterns and, often, the frequency of inter-process communication exhibited by an application. This, in turn, affects the design of any partial checkpoint and migrate system which must be sensitive to ensuring that migration does not negatively impact subsequent communication overhead.

5.2 Unit of Program Modularity

The unit of modularity of a program determines what parts of the code may access what data. This is important in designing any partial checkpoint/migrate system as it determines what program and data units are related and might have to be migrated together. Several possibilities exist depending on the execution unit chosen (processes or threads or both) and data model (data structures vs. objects).

5.2.1 Processes Accessing Data Structures

In a distributed computing environment where a process based model is used to perform a job, multiple processes will be running on several distributed compute nodes. Further, each process will have its own data structures to work on. If there are

to be shared data structures, they must be replicated and, their contents transferred between nodes to ensure consistency between the copies. In general, the programmer has to write the code in such a way that there is no potential for conflict in accessing and modifying data. If a "shared" data structure is used, then the communication incurred to maintain consistency of the data will be an overhead particularly for systems using conventional local area or wide area interconnects[1] for communication. Such sharing will significantly affect the design of any partial checkpoint/migrate system by increasing the frequency and/or volume of data communication.

5.2.2 Threads Accessing Data Structures

In a thread based environment a program is written in such a way that multiple tasks can be performed by a single program by assigning one task to each thread. In such a thread based environment a program can be divided into processes based on the data structures used by each thread. If threads using the same data structures are kept together with a single process then the overhead of communication is reduced because of the inexpensive inter thread communication. Again, this has an impact on the communication issues related to partial checkpoint/migrate.

5.2.3 Processes Accessing Objects

Processes might, instead, access data stored using objects. Processes can create objects of different types and then access data using the publicly available methods of those objects. Using objects to modify the data is possibly a better approach because

[1] As opposed to custom cluster interconnects such as Infiniband [Inc03].

then processes will work exclusively on their own, local objects. Objects are also easy to migrate due to their serializability property [BHP03].

5.2.4 Threads Accessing Objects

Finally, threads may also access objects just as processes can. With threads, however, multiple threads may sometimes be allowed to access the same objects and this must be considered in any partial checkpoint/migrate scheme. Threads accessing the same object must be grouped for checkpoint and migration with the corresponding object.

5.3 Model of Independence

The advantage of having independent execution units is that processes/threads which are on slower machines can be independently checkpointed and migrated to new machines. While migrating such execution units (e.g. processes), other processes which are executing on other machines are not affected. Due to their continued execution there will normally be some subset of processes in execution so progress will continually be made.

An "enforced" model of independence would require the application developer to make sure that the code is written in such a way that the units of modularity (i.e. computational entities) are independent (hence the term 'enforced'). This eliminates the need to identify groups of related processes/threads but is impractical and would be a significant burden on the application programmer.

A more practical approach might be to provide some framework or tools for writ-

ing the code in a way that is amenable to checkpointing and migration. For example, providing the application programmer with some libraries that encourage the programmer to write the code in the form of generally independent pieces (be that processes or threads accessing data structures or objects).

Yet another way of promoting independent pieces might be a compiler based approach. In such a scenario, the application programmer would write the application and the compiler would process the application code to determine the independent pieces and separate them, generating independent executables for distribution across processors. This is the most attractive approach but also is difficult to implement being, in essence, a form of automatic parallelization.

Finally, the user or the application programmer or compiler might also provide feedback on how to distribute the application. Such a system would use the previous behavior of the application (however made available) to determine an appropriate distribution of pieces on different machines. For example, GrAds uses COPs (Configurable Object Programs) [BCC+02] to map application pieces onto a collection of computing resources.

5.4 Discussion of Factors Affecting the Implementation and Efficiency of Partial Checkpoint and Migration

The enforced model of independence requires the application programmer to take all responsibility to ensure independence of code pieces. Thus, the programmer must

make sure that the processes that will be running on different machines are independent of each other. Instrumenting the communication library used for an existing application might help in providing information needed by the programmer to refactor code to create independent processes.

When data structures are used as the unit of *data* modularity, the programmer must write the code in such a way that computations that depend on specific data structures are co-located with each other. In a computational application, many data structures may be used to store the computational data during the execution of the program and large ones may be subdivided ("data decomposition") as needed. The data stored in the data structures is updated as the computational process proceeds. If the programmer makes sure that the data structures do not get separated from the code accessing them when distributing processes on different machines, then at the time of checkpointing and migration the processes working on different (distinct) data structures can be migrated independently. When processes working on one data structure are being checkpointed and migrated, other processes working on other data structures may continue working on their data structures ensuring that progress in the application as a whole continues to be made. Apart from this, the programmer might also provide a routine which the process can initiate, before checkpointing, on the new machine, so that a skeleton of the process is created (as done in [CCK+95]). Once the skeleton is ready the process can be checkpointed and migrated with all the data and state being copied to the new skeleton. The advantage of having such a skeleton initiation routine is that at the same time we are checkpointing the state, the destination location can download the skeleton initiation routine and start creating

the new task. This will save time when actual state migration happens since the skeleton process will already be waiting for the state data to be mapped into it rather than having to start this process during state migration.

When threads are the unit of modularity, the code must be designed using a multi-threaded approach that makes sure that multiple threads do not interfere with each other's operation. When using data structures (rather than objects), the programmer should write the application code in such a way that the threads needing to work on the same data structure should fall in one group running in a single shared address space. Since there will typically be several groups of threads working on their own data structures, we can checkpoint and migrate one group of related threads first while the other groups of threads are still executing. This will preserve performance because at least one group of threads will still be executing when some other group of threads is being migrated.

There may also be cases where a group of threads in one process are communicating with a group of threads in another process. In this case we might instrument the communication library to detect which groups of threads are dependent by keeping track of the communication patterns between threads at runtime so we can checkpoint and migrate all groups of communicating threads together.

The unit of data modularity might be objects rather than data structures. Code in this case is often naturally written to make these objects largely independent of each other. (Due to the data encapsulation property of OO design, all objects normally operate on independent data.) Java, for example, also provides object serialization which is an important capability for easily supporting checkpointing and migration.

Objects running on one machine can be serialized into network streams and later can be recovered from the network stream to work at a different location, without any specific requirements from the OS kernel and only minimal overhead on the part of the application programmer.

Processes and threads manipulating data structures require more effort when it comes to checkpointing and migration. The entire state of a process (registers, data-structures, etc.) needs to be saved and migrated to a new location. Restarting a process at a new location also requires significant effort, mapping the state and memory addresses, etc.. Both of these operations requires kernel-level support from the OS. In comparison, object serialization in Java is easier and does not require support from the system requirement perspective.

In a multi-threaded environment, where several threads are responsible for executing an application, checkpointing and migration can be achieved by serializing the threads [BHP03] and then migrating them to a new location and re-starting them. As mentioned previously, threads can be grouped based on the data that they are working on and migrated in groups as needed.

Ultimately, however, the convenience with which partial checkpoint and migration can be implemented will not likely be the deciding factor in determining what parallel programming environment will be used in practice. In reality, people will tend to use the most widely adopted software (which has been selected for a range of desirable features). For this reason, in my research, I consider how to partially checkpoint/migrate MPI processes and associated data structures (running in LAM runtime environment). Threads are therefore not a consideration in my partial check-

point/migrate system. The challenges in partially checkpointing and migrating MPI processes include stopping, checkpointing and moving only selected processes without affecting the other processes and determining the groups of processes that should be moved together (because processes communicating with poorly performing process will also be affected and thus perform poorly).

5.5 Possible Partial Checkpoint and Migrate Strategies for use with MPI

Several basic checkpointing and migration approaches might be envisioned to maintain the performance of a running MPI job, when part of it must checkpoint and migrate to a new location. These are worth considering to understand the important characteristics of a *practical* partial checkpoint/migrate approach.

5.5.1 Baseline Checkpoint/Migrate

An approach useful only as a baseline for making comparisons is to stop all the processes running on the various machines, checkpoint all process states, migrate all the processes to a new set of machines and then restart the processes on those machines. This is what existing systems do. The problem with this approach is that there may be a large delay involved before actually restarting the processes at their new locations and this results in lost potential performance if only a small part of an application is performing poorly, since the whole application must be checkpointed and migrated. This is wasteful.

5.5.2 Weakest Link First Approach

One possible approach would be based on focusing on the main cause for degradation in performance. We could identify the "weakest link", the machine that is the primary cause of degradation in performance, and address that problem first. Detection of such a weak link might be done by a monitoring mechanism that keeps track of the load on each compute node. When such a machine is identified, we could decide to replace the weakest link with another machine, moving the necessary processes from the old machine to the new one, and leave the rest of the running processes in place.

It will not always be possible to easily identify a single process. The decision of exactly which processes must move must be based on the communication patterns of the running processes. If there has been a slowdown on one machine and, based on the communication patterns, it turns out that the single process running on that machine was not a communication intensive process, it may be possible to simply bring a new nearby[2] machine into the existing cluster so that the weakest link machine can be replaced with the new one. On the other hand, however, if the single process is communication intensive then other processes with which it is frequently communicating will also be affected by the slowdown which may lead to a situation where many or even all the processes *appear* to be slow. In such cases, detection of a single weakest link could be difficult. Actual load conditions on nodes not just MPI application process performance should therefore be analyzed to find the weakest link(s). Further, in a practical system, communication patterns would have to be

[2]Again, "nearby" is meant in terms of network proximity.

used to define groups of tightly communicating processes to move together with any "weakest link" process(es). Analysis of communication patterns could be done statically (before runtime) at low cost but with possibly limited accuracy or dynamically (during runtime) at higher cost but more accurately.

A straightforward approach for finding the weak link in a low communication environment is to find the process which is performing most below its expected performance level (as, perhaps, determined by analyzing previous runs of the same application or by comparison to a pre-established performance contract by a "contract monitor").

Issues related to locality also play a role. When the decision is made to migrate a process to another location, the location must be chosen considering the locality of the rest of the MPI job. Choosing a distant location to migrate to is impractical as the cost of migration will be high and subsequent performance will suffer when any inter-process communication is required. Thus, the selection of new execution sites must be restricted to those near to the location of non-migrated processes to preserve inter-process communication performance after migration.

The goal of the weakest link first approach is to migrate the most poorly performing process(es) first so they are restarted earlier on better hardware and can potentially "catch up" while processes performing well continue to make progress. Difficulties due to frequent inter-process communications might be dealt with by migrating the processes on a group basis, as discussed in greater detail later.

5.5.3 Considering Important, Upcoming Computations

Another possible approach might be identifying the "important" upcoming computations in each process and checkpointing and migrating them first so that they are ready to run efficiently at the new site(s) when needed. An issue here is how to determine the "important" upcoming computations. Pre-analysis of the code either manually or by compiler could potentially be used to determine the "important" sections of the code to be those which are most visited (for example a routine or a loop body). Communication patterns may also help in determining important computations since, for example, some other processes may be waiting for some process to provide some input making the waited-on process more "important".

In general we would want to determine the piece(s) of code whose execution limits overall performance, possibly by using an idea like critical paths [Hol96]. Potential parallelism and performance effects can be determined if we have a critical path graph of the overall application's execution. Given a program activity graph (a graph of all the events in a single program with nodes representing the events, arcs connecting them representing communication and data sharing dependencies between processes and values on arcs representing the total time taken to communicate from one event to the other) then the longest CPU time weighted path is the "critical path" of that program [Hol96]. Parallel parts of the application are represented by independent paths in the activity graphs. If we can determine which part of the application or program is performing poorly, we could choose to migrate upcoming parts along the same path in the activity graph first so that they get started earlier at a better machine. The rest of the parts could then be transferred lazily there after.

The benefit of such an approach would be faster restart at the migration point and lazily sent checkpoint data might present less immediate demand on the network (as bandwidth use might be spread out over a longer period of time). Unfortunately, the complexity of determining the "important" computations is challenging. Further, it would be hard to guarantee that what we have determined to be important will actually be important and will never need ongoing communication with other computations that are being lazily transported. Hence, accurately identifying the correct, important upcoming computations to move first is critical. Furthermore, it is important to note that the pieces that can be easily considered for migration are those pieces of code, at the language level, which are easily separable such as objects or processes, not finer granularity parts of the code such as a loop which would have to be pulled out of a process or object somehow for migration. This would further complicate the implementation of such an approach likely making it impractical.

5.5.4 Move Processes Per Machine

Another simple strategy might be to migrate processes on a per machine basis. If there is low dependency between processes on different machines (for example, due to use of a hybrid shared-memory/message-passing design for multicore machines which would naturally focus communications/data sharing within each machine) we might decide to migrate processes on a per machine basis. In this case, multiple processes running on a single machine could be easily moved together since they are not communicating extensively with other processes running on different machines.

In parallel programming it is not always necessary that all processes are always

communicating. For example, in some applications only subsets of processes communicate regularly. Also, there are cases in some parallel applications where processes go through an initialization stage and only after having done some computation do they need to communicate with each other. In such cases, when the processes are in the initialization stage or another stage in which they are not communicating frequently with remote processes, we could easily migrate the processes on a single machine. This approach is simple. The processes running on the machines can be monitored and, upon degradation of performance, machines can be selected for checkpointing and migration based on such factors as there being fewer processes on the machines or machines that have "crucial" processes might be migrated first. Determining this crucial nature of a process is also an issue, which might be dealt with by using some information provided by the programmer. Overall, however, this approach suits only a limited subset of all parallel applications.

5.5.5 Move Tightly Communicating Processes Together

It seems clear that we would want to migrate groups of tightly communicating processes together across a group of machines. Further, it should be possible to determine what groups of processes need to be migrated together based on high communication rates between the processes. These communication patterns could be determined by using some communication library instrumentation (such as in MPICL [GHPW90]). Using this approach, all related processes would be migrated together thereby decreasing long communications over the network (which would happen if a process were left behind that must communicate with one or more processes which were mi-

grated to a remote site). This approach will likely be of only moderate difficulty but will be practical only if the determination of machine groups is accurate and not a significant runtime overhead.

5.5.6 Partial Checkpointing and Migration of Parts of Processes

The hardest, but most flexible, approach would be moving *parts* of individual processes without stopping any of the processes. This approach is potentially very efficient because, when only parts of processes are being checkpointed and migrated, none of the processes are actually entirely stopped. In such a scheme, applications might be written using an object-oriented approach where an application would consist of several processes and each process would consist of several, hopefully largely independent objects. Since objects have their own local data on which they work, it is conceptually easy to migrate them one by one to other machines without having any interference on other objects left behind and continuing to execute. This approach would minimize loss of performance because when some objects are being migrated other objects will still be executing. Unfortunately, such a scheme does not fit within the MPI programming model.

Chapter 6

Partial Checkpoint/Migrate of MPI Processes

Considering the factor influencing partial checkpoint and migration discussed in the preceding chapter and focussing on an MPI environment, I have chosen to consider a partial checkpoint/migrate system that tries to migrate tightly communicating processes together. This chapter describes how such system can be built using LAM/MPI.

6.1 Assumed Architecture

My assumed architecture is shown in Figure 6.1. I assume that there exists a Distributed Services environment which tracks the compute resources (i.e. nodes) that are available. This is done by a Resource Discovery component, and allocation of the resources to users is handled by an Allocation Manager (all as shown in Figure 6.1).

The end users must interact with these components to request the execution of their jobs. Thus, the assumed Distributed Services environment supports job submission and must also support the monitoring of job performance and host load to ensure acceptable performance. These features are necessary in a real system but their design and implementation are outside the scope of this thesis.

To provide the user with an environment where she/he is free from the concerns of changes in performance of the jobs submitted due to overloading on execution nodes we need to be able to detect performance degradation (due to overloading) and take necessary steps to maintain desired levels of performance. The *Application Manager* in Figure 6.1 is assumed to monitor application execution to assure a reasonable level of performance.

The Application Manager could either use statistics from previous runs of the job (assuming the job is being re-run), or it could build an estimate of performance on the fly. Given such run statistics it should be straight forward for the Application Manager to assess performance. User requirements could also be considered when assuring such levels of performance though this is, again, beyond the scope of this thesis.

The monitor and the lightweight daemons, also shown in Figure 6.1, are assumed to exist at the remote execution sites as shown in Figure 6.1. The daemons should periodically send execution statistics/average loads on all the nodes involved, to their monitor. If a monitor detects that there is a performance problem with any of the execution sites then it will notify the Application Manager so that any necessary new execution locations can be chosen.

It is assumed that the nodes that are selected for job execution are initially free of any overload. Each monitor gathers the load averages of all its nodes and shares this with the Application Manager which can then determine an estimated level of load for each node by analyzing the gathered load averages and the expected level of performance for application execution. This allows an estimate of performance to be made on the fly when necessary.

The monitor will not make any decision whether or not to migrate MPI processes to other locations, it will just inform the Application Manager about any performance deviation. It is the Application Manager, which is assumed to have broader knowledge of the application as a whole, that should decide whether or not slow running MPI processes should be migrated to other location(s). This concept of having an Application Manager and assuring necessary levels of performance is common to existing distributed execution systems [BCC+02; AAB+02; CFK+02; AAF+01; VD03].

There are two situations when a Monitor would need to notify the Application Manager about performance deviation. First, when there is no response from one of the Daemons. In this case, the Monitor should assume that there has been a failure and notify the Application Manager. Second, and of interest to this thesis, when the Daemons send the load average information to the Monitor, it should perform a "lightweight" computation (checking to see if the load average on each node is in approximate agreement with the expectation set by the Application Manager) to see if there is a performance deviation and, if so, it should inform the Application Manager.

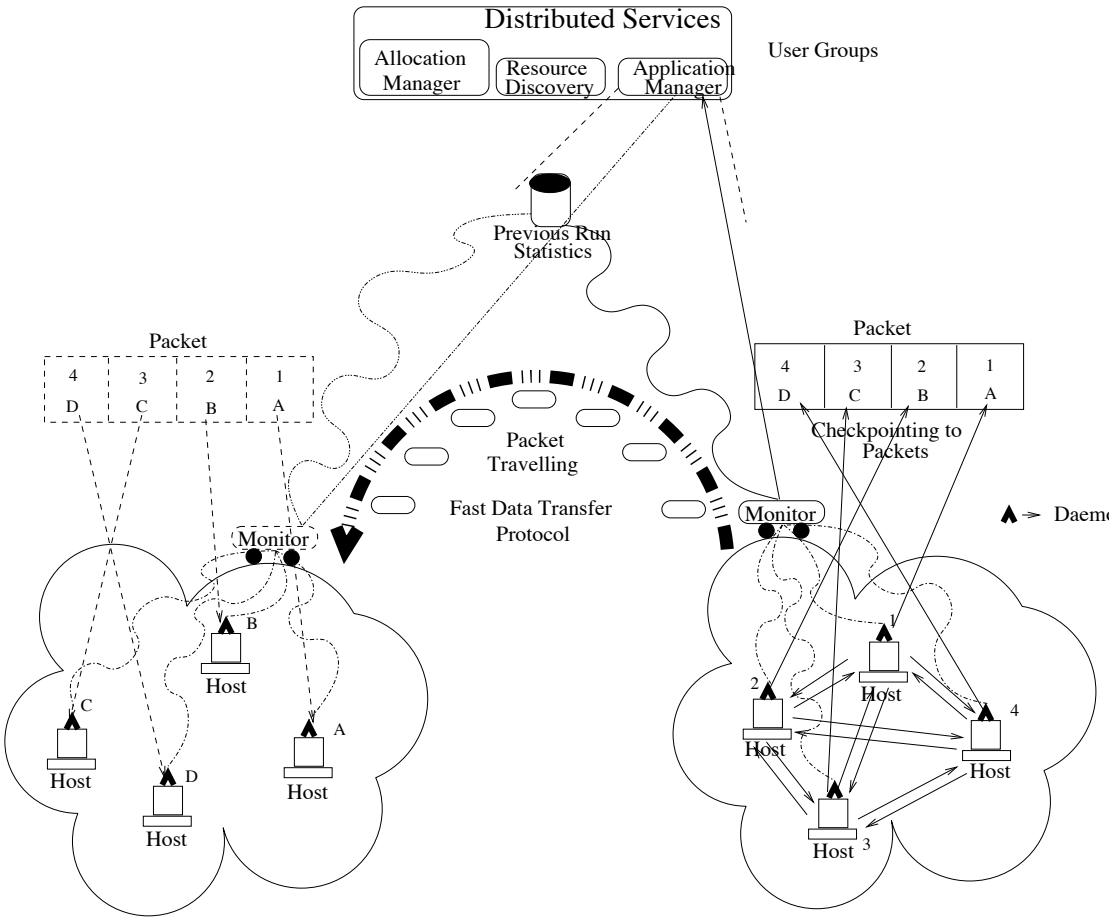

Figure 6.1: Assumed Partial Checkpoint/Migrate Architecture

The Application Manager's decision to partially checkpoint/migrate could be based on several considerations, such as the cost of migrating the MPI processes to some other location, how much the performance will be affected due to migration and whether migrating specific processes to a new location will still be able to maintain the necessary level of performance that was originally estimated. The whole idea is to try to assure a reasonable level of performance for the job in total, whether the job is executed without degradation of performance or the MPI processes have to be migrated several times due to performance degradation.

When a decision is made that one or more new nodes are to be used, the Application Manager will ask the Resource Discovery component to select suitable nodes, considering the locality of the nodes to the current application execution location, and the appropriate process(es) will then be migrated. If the Application Manager decides that the performance of the entire "cluster" executing the job is deteriorating it could also choose to migrate the whole job to some other suitable location(s). A suitable alternate location (in this case an entire cluster of machines) would then be selected by consulting the Resource Discovery component and the application would be migrated. The case of interest in this thesis, however, is when only a subset of the MPI processes must migrate.

Two possible mechanisms for the actual transfer of checkpoints to the new location could be used: fast transfer and lazy transfer. The purpose behind using two mechanisms would be to achieve effective migration of jobs at reasonable cost (in terms of network overhead). Consider an example of how such a transfer might work. On a given machine running four processes the system might determine that one of the processes which is heavily communicating with processes running on *different* machines is performing poorly due to unexpected load on the machine. The other three *local* processes communicate only sporadically with the process which is performing poorly. It makes sense to move the poorly performing task first and eventually (lazily) the remaining three processes later. For the poorly performing process the checkpoint data should be sent immediately via a fast transfer mechanism, because the poorly performing process needs to be up and running again at the new location as soon as possible so it does not adversely affect the other processes (running on different ma-

chines) which may be waiting to communicate with it. The checkpoint and migrate of the other three processes could then be done at an opportune time using the lazy approach. Again, while this seems to be a useful technique, its implementation is beyond the scope of this thesis. It could be incorporated as future work.

6.2 Assumed Programming Environment

My system is designed to run in an environment that will consist of machines running Linux that are LAM/MPI capable. The Berkeley Lab Checkpoint Restart (*blcr*) software [DHR02] is also a core requirement as *blcr* is the only checkpoint and restart module that is currently supported by the LAM/MPI run-time environment. Further, my assumed computing environment considers only homogeneous machines since *blcr* is a process-level checkpointer and because it would be undesirable to require application-level checkpointing which is necessary to support the use of heterogeneous machines. Using application level checkpointing, the application developer is responsible for writing the specific code for taking checkpoints. This can be complex and is unlikely to be readily adopted by computational scientists who would rather rely on the execution environment to provide a checkpoint and restart facility transparently.

The Berkeley Lab Checkpoint and Restart facility checkpoints running MPI jobs to disk for later restart on the same type of node at a different location. I assume *blcr* will be available as a selectable LAM/MPI module on all the machines on which an HPC job is to execute. To decrease the delay in checkpoint and migrate, checkpointing could easily be redirected to network endpoints rather than files though this was not

done in my prototype implementation.

The computing environment chosen simplifies the problem of implementing partial checkpoint and migration in two ways. First, using *blcr* eliminates the need to develop my own checkpointer. Second, due to the component framework of LAM/MPI it becomes easier to add functionality to the existing LAM/MPI RTE by developing the desired module (plug-in, System Software Interface (SSI) module).

6.3 Implementing Partial Checkpoint and Restart

An understanding of the overall procedure that takes place when an MPI application is started, checkpointed and then restarted from a saved checkpoint state (as discussed in Sections 3.2.1 through 3.2.3) reveals the points where the main implementation efforts had to be made. As shown in Figure 6.2, in a normal ("complete") checkpointing case (i.e. checkpointing of all application processes, as it currently exists in LAM/MPI) when a checkpoint request is received by the *mpirun* process, it *forks* a child process to take care of preparing all the application processes to take the checkpoint and then waits for the child process to indicate that the application processes have checkpointed their states successfully. The child signals the application processes (e.g. A, B and C, shown in Figure 6.2) to prepare for checkpoint and the processes temporarily suspend their execution to take the checkpoint and then return a success indication to the child process. The child then returns a success indication to its parent process (*mpirun*) which then creates an application schema which contains information about all the processes' (A, B and C) checkpoint images and saves it as part of the checkpoint file associated with the *mpirun* process.

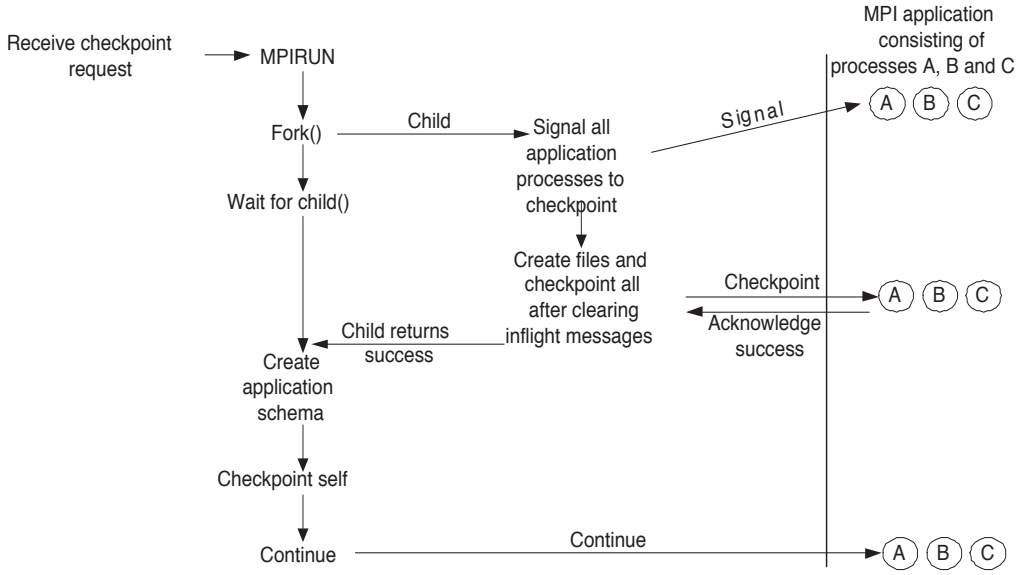

Figure 6.2: Normal ("complete") Checkpointing in LAM/MPI

The changes in processing needed to support partial checkpointing are sketched in Figure 6.3. When a partial checkpoint request (for the selected processes) is received by *mpirun*, the processes that do not need to be checkpointed are informed that a partial checkpoint request has arrived and that they are not required to checkpoint themselves but must help in checkpointing the other processes (e.g. by clearing the outstanding/in-flight messages for the checkpointing processes). In Figure 6.3 processes A and B are informed that process C is going to get checkpointed so processes A and B must temporarily block communication with process C, and help it in clearing any in-flight messages so that it can take a successful checkpoint. In this way A and B will not be stopped after the checkpoint of process C is taken (and process C is removed from the group of running application processes). Thus, they will be able to continue with their usual execution up to the point of any attempted communication with C when they will be blocked until C restarts.

Currently, if any application process dies, *lamd* informs *mpirun* about the failure and *mpirun* aborts all the processes. In partial checkpointing, the selected processes must be able to checkpoint and then terminate, without causing the termination of other processes. In my implementation, after informing the processes that are not to be checkpointed, a child process is again forked by *mpirun* to prepare process C (in the example) to be checkpointed. Process C then prepares itself for checkpointing

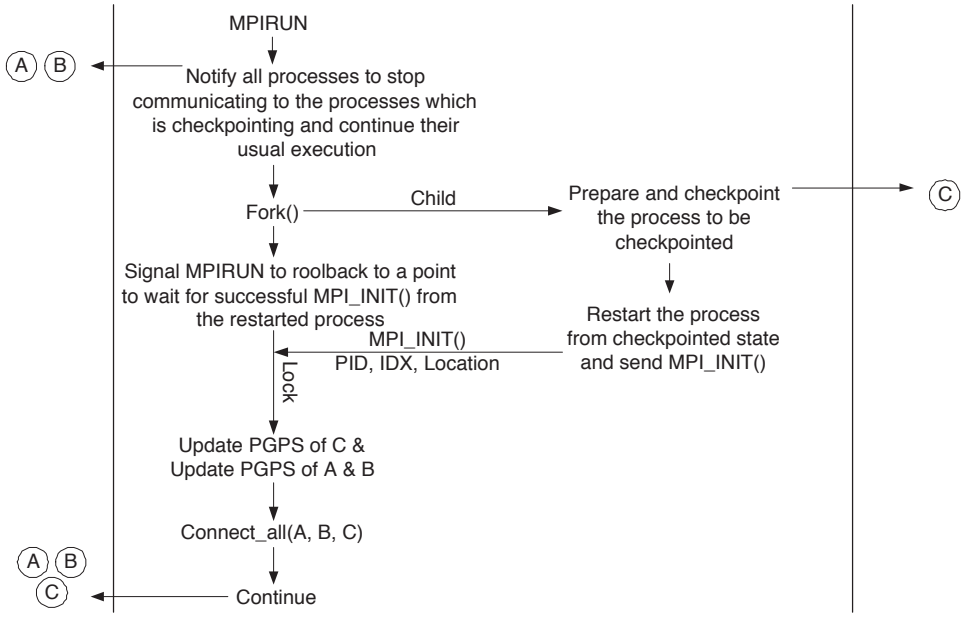

Figure 6.3: Partial Checkpointing in LAM/MPI

and processes A and B help it to clear any in-flight messages. Process C then writes its checkpoint image to a file and terminates while processes A and B continue with their normal execution.

There are two distinct challenges to implementing partial checkpointing in LAM/MPI: relaxing the requirement that all processes be active to allow continued application execution, and handling changes in communication between MPI application processes

during both checkpointing (which requires temporary blocking of processes trying to communicate with checkpointing processes) and migration/restart (which requires establishing of new communication channels between migrated and non-migrated processes). Accordingly, and for simplicity, my prototype implementation and the rest of this section are divided into two parts, the first of which ignores issues associated with communication. In the absence of inter-process communication, informing the processes about the partial checkpoint is not necessary. This is only necessary when the processes are communicating. In such a scenario, in the example from Figure 6.3, after process C is successfully checkpointed and terminated, *mpirun* is signaled for the specific restart of process C on a different node. The *mpirun* process will then re-adopt process C and again make it a part of the normal execution environment. More specifically, after receiving the signal to restart, *mpirun* will restart process C from its checkpointed state and wait to receive a message based on successful initialization that includes the new process' PID and IDX (see Section 3.2.1). The *mpirun* process then updates the newly joined process' PGPS (Section 3.2.1) by sending it the latest PGPS and at the same time it updates the PGPS structures of all the other processes to let them know that process C has returned and again joined the execution environment (at a new location). Having received the updated PGPS, the processes that were already running must wait for the newly joined process to establish communication channels with them (even though no communication is assumed for the moment) and then normal processing resumes from this point forward including the newly restarted process, C.

6.3.1 Partial Checkpoint and Restart with Non-Communicating Processes

Consider a trivial non-communicating MPI application consisting of two independent processes. The MPI execution environment contains two processes with rank 0 and rank 1. The goal is to be able to checkpoint one process, say rank 1, terminate it, and then restart it and make it again a part of the MPI application processes without affecting the execution of the other process, having rank 0.

In normal checkpointing when a checkpoint command is issued, all the processes of the MPI application are checkpointed. To take a partial checkpoint we obviously need to omit taking checkpoints for the processes that are not involved. We want calling the *cr_checkpoint()*, the command issued to take a checkpoint of the MPI application (refer to Figure 3.4) to be restricted to the processes that are to be checkpointed. The *blcr* plugin facilitates checkpointing by allowing the processes to register a thread based callback function for taking the checkpoint. Once a thread based callback is registered with *blcr* and a checkpoint command is issued, no matter what, *blcr* currently makes sure that a checkpoint of *all* the process is taken.

Two approaches are possible to implement partial checkpointing. First, signal only those processes that need to be checkpointed and second, signal all processes for checkpoint but checkpoint only those that need to be checkpointed. I chose the latter approach, because I wanted to make maximum re-use of the functionality that is already provided in the LAM/MPI code base. Further, in the first case, if only those processes that need to be checkpointed are signaled then the processes that are not being checkpointed will not know to help the processes that are getting checkpointed

in clearing their in-flight messages (an extra overhead would then result at restart time to synchronize such messages). On the other hand, if all the processes are signaled for the checkpoint then the processes that do not need to be checkpointed can easily help the processes that need to be checkpointed in clearing their in-flight messages. After the in-flight messages are cleared, the processes that do not need to be checkpointed can then be simply omitted from checkpointing.

The call to *cr_checkpoint()* in the thread callback function can take three different possible argument values. Zero, which is the normal case, indicates ready for checkpoint (or CR_CHECKPOINT_READY) and means that every thing is fine and a checkpoint can be taken. CR_CHECKPOINT_TEMP_FAILURE normally signals to abort checkpointing and continue. Since this is not a functionality required in my prototype implementation, this is the one that I chose to pass to *cr_checkpoint()* to cause it to omit checkpointing for the processes that do not need to be checkpointed. CR_CHECKPOINT_PERM_FAILURE signals to abort and terminate all processes.

Once a partial checkpoint is done, the files produced naturally correspond to the processes that were checkpointed. The checkpointed files contain all the process images and each can be independently restarted by using BLCR's *cr_restart* command.

Removing the Checkpointed Processes from the Execution Environment

To be able to checkpoint only selected processes and then terminate them without affecting the other running processes, a slight change in the *lamds* code base had to be made. After terminating the desired processes (processes that were checkpointed) *lamd* discovers this "failure". I have changed the code so *lamd* ignores the "failure"

and therefore does not inform *mpirun*. This way, *mpirun* does not terminate the other processes but, instead, continues execution as usual.

Signaling MPIRUN to Restart a Checkpointed Process

After migration, when processes must be restarted, *mpirun* is signaled using a user defined signal to indicate that the processes should be restarted. Upon receiving this signal, *mpirun* constructs the commands that need to be invoked on the new execution nodes to restart the checkpointed and migrated processes and then issues the commands following the same procedure as if a new process was being started. The *cr_restart* command invoked on the new nodes to restart the processes from their checkpointed files, starts the processes which then send their PID, IDX and indication of successful initialization to *mpirun*. After the PID and IDX are received by *mpirun*, *mpirun* propagates the updated PGPS struct (which holds the latest information about all the processes) to all MPI application processes as described earlier. All the processes receive the PGPS and perform the same steps to connect with each other as they did when they were initially started. After the new connections are established, the processes that were running and the newly restarted processes continue their execution.

6.3.2 Partial Checkpoint and Restart with Communicating Processes

In a non-communicating processes execution environment, the processes running do not utilize the socket connections that are established when the virtual machine

is setup. Naturally, in a communication oriented environment processes do communicate with each other using the socket connections. Handling these communication connection is the other challenge in supporting partial checkpoint and restart.

The most fundamental communication operations in an MPI application are blocking write and blocking read. In a blocking write if the buffers to which an application process writes gets filled, the application will be blocked. Similarly, when there is a blocking read operation, it must wait for the application to write some data in the buffers which are read from. If there is no data available the blocking read will not return control to the application but will wait until data becomes available. Numerous other forms of communication are also provided including non-blocking variants of send and receive and collective communications operations. I will discuss only the two basic operations upon which the others can be built.

In the case of communicating processes we first need to figure out which processes are communicating so we can treat them specially during partial checkpointing. Currently, if a process were to be checkpointed and stopped that was communicating with another process, and that process attempted to send a message to the stopped process, it would throw an error due to a broken socket connection because the process that is supposed to read from the buffer is seen as having terminated. The mpirun process will then pick up this error and terminate all the processes. Also, if process was to read a message from the checkpointed process, then it would again throw an error due to a broken socket connection as a result of the blocking read and mpirun would terminate the application.

To deal with the problem of avoiding communication between running and stopped

MPI processes, we need a mechanism to find out who is communicating with whom to checkpoint groups of communicating processes together. To establish a proof of concept that we can successfully partially checkpoint with communicating processes, I use the simple approach of checkpointing the entire group of processes communicating with any process to be checkpointed. This is done without affecting the processes that do not communicate with the process to be checkpointed. I also assume that *mpirun* knows the members of these communicating groups of processes.

Consider an example MPI application with 4 processes where process 0 sends messages to process 1 and process 2 sends messages to process 3. If we were to checkpoint process 1 then process 0 should also get checkpointed with process 1, but processes 2 and 3 should still continue their normal execution even after processes 0 and 1 are checkpointed and removed from the *mpirun* execution environment. After restart *mpirun* should incorporate processes 1 and 0 back into the execution environment and let them continue from the point where they left off.

Checkpointing Communicating Processes

Checkpointing of communicating processes, as described in Section 6.3.1 is not the same as checkpointing non-communicating processes. In checkponting communicating processes the same general approach of checkpointing only the needed processes and omitting the checkpoint of other processes is, naturally, followed.

The way LAM/MPI takes a process checkpoint using *blcr* is it invokes a callback function that it registered with *blcr* which invokes CR (checkpoint/restart) handlers and all the processes yield execution to their CR handlers. The CR handler first

acquires a lock on the MPI library to prevent further MPI calls. This is the point where all the processes yield to their CR handler (i.e. they stop use of any MPI library routines) and then wait for the CR handler to finish the checkpointing. As part of the process of checkpointing, the processes exchange messages using *out-of-band* communication channels to clear all inflight messages to achieve a globally consistent checkpoint state. Once the processes have finished clearing these messages, the CR handlers create their respective checkpoint files for each of the processes and the checkpoint images are written to these files by all the processes independently by calling *blcr*. After a successful checkpoint, control is returned to the CR handler. The CR handler then releases the lock that it acquired on the MPI library and all the processes can continue with their usual execution as if nothing has happened.

There are two possible ways to implement partial checkpointing of communicating processes. First, we could choose to checkpoint and remove only those processes that are performing poorly and temporarily stop the other processes that belong to the same group of communicating processes. The processes that are temporarily stopped, would then wait for the checkpointed processes to restart and then they can continue their processing. The other approach is to checkpoint and remove all the communicating processes that are in the same group as those processes that need to be checkpointed due to poor performance. In both approaches, the other processes (those not belonging to the group of poorly performing processes) can continue with their usual processing. I chose the first approach to establish the proof of concept but either approach could be used.

In the case of communicating processes we need to omit the checkpointing of

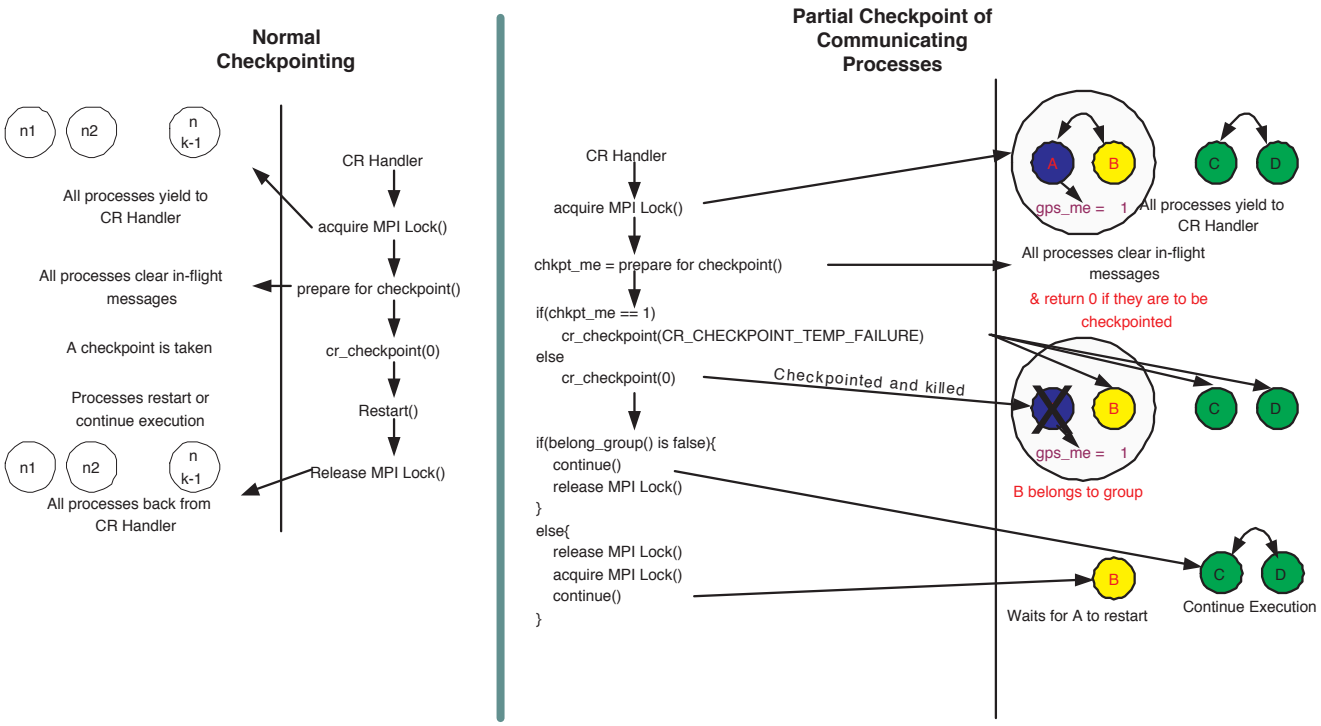

Figure 6.4: Partial Checkpoint and Restart

processes that should not be checkpointed. In my implementation I maintain a flag "gps_me" in the PGPS struct for each process to specify that the process needs to be checkpointed. A process is thus checkpointed if the gps_me field is set to 1 and if it is set to zero it is not checkpointed. Except for the setting of flag, the procedure of partial checkpointing is the same as checkpointing for non-communicating processes. The difference comes after checkpointing the processes. For all the remaining processes that were not checkpointed, the CR handler releases the lock and allows them to continue. But for the processes that are in the group of processes that were checkpointed, the CR handler re-acquires the lock preventing them from making any MPI library routine calls.

A high level comparison of the complete and partial checkpointing processes for communicating processes is shown in Figure 6.4. On the left side of the figure, the processing done during complete checkpointing is shown, on the right hand side, the processing done during partial checkpointing is shown. In the figure, four MPI processes are running that communicate pair-wise and process A needs to be checkpointed. Group formation is assumed to have paired process A with B and process C with D, reflecting the pair-wise communication pattern. After A is checkpointed, B, which is in the same communicating group, waits for A to be migrated and restarted but processes C and D continue execution. As can be seen in Figure 6.4, the gps_me flag for process A is set to 1 which signifies that process A needs to be checkpointed. The group formation based on who communicates with whom is represented by double ended arrows (A with B and C with D). When all processes prepare for the checkpoint of A they yield to the CR handler and a checkpoint is taken for process A by passing the value 0 to *cr_checkpoint()* while for all the other processes CR_CHECKPOINT_TEMP_FAILURE is passed. After process A is checkpointed the lock is released for all the processes which allows processes C and D to continue with their execution. The lock is re-acquired for process B so it waits for process A to be restarted.

If the communicating group formation is incorrect and, say, process C sends a message to process A, process C is allowed to wait for process A to come back. This is determined by consulting the gps_me field if the message is to be sent to or received from a checkpointed process. The gps_me field plays a major role from the point a process is checkpointed to the point it is restarted. Once the checkpointed process is

restarted the gps_me field is reset to 0 and processes continue with their execution.

Restarting Communicating Processes

During partial restart, *mpirun* constructs the command to be invoked on remote processes to restart the processes that were checkpointed. The other processes just receive a signal from *mpirun* to run the restart routine to help the checkpointed processes restart. This is necessary so that all the processes will go through the steps of sending their PID's, etc. to *mpirun* so *mpirun* can create and send the complete new PGPS to all processes which will connect to each other by calling the appropriate routine, connect_all(), to establish the necessary new socket connections and then resume normal execution.

6.3.3 The Implemented Prototype

The implemented prototype currently runs on Linux Fedora Core 4 and BLCR version 0.5.3. I used the development version of LAM/MPI, version 7.1.4b3svn/MPI 2., for implementing the partial checkpoint/restart functionality. After building and deploying the necessary custom Linux kernel, I successfully implemented partial checkpoint and restart for both non-communicating and communicating MPI applications. I have tested several applications, successfully checkpointing and restarting parts of MPI applications without affecting the execution of other, non-checkpointed, processes. My prototype implementation is described in [SG08].

Chapter 7

Grouping Communicating Processes to Checkpoint and Co-migrate

As discussed earlier, it is ineffective to checkpoint and migrate a *subset* of processes that communicate frequently with one another. The processes not checkpointed will quickly block waiting for the checkpointed processes to be restarted. Further, if the checkpointed processes are migrated relatively far away from those "left behind" poor communication performance will be experienced after restart. For these reasons, it is important to be able to group together those processes which are tightly coupled for communication so they can be migrated together as a group. By doing this, communication efficiency will be maintained. For long running applications with identifiable subsets of processes that communicate frequently, partial checkpoint and migrate should then be effective. As there are different kinds of applications with

different communication behavior, it would be a valuable contribution to be able to group processes for some widely used applications with different communication behaviors. Clearly, one or more algorithms to group processes by analyzing the collected communication data is needed. How to effectively do this grouping is a core contribution of this thesis. A number of possible matching algorithms from different areas of computer science were considered (as described later) and I chose to adapt the TEIRESIAS patterns matching algorithm from bioinformatics to the problem of identifying commonly communicating groups of MPI application processes.

7.1 Communication Characteristics of Applications

In high performance computing there are many different kinds of applications. The communications behavior of these applications depend on the design of the applications. Some factors that affect communication behavior are parallel programming style (e.g. use of point-to-point vs. collective communication), choice of algorithm (e.g. different types of data decompositions), and the characteristics of the data itself which, for some problems, determines the needed inter-process communication. All the different categories of applications have different communication requirements leading to different communication patterns. To be able to checkpoint and migrate parts of an MPI application we need to be able to effectively find groups of processes that commonly communicate with each other. Such a group finding mechanism can be used to predict a future group of communicating processes by analyzing the earlier communication behavior of the processes of an application.

To illustrate some communication characteristics of typical application types and

to better understand some of the issues associated with a mechanism for finding groups of communicating processes, I now briefly review some popular applications types.

One commonly used programming style for several parallel MPI applications is the Master and Slave (master/slave) approach. In this approach, a single master process coordinates several slave processes that collectively solve the problem in parallel. Communication between the master and the slaves (and sometimes among the slaves) can occur and recur over a period of time. A typical master/slave approach involves three phases: distribution, computation and result collection. The master process distributes data to the slave process in equal (or close to equal) chunks. The slave processes then perform computation on the data they have and, if required, communicate with other slave processes to acquire additional partial results they may need. After having performed the computation, the slaves send their computed results to the master process. This procedure of distribution, computation, and collection can be recurring or may involve a single distribution phase followed by a potentially long computation phase and finally the collection phase. In both scenarios, the communication pattern is regular.

An example of data induced communication patterns in an MPI Application would be the use of a Finite Element Method (FEM) for a heat transfer problem. In a classic instance of this problem, the two ends of a hot, insulated metal rod are kept in buckets of ice water at 0 degrees celsius. The temperature from both the ends cools down the rod while the heat from the rod dissipates into the buckets. It needs to be determined how much time it takes to cool the rod (i.e. for the temperature of the buckets and

rod to reach a steady state). For calculating the temperature of the rod overtime we conceptually divide the rod into N fragments and assign N/K adjacent fragments to K processes. The K processes collectively compute the changing temperatures of all the fragments as the rod cools. The temperature of each fragment at time t depends on the temperature of its neighboring fragment at time $t-1$. The end fragments of the rod have neighboring temperatures fixed at 0 °C due to the ice water baths surrounding them. Hence for the computation of the temperature of a fragment each process needs to know the temperature of its neighboring fragments. The temperature of a fragment that lies on a boundary between processes can be computed only using the temperature from its neighboring fragment which, in this case, is computed by another process. This induces inter-process communication that is determined by the structure of the problem data (hence "data induced communication"). In such data induced communication, the patterns of communication are commonly regular for the entire period of time the application is executing but the frequency of communication may change over time.

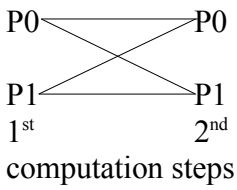

Figure 7.1: A Butterfly in FFT

Some communication patterns are algorithm induced. An example of this would be the solution of a Fast Fourier Transformation (FFT) using the butterfly approach. In FFT a butterfly communication pattern is formed when the values from previous

computation steps are exchanged between processes to compute the next value, as shown in Figure 7.1 where processes P0 and P1 exchange values from their computation step 1 to compute their values in computation step 2. The communication shown by the lines in Figure 7.1 form a "butterfly" pattern of communication (one which is reminiscent of the wings of a butterfly). In an FFT computation where we have 8 points allotted to 8 processors, then applying the butterfly approach will require different processes to communicate with their *changing* butterfly counterparts as they proceed in computation, as shown in Figure 7.2. For example, process P0 will need to communicate with process P4 in the first phase of the computation, to communicate with process P2 in the second phase of the computation, and to communicate with process P1 in the final phase. Depending on the algorithm used to solve an FFT, the

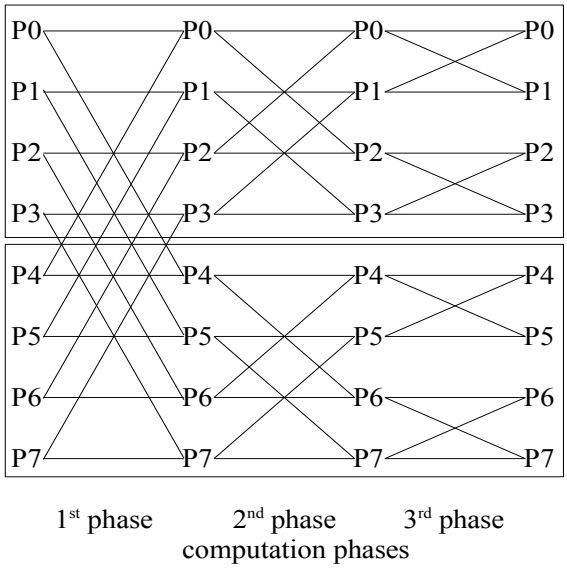

Figure 7.2: 8 Process FFT Butterfly

pattern of inter-process communication may change. Hence, the pattern of communication is algorithm based. While the communication pattern is not consistent, it is

regular.

The communication patterns for all the examples discussed have some degree of regularity. This is important because regularity provides a basis for prediction. If communication is regular and repeating then past patterns can be used to predict future behavior. Not all parallel programs have regular and repeating communications but many do and it is these programs that I am interested in, in this thesis. I considered regularity of communication patterns in the creation of my predictors which is described next.

7.2 Possible Types of Predictors

Figure 7.3 shows one possible classification of different types of predictors. There are potential advantages and disadvantages to choosing each type of predictor. A general application independent predictor might be a cheap predictor, but intuition suggests that it could give the worst results since using just one predictor for all types of applications might not be able to recognize the different types of communication behaviors.

The most accurate type of predictor would likely be some type of custom predictor which is application dependent. Unfortunately, to use this type of predictor significant effort will be required from the application programmer. The application programmer will, at least, have to specify the application type to pick the best available predictor and may have to write the predictor itself. Though a custom predictor would likely give the best results, it is highly undesirable for the application programmer to have to write or even select the predictors themselves.

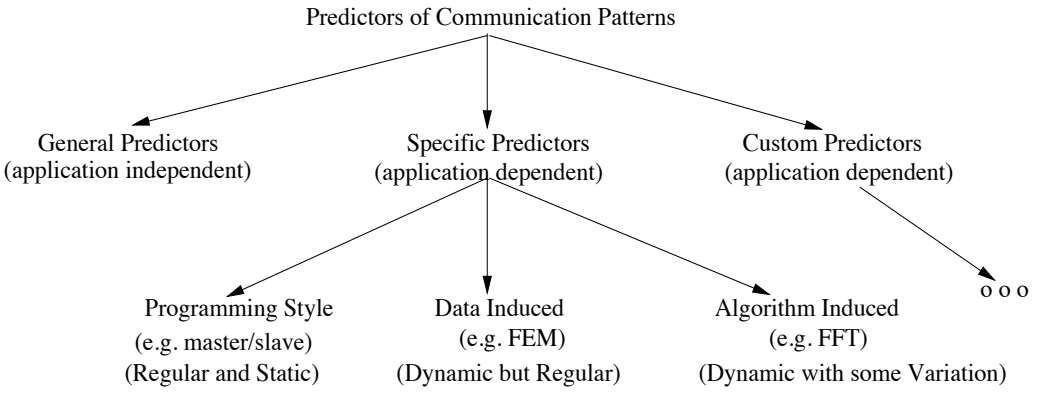

Figure 7.3: A Classification of Possible Types of Predictors

There is a middle ground which might involve the use of a small number of specific but widely applicable predictors (which are also application dependent). Under this category, we might broadly categorize applications into a few major categories. In Figure 7.3, I have shown three categories corresponding to the general types of communication patterns discussed previously (of course, others might also be included). The potential advantage of such an approach is that it is simpler in terms of programmer effort than custom predictors and the quality of prediction is likely to be better than a general, application independent, predictor because these predictors can utilize some knowledge from the programmer to select a more appropriate type of predictor. Using the approach the programmer only has to decide what category the application would fall under. The application programmer presumably has sufficient knowledge of the application to tell whether the communication involved is, for example, due to programming style, data structure or algorithm behaviour. This is a reasonable assumption even for "maintenance" programmers or others who are using the code but may not have written it and are therefore not entirely familiar with the intricate

details of the code and its communications.

7.3 Requirements for Prediction in a LAM/MPI Environment

There are some assumptions that are required prior to using such predictors. First, we must assure that the initial allocation of processes nodes will result in relatively good performance for a period of time to permit collecting quality sample data that can be used in determining the patterns of communication among the processes and coming up with an accurate grouping of processes that frequently communicate with each other.

The predictors are, further, dependent on how well we can extract meaningful information (patterns of communication) from the collected raw communication information from the application. Clearly, we need a mechanism that can accurately identify repeated communication patterns from the raw communication information to come up with groups of communicating processes.

Pattern matching/discovery algorithms can be used to identify frequently occurring patterns of communication in the collected raw communications data from executing MPI applications. These patterns of communication contain information about the groups of processes that communicate with each other and temporal and spatial analysis of these patterns of communication can allow us to make predictions of future patterns of communication which would then allow us to group related processes for checkpoint and co-migration.

My overall goal in this part of the thesis is to be able to produce a generally applicable predictor for a wide range of MPI applications having regular communication patterns and specialize it only if/when necessary.

7.4 Selecting a Matching Algorithm

Any non-trivial MPI application involves communication between processes at some point(s) in its execution. Different MPI applications, as we have seen, have different communication requirements and patterns depending on programming style, data structure, algorithm chosen, etc. We are interested in deriving meaningful information about possible future inter-process communication patterns from the observed communication behavior of an MPI application. Then, at the time when we need to do a partial checkpoint/migrate we should be able to predict what the groups of frequently inter-communicating processes in the application are. If we can accurately determine these groups by studying past communication behavior of the processes using collected communications information, we can make a prediction about the processes that will be affected if a poorly performing process is checkpointed/migrated, so we can checkpoint and migrate the entire affected group along with the poorly performing process to help ensure future efficiency.

The communication behavior of some MPI applications may change over the period of its execution. A classic example of this is computing the FFT which is well known for its butterfly pattern of communications (as described in Section 7.1). We need a matching algorithm that can tolerate such changes in recorded patterns of communication and can identify both exact matches in the patterns of communica-

tion where they exist as well as important partial/imprecise matches of patterns of communication. Such an algorithm will be able to tolerate minor differences in communication behavior while still identifying key, recurring patterns. Exact and partial matching might also be combined and used to derive some useful information. For example, if we find a small exact match such as "*a b c*"[1] and a long partial match such as *a b c* ? ? *a b c* (where '?' indicates variation in communication), then we might realize that *a b c* is a pattern that has a tendency to repeat after some interval.

There are many areas in computer science that employ pattern recognition to solve specific problems. These include bioinformatics, as well as machine learning and data mining techniques, among others. Examples of such algorithms and their merits and limitations are briefly reviewed and assessed before discussing my selected approach.

7.4.1 Bio-informatics: Matching Algorithm

In genomic sequences, patterns correspond to functionally or structurally important elements encoded in the DNA [BVL03]. In problems such as protein structure and function prediction, drug target discovery, etc. pattern matching and discovery have become important bio-informatics tools. There are several bioinformatics related matching algorithms among which TEIRESIAS [RF98; BVL03] is chosen as a representative example. Such algorithms have the general characteristic of finding exact as well as long but imprecise repeating sequences.

The TEIRESIAS algorithm is applied to genomics sequences to discover sequence similarities in primary structures related to proteins and genes. The algorithm detects

[1]Here each letter might represent a pair of communicating MPI processes.

patterns in a set of input sequences without using "alignment" and "enumeration" techniques. By *aligning* two or more sequences, for example protein coding sequences, evolutionary relationships between a set of input sequences can be studied by analyzing the points of matches and mismatches when the sequences are aligned. *Enumeration* is an exhaustive search technique where all possible patterns are generated that satisfy a certain constraint and then checked for occurrences in sequences. Based on the occurrences, scores are given to patterns. Patterns that have high enough scores compared to a given score threshold are then chosen as the final output patterns.

TEIRESIAS, uses a pruning pattern enumeration technique which is based on depth first search of a tree of all possible sequences with pruning of branches which do not yield "supported" (sufficiently frequently occurring) patterns [BVL03]. The tree and its branches are constructed along the way by starting with a length 1 pattern with sufficient support and subsequently extending it by suffixing with all possible "residues" (additional symbols) from the alphabet of available nucleotides, pruning the branch (further down from there) when the pattern is no longer "supported". Once a pattern with sufficient support is reached which can no longer be extended, the tree node is kept as a found pattern with sufficient support.

The worst case running time of TEIRESIAS is exponential in the number of patterns it generates but it is guaranteed that all "maximal" patterns are found. Maximal patterns are those patterns that are more specific and from which other sub-patterns can be derived by removing characters from both ends and replacing characters with wild cards. The main concept behind the algorithm is that if P is an $< L, W >$ pattern[2] which occurs in at least K genomic sequences, then all the sub

[2] An $< L, W >$ pattern is a pattern in which all the sub patterns of a given pattern have lengths

patterns of P also occur in at least K sequences and they are also $< L, W >$ patterns.

There are two phases in the TEIRESIAS algorithm, the scanning phase and the convolution phase. In the scanning phase a pruned exhaustive search (as just described) is used and all the $< L, W >$ patterns with L non-wildcards are found that occur in at least K sequences (L, W and K are chosen by the user). The search starts with the smallest possible pattern that exists in at least K sequences and then the pattern is extended by suffixing each residue from the alphabet in each step as long as the extended pattern is still found in at least K sequences. The convolution phase is where the patterns found from the scanning phase are "glued together" to create new extended patterns that also occur in at least K sequences. Patterns P and Q can be glued together only when the suffix, which is exactly $L-1$ non-wildcard symbols, of P is equal to the prefix, which is exactly $L-1$ non-wildcards, of Q. That is, the $L-1$ non-wildcard symbols overlap. The occurrences of this pattern need not be "scanned" for again because the occurrences of P and Q are used to build the new pattern and we know that they each occur in at least K sequences.

The TEIRESIAS algorithm is, essentially, an algorithm for creating longer patterns from known short ones. This algorithm could also be effectively used for identifying useful patterns in collected inter-process communication information. The primary change needed would be to replace symbols for nucleotides by symbols representing communication between pairs of processes. Also, instead of considering multiple sequences of symbols to find sequence similarities, we would consider only a single sufficiently large sequence of symbols and try to discover repeated patterns of

W or more and contain at least L residues. A residue is a set of symbols from an alphabet, in this case, the set of nucleotides in DNA.

communications that occur within the sequence (i.e. over time).

7.4.2 Machine Learning Methods

There are many machine learning methods applying a range of techniques that might also be useful in recognizing patterns in communication data. Hidden Markov Models [Gol08] are one such method.

A Hidden Markov Model (HMM) is a statistical model that has hidden "Markov" states that change over time and a set of observable states that are related to the hidden Markov states. More precisely an HMM consists of hidden states (which can be easily described by a Markov process[3]), observable states (which are present as observations), a pi vector (which describes the probability that a model is in a given state at time t=1), a state transition matrix (given the previous state, this matrix contains the probability that a hidden state will be chosen next) and a confusion matrix (given that a model is in a hidden state, the confusion matrix defines the probability of observing a given observable state) [BVL03; Boy10]. An HMM can be described by a triple <pi, state transition matrix, confusion matrix>.

Usually HMMs are used to solve pattern recognition problems using "evaluation" and "decoding" techniques. Evaluation is a process in which, if an HMM is known, then the probability of observed sequences can be found (using a "forward" algorithm) while decoding is a process in which, if an observed sequence is known, then the probability of the hidden states that generated it can be found (using the Viterbi algorithm [Vit67; Jr.73]). Finally, if the sequence of observations are known, then by

[3]A process in which a move from one state to another is entirely dependent on the previous state(s) and the choice is made probabilistically.

using them we can generate an HMM and this results in a learning technique.

In pattern recognition given an HMM and a sequence of observations we can determine if the sequence was generated by the given HMM. We can also compute the most probable path for a sequence using the HMM and this path then represents the probability of the pattern occurring in the sequence.

For application in learning communication patterns, an HMM can be used because we have observations in the form of who communicated with whom in the recent past and we can construct an HMM model using the observations which can then be used to make predictions of likely upcoming patterns of communications.

7.4.3 Data Mining Techniques

There are many data-mining techniques that are used to find associations between the discrete items stored in various sources of data such as transactional databases, telecommunication network logs, etc. Apart from finding simple associations such as, for example, event B occurs after event A, we can also find more detailed information such as event B occurs after event A within a specific time period. Sequential pattern mining [ZB03] is one technique that deals with such problems, that could be of use in finding associations between communication events because the communication information collected is sequential (and time-stamped).

Sequential mining is a technique in which sequential patterns (a sequence of discrete events that have temporal or spatial ordering) are extracted based on some predefined "support" threshold. For example, in a purchase analysis system, all the transactions (represented by a *sequence* consisting of a list of items purchased)

made by a customer having temporal/spatial ordering taken together are called a *customer-sequence*. A *sequence* is "supported" by a specific customer if that *sequence* is contained in the *customer-sequence*. The overall support of a *sequence* is the fraction of all customers that support that *sequence* divided by the total number of customers [ZB03]. The two most common types of algorithms used in sequential mining are the Apriori-based and the Pattern-Growth based algorithms. GSP [SA96] and PrefixSpan [PH01] are good representative algorithms of each type, respectively.

Apriori-based algorithms are based on candidate generation (generation of all the sequences that may be frequently occurring) and scanning of the entire data-set to check if each candidate has enough support (to determine a frequently occurring sequential patterns). To generate K length frequent sequences, the information from K-1 length frequent sequences from the previous step of the algorithm is used. Each sequence that is generated is then checked for validity (whether it is frequent or not) by scanning the entire database [AO04]. To increase the efficiency of the basic apriori algorithm, some techniques are used such as storing the generated candidates in a hash-tree or creating K sequences only when some criteria is met on the previous step's sequences and pruning those sequences that do not meet some specific criteria. Apriori-based algorithms use breadth first search in a tree of generated candidates to count the candidate item sets efficiently, starting at the root node and exploring all the neighboring nodes (for counting candidate item sets) followed by exploring the neighboring nodes of those neighboring nodes and so on.

Pattern-growth methods such as PrefixSpan are methods in which candidate generation is entirely eliminated and the focus is maintained on a specific portion of the

database. The basic idea behind such algorithms is to recursively construct patterns by focusing only on a projected[4] database. Patterns that have a prefix subsequence, α, are examined by projecting only their corresponding suffix subsequences into the projected database which is then called an α-projected database. In each step of the algorithm, the search space is reduced because in each step the new projected database is scanned to search for only the frequent events that can be added to the end of the prefix α to form a sequential pattern and each resulting pattern is again output to represent a new prefix (α) projected database and explored in the same way [Aho01].

Sequential pattern mining could be used to discover meaningful patterns of communication in collected communication information since the nature of the information is sequential.

7.4.4 Assessment of Matching Techniques

The various algorithms discussed are all potentially useful in determining repeating communication patterns from collected communications data. HMM, is a probabilistic approach and employs learning techniques before any meaningful results can be achieved. Assuming communication patterns that vary with time, we need to be able to predict on the fly when a checkpoint request arrives based on an analysis of the summarized communication information of an application. We also have a restricted window between checkpoint arrival time and when a prediction must be

[4]Projection is a technique to make the data base much smaller for the next pass of the algorithm by avoiding generation of candidates and recursively partitioning the database according to their prefix (which can progressively increase starting from length-1, length-2, so on until no more length-k prefix patterns can be generated.). This speeds up the next pass and subsequently the algorithm also gets sped-up[ZB03].

made. Because HMMs require learning before any useful information can be derived, their use would be limited early in a program's execution. More importantly, it is unclear how/if they could be used to handle applications with changing communication patterns in a timely fashion. Many models would have to be generated and it would be unclear when to apply a given model.

Data mining techniques are capable of discovering meaningful associations between discrete events in sequential data which should be useful in understanding the temporal behavior of communication patterns. Data mining techniques can recognize precise matches between specific items but we are also interested in matching long and possibly imprecise patterns of repeated communication from MPI applications. It is unclear how recognizing such repeated *sequences* would be done with standard data mining techniques. Further, for some algorithms, the repeating communication pattern information in the collected communication information might be far apart and this might be difficult to handle as well.

TEIRESIAS is known for discovering *maximal* patterns, ranging from small precise patterns to long imprecise patterns (those which have *don't care* symbols (.) in them). The benefit of knowing long imprecise patterns is two-fold. First, if a small precise pattern is part of longer imprecise pattern then we get the assurance that the small precise pattern is important because it is occurring after a possible interval of *don't cares*. Secondly, we also get to know about the repetition interval of a pattern. An example of precise and imprecise patterns is shown in Figure 7.4 where two different sequences of symbols (shown at the top of the Figure) were input to TEIRESIAS. We can see from the Figure that the precise pattern [3 4 5] is part of larger imprecise

pattern and this imprecise pattern occurs twice. The pattern [3 4 5] is repeated in the first sequence after 7 "random" symbols and also in the second sequence after 7 symbols as well. Overall, [3 4 5] is the most frequently occurring pattern.

```
3 4 5 4 6 8 45 344 5 6 3 4 5 8 8 7
3 4 5 6 4 8 54 433 6 5 3 4 5 7 8 8

    4: [3 4 5]
    3: [3 4 5 . . 8]
    2: [3 4 5 . . 8 . . . . 3 4 5 . 8]
    2: [4 . . . . 5 . . . . 8 8]
```

Figure 7.4: Long Imprecise Patterns

Apart from finding the most frequently occurring pattern in one or more sequences TEIRESIAS also creates an *offset list* (for more information refer to Appendix B, Figure B.3) which is especially while making predictions because by using the *offset* and *length* of a pattern we can easily determine patterns that are close to the checkpoint arrival time (as described in detail later).

The capability of TEIRESIAS to discover and report small exact as well as long imprecise patterns in a sequence of communication events represented by unique symbols makes it an attractive choice for discovering useful data from communication information collected from MPI applications. Not only does its support for imprecise matching provide the flexibility needed to deal with minor changes in parallel program communications behavior but it can easily deal with changing communication patterns by matching recent behavior against *multiple* previously seen patterns.

7.5 Implementation and Use of TEIRESIAS

I implemented the TEIRESIAS algorithm for pattern discovery and used it to detect patterns of communicating processes in a sequence containing collected communication information about all the processes in an application[5]. Each pair of processes (between which communication may occur) is mapped to a unique symbol using a simple mapping function. The mapping function is defined as: $\Theta < p1, p2 >= (p1 \times NumProcs) + p2$, where $p1$ and $p2$ are the process ranks, and *NumProcs* is the total number of processes involved in the computation. For example, if *NumProcs* is 16, $p1$ is 1 and $p2$ is 8 then using the mapping function for a communication from $p1$ to $p2$ maps to the unique symbol $(1 \times 16) + 8 = 24$. Any pair of processes involved in communication, is thus mapped to a unique symbol. These sequences of symbols are collected while running applications. My instrumented version of LAM/MPI collects this information at a very low level (the RPI level) with low overhead. LAM/MPI's RPI code already maintains the information about who has communicated with whom, using it at the time of checkpoint to clear any inflight messages. I simply extract this information, map it to symbols and store them in memory. Each unique symbol generated is recorded along with a time-stamp indicating when the communication event occurred.

Periodically, TEIRESIAS is called to summarize this "raw" data, then the collected data is discarded and the much smaller summary of patterns found by TEIRESIAS is maintained. For long running applications, it is wasteful to maintain the raw data because all we are interested in is the patterns of communication in sampling

[5]The communication information is collected "on-the-fly" in memory using an instrumented version of the LAM/MPI Request Progression Interface (RPI) code I developed.

2 13 28 6 22 7 11 2 13 28 6 22 37 21 2 13 28 6 22 1 7 2 13 28 6 22 7 2 5 1 38 38 13 7 2 5 1 38 19 20 7 2 5 1 38
19 31 7 2 5 1 38 10 23 19 3 29 22 55 10 23 19 3 29 11 21 10 23 19 3 29 30 11 10 23 19 3 29 1 29 21 47 46 19 55
1 29 21 47 46 21 11 1 29 21 47 46 14 7 1 29 21 47 46 29 47 39 31 19 46 7 29 47 39 31 19 38 30 29 47 39 31 19 6
13 29 47 39 31 19 55 12 23 38 4 12 19 55 12 23 38 4 1 31 55 12 23 38 4 4 31 55 12 23 38 4 1 39 28 2 6 3 4 1 39
28 2 6 23 55 1 39 28 2 6 37 30 1 39 28 2 6 15 39 22 6 29 38 47 15 39 22 6 29 47 1 15 39 22 6 29 22 46 15 39 22
6 29 31 22 46 19 13 7 19 31 22 46 19 13 39 6 31 22 46 19 13 1 2 31 22 46 19 13 10 37 4 12 13 7 47 10 37 4 12
13 1 3 10 37 4 12 13 38 1 10 37 4 12 13 39 46 12 23 2 6 10 39 46 12 23 2 31 19 39 46 12 23 2 38 14 39 46 12 23
2 31 38 14 37 10 37 46 31 38 14 37 10 30 5 31 38 14 37 10 7 1 31 38 14 37 10 12 21 30 4 31 55 29 12 21 30 4 31
1 19 12 21 30 4 31 10 31 12 21 30 4 31 28 30 7 5 22 11 13 28 30 7 5 22 10 19 28 30 7 5 22 5 7 28 30 7 5 22 19
46 14 38 1 23 1 19 46 14 38 1 28 6 19 46 14 38 1 13 22 19 46 14 38 1 20 5 13 38 21 4 12 20 5 13 38 21 11 28 20
5 13 38 21 55 12 20 5 13 38 21 6 39 14 21 47 30 55 6 39 14 21 47 3 2 6 39 14 21 47 7 23 6 39 14 21 47 46 2 11
13 38 5 14 46 2 11 13 38 5 20 46 2 11 13 38 7 22 46 2 11 13 38 13 55 37 3 23 55 29 13 55 37 3 23 3 22 13 55 37
3 23 13 15 13 55 37 3 23 20 46 4 3 5 7

Figure 7.5: Example of TEIRESIAS Sequence of Symbols

"intervals" corresponding to periods of raw data collection. When TEIRESIAS is run on the raw data, it discovers the patterns of communication that can be used in making future predictions of upcoming patterns. These discovered *patterns* of communication are used when a prediction must be made. Because we only need to analyze the smaller summarized data rather than the entire raw data collected, the prediction algorithms are able to run faster.

7.5.1 Validation of my TEIRESIAS Implementation

To test the correctness of my TEIRESIAS implementation I created a program that generates regular but changing patterns of symbols (i.e. a form of synthetic data for testing). The program takes as input the length of sequence to be generated, the number of processes involved in the would-be computation, the length of symbols before a repeated pattern changes (to signify the transition from one pattern to another), the length of the repeating pattern (length of a pattern that is repeating before transition to another repeating pattern), the gap size between repeated patterns, the size of the sampling period (which divides the entire data sequence into smaller

pieces that were fed to TEIRESIAS) and the time at which a hypothetical checkpoint request arrives. An example of a generated sequence with TEIRESIAS recognizable symbols is shown in Figure 7.5. This simple example sequence was generated using parameters of 500 as the length of the sequence, 8 as the number of processes, 25 as the length before a pattern changes, 5 as the length of the pattern, 2 as the gap size between repeated patterns, 20 as the sampling period and 280 as the checkpoint arrival time. The colors in Figure 7.5 are used to highlight the repeated patterns. The gaps between the repeated patterns were filled with randomly generated symbols.

The sequences created using this program attempt to reflect communication data from an application which has regular but changing patterns but in an easily controlled and understandable way. The length of the repeating patterns and the gap size controls the regularity of patterns and the length of symbols before a repeated pattern changes, controls the change in pattern to another repeating pattern after the previous pattern has repeated for some time. The gap size between repeating patterns is used to insert random symbols between the repeated patterns to model minor variations in communication patterns. By creating such sequences, I was able to successfully test the correctness of my implementation of TEIRESIAS by checking the patterns found by TEIRESIAS and manually verifying them to see that all the patterns were detected. Experience with the patterns found and my analysis of the patterns also helped provide a basis for my initial prediction mechanisms.

Each symbol in the example (Figure 7.5) represents a pair of processes. Given such a synthetic sequence of communication events, the sampling period (one of the inputs) determines the length of the sub-sequences to be fed to TEIRESIAS at one

time. This breaks the sequence of communications information into smaller pieces reflecting periodic use for long-running applications in the real world. An application may run for some time before a checkpoint request arrives. At that time, the patterns of communication from the execution of the application prior to the arrival of the checkpoint request must be analyzed to determine who communicated with whom and thus to form groups of frequently communicating processes. Regardless of when a checkpoint request arrives, there must be some data available to base a prediction of future communications on. This is why sampling must be done in intervals/periods. Not only does collecting the communication information periodically ensure data availability, but TEIRESIAS can be run periodically to summarize the patterns of communication from the collected raw communication information (as described earlier) thereby saving space in memory and minimizing TEIRESIAS' runtime. The repeated patterns in each sampling period can then be quickly analyzed to predict the likely patterns of upcoming communication when a checkpoint request arrives.

Division into sampling periods is shown, abstractly, in Figure 7.6 where we see that the repeated sequences may cross sampling periods. In sampling periods 1 and 2, and 2 and 3, we see that the "$-$" and "$\sim$" patterns overlap sampling boundaries (i.e. the same pattern of communication is continued forward to the next sampling period). This must not impact our ability to detect the correct repeated patterns. To deal with the overlap of communication patterns between any two sampling periods we could create a sequence from both sampling periods and feed it to TEIRESIAS which could then find the repeated patterns. Unfortunately, doing such merging of

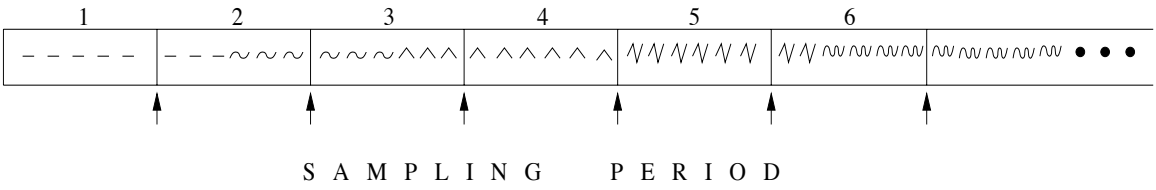

Figure 7.6: Example of Repeated Patterns Crossing Sampling Periods

adjacent samples would only create larger samples that might also experience the same problem. Further, we cannot combine all samples together for reasons already discussed[6]. In general, we need to make accurate predictions for a running application on the fly so the only information we will have is the communication information from the samples since the start of application execution up to the point when a checkpoint request is made. Any predictor created must operate with this data as input and deal with potential overlap between sampling periods.

7.5.2 Grouping Information and Basis for Predictors

Using the TEIRESIAS recognizable symbol sequence in Figure 7.5, we can examine the results obtained from feeding the per sampling period sequences to TEIRESIAS and formulate a basis for creation of the needed prediction mechanisms. The total number of sub-sequences fed to TEIRESIAS was $Lengthofsequence/samplingperiod$ (i.e. $500/20 = 25$) smaller sequences. A snapshot of the patterns discovered using TEIRESIAS is shown in Figure 7.7. Figure 7.7-A shows the discovered patterns in the smaller sub-sequences corresponding to each sampling period. Symbols in red show the repeated patterns in the sub-sequences and symbols in green show the

[6]The only case where samples might be effectively combined and analyzed would be if we had results from a previous run of the application.

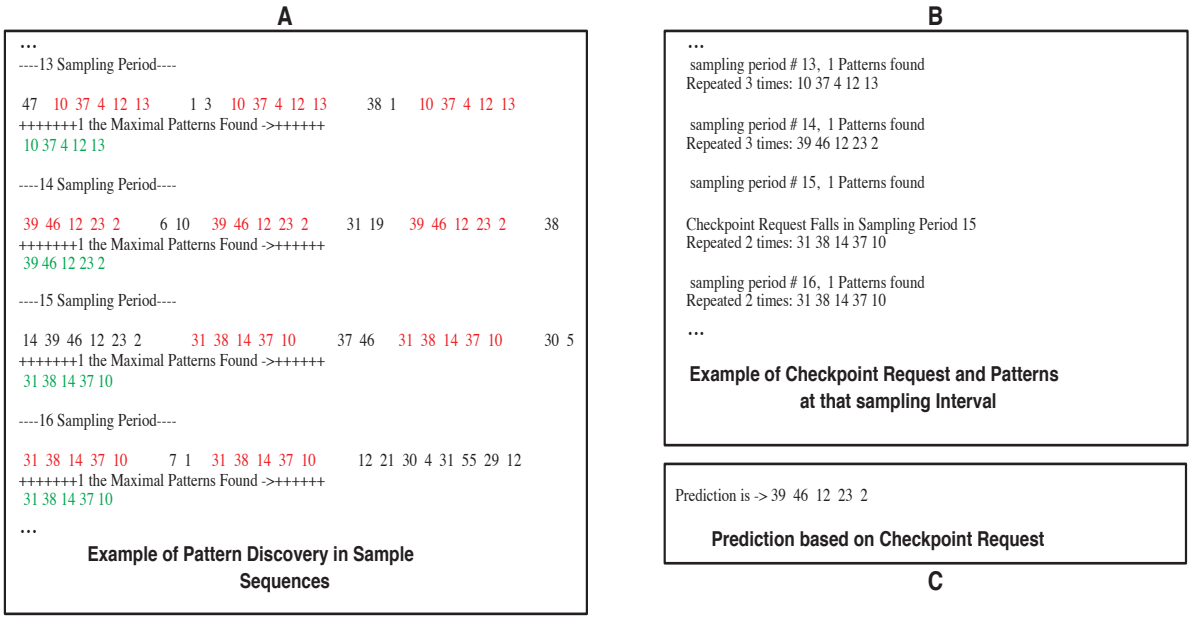

Figure 7.7: Example of Running TEIRESIAS on the Example Synthetic Communication Sequence

results obtained after feeding the respective sequences to TEIRESIAS. In this simple example, the repeated patterns are found without any inaccuracy. Figure 7.7-B shows the arrival of a checkpoint request at time 280, a randomly chosen number between 0 and 500 (the length of the example sequence). Since each sampling period consists of 20 symbols the checkpoint request falls in the $280/20 + 1$=15th sampling period (counting from 0). We can see the patterns found in the preceding sampling period in both Figures 7.7-A and B: "39 46 12 23 2" in the 14th sampling period. Since the checkpoint request time is 280, which makes is fall right at the start of the 15th sampling period ($280 \bmod 20 = 0$), a prediction can be made by choosing the last repeated pattern from the previous (14th) sampling period which is: 39 46 12 23 2. The upcoming pattern in the 15th sampling period, as shown in Figure 7.7-A and B, is also 39 46 12 23 2. Thus, a first predictor could be based on the very simple idea

of assuming the recent most communication pattern will continue in the immediate future. I call this predictor, a *basic* predictor.

This *basic* predictor, however, might not always give accurate results, as it did in the simple example above, because communication patterns change over time and it might happen that the checkpoint request might arrive right at the point where the communication patterns changes. For example, in Figure 7.7 A, if the checkpoint request arrived right at the beginning of sampling period 14, then the *basic* predictor would make its prediction based on the most recent repeated pattern in sampling period 13: 10 37 4 12 13, assuming it would continue. This would be incorrect because from the 14th sampling period on, the communication patterns change.

Another more accurate *specific* or "*historic*" (since it is based on analysis of past communication information) predictor could be created that considers communication data from a larger number of previous sampling periods. This predictor might make more accurate predictions because it would analyze a larger number of past/historic communication patterns before making a prediction and potentially recognize the start of a previously repeated pattern right before the point of checkpoint request. Such a predictor could take the predicted pattern from the *basic* predictor and match it in the past sampling periods to find if the pattern has occurred in the past. If an exact match is found then the pattern that occurred immediately after this pattern in the past might be likely to occur now as well. Thus, it should be chosen as the new prediction. I call this type of predictor a *historic* predictor.

This *historic* predictor, however, looks for an "exact" match in the past, which is not guaranteed to be found due to the dynamic nature of communication of an

application. In such cases, we might want instead to be able to find the closest match possible to the base pattern in the past. We can use the concept of Levenshtein distance to create, yet, another predictor which I call as *Levenshtein* predictor. The Levenshtein distance [Lev66] is defined as follows: for any string $s(i)$ where i represents the ith character in the string, for any two characters a and b,

$$f(a,b) = \begin{cases} 0 & \text{if a = b} \\ 1 & \text{otherwise} \end{cases}$$

For any two strings (or sequences in my case) of length n and m we develop an $n+1 \times m+1$ matrix M and the Levenshtein distance $L(s_1, s_2)$ is obtained from $M(n+1, m+1)$. The matrix M is filled by using the following equation,

$$M(i,j) = minimum(M(i-1,j)+1, M(i,j-1)+1,$$
$$M(i-1,j-1) + f(s_1(i), s_2(j)))$$

The first row and column of the matrix M are filled initially with the values $0, 1, 2, ..., n$ and $0, 1, 2, ...m$ respectively. My proposed *historic* predictor would look for an *exact* match of the pattern from the basic prediction and if the pattern has never occurred in the past it would fail to predict. Whereas, using a predictor based on Levenshtein distance we can look for a partial match in the past. The lesser the distance between the patterns we get, the better the match we have. My *Levenshtein* predictor guarantees to make a prediction.

7.6 The Initial Predictors

Based on the predictor types from Section 7.2 and my analysis of TEIRESIAS results described in Section 7.5, I propose two categories and three predictors that are

expected to meet the requirements of the types of applications discussed in Section 7.1. The first category is the "basic" category comprised of my proposed *basic* predictor and the second category is the "historic" category comprised of my *historic* and *Levenshtein* predictors.

***basic* Predictor:** The *basic* predictor is the base for the other two predictors. Basic prediction is done using only data obtained from the "current" sampling period (in which the checkpoint request arrives). I find the pattern in the current sampling period which has occurred at-least twice before, and is closest (in time) to, the checkpoint request arrival time. If I am able to find a pattern meeting this criteria I designate it to be my basic prediction (because it is the pattern of communication that is on-going and is likely to continue). If I don't find any pattern that meets the criteria in the current sampling period then, as a special case, I pick the last repeated pattern found from the previous sampling period and designate it as the basic prediction. This is needed when a checkpoint request occurs very near the start of a sampling period.

***historic* Predictor:** The basic prediction from the *basic* predictor is taken as the base prediction in the *historic* predictor. The basic prediction is taken as the input and checked for an "exact" match in previous sampling periods. If I find an exact match in a previous sampling period then the pattern that occurred immediately after the "matched" pattern becomes my prediction pattern. If there is no exact match found then the *historic* predictor fails to predict any thing and the basic prediction must be used. The idea behind the *historic* predictor is that an application that has regular, changing but repeating patterns will have repetition of patterns after some

time interval and will follow the same line of communication as it followed in the past, (e.g. Fast Fourier Transformation (FFT) application). In such cases, if we have collected enough communication information (i.e. the communication has crossed the point where repetition of patterns that have occurred in the past has started) then the likelihood of finding an exact match of the current pattern of communication in the past increases.

***Levenshtein* Predictor:** The *Levenshtein* predictoralso takes the basic prediction from the *basic* predictor. It works in the same fashion as the *historic* predictor with an added advantage that, it can make precise predictions even when an exact match is not found (using the *historic* predictor). I take the basic prediction and go through the collected communication information in previous sampling periods and compute the Levenshtein distance between the basic prediction and the patterns in each previous sampling period. I find the pattern that has the minimum Levenshtein distance and designate it to be the pattern that matched the basic prediction. The prediction is then made the same way as with the *historic* predictor, by finding the next pattern following the matched pattern.

Chapter 8

Experiments and Results

Two broad questions must be answered to determine the usefulness of partial checkpointing and migration for MPI applications as described in this thesis. First, can accurate predictions of upcoming communication patterns be made and used to group closely communicating processes. Second, assuming successful predictions, under what conditions (execution environment and application behavior) will partial checkpoint and migrate provide sufficient performance improvement to warrant its use. I will provide a reasonable but necessarily incomplete answers to these questions.

In this Chapter I assess the effectiveness of my partial checkpoint and migration system along with my group formation techniques for different types of communication patterns and MPI applications. I also assess the impact of the frequency of overload on computational nodes and migration cost (i.e. due to low bandwidth over longer distances between nodes and size of checkpoints) on the performance of my partial checkpoint/migrate system.

I developed three sets of experiments to assess my system. First, I created syn-

thetic communication data corresponding to various types of applications to test the initial behavior of my predictors and to provide a basis for any necessary tuning of my initial prediction algorithms. Synthetic data was chosen because of the well defined (hand crafted) communication patterns offered in synthetically generated data. This avoids the irregularities that may occur in the communication information from real runs of applications due to, for example, lack of complete synchronization between processes.

Following the first set of experiments and improving my predictors and grouping mechanism to handle anomalies seen in the initial results, I collected real-world MPI application communication data for my second set of experiments. The experimental environment for running MPI applications to collect the needed communication information consisted of a single dedicated machine (teak.cs.umanitoba.ca) in the Parallel and Distributed Laboratory which has my modified LAM/MPI code supporting partial checkpoint/restart and the required customized Linux kernel installed on it. My partial checkpoint/restart system requires BLCR to be pre-installed, after which LAM/MPI can configure itself at install time. Further, BLCR requires access to certain Linux kernel data structures at install time for which I built a custom Linux Fedora Core 4 kernel and installed it on teak. The need for dedicated hardware with a custom OS kernel prevented me from deploying a real distributed LAM/MPI runtime environment for execution of MPI applications and collection of communication data on a large scale. While this did not affect my ability to collect the needed communication data (which is based on MPI ranks, regardless of where the associated processes are run) and demonstrate my proof-of-concept implementation, it did

restrict the number of processes and the size of problems that my MPI applications could run. For this reason, the set of experiments I used to confirm the effectiveness of the predictors deal only with small scale problems. I make arguments for why the small scale results reported can scale to larger problem sizes later in this chapter.

Finally, the third set of experiments I performed was done using a simulated partial checkpoint and migration environment, which imitated the core functionality of my overall system, to assess the use of my system in a larger scale environment. In particular, I ran simulation experiments to assess the effect of frequency of overload on execution nodes and the cost of partial checkpoint/migration to nodes under different network characteristics on the overall performance of MPI applications requiring partial checkpoint and migration. For assessing the impact of the cost of migration on the overall performance of an MPI application I gathered network bandwidth and latency data for locations around the world, which is used to tie my results back to actual distributed environments.

The results of all three sets of experiments are now described, in sequence. Sampling period in the text, represent the length of the sampled communication data used to run TEIRESIAS and my predictors on.

8.1 Experiments, Predictors, and Results

In this section I discuss my experiments with synthetic and real communications data. The simulation results are left till the next section.

To evaluate the use of TEIRESIAS and the effectiveness of my initial predictors I first used synthetic data that reflects the expected patterns of communications from

some real applications. My goal was to understand how to use TEIRESIAS effectively for my problem of detecting groups of frequently communicating processes in an MPI application so the information can be used for partial checkpoint and migration (to ensure ongoing application performance in a distributed environment of shared computing resources). I constructed synthetic communications data for two kinds of applications. An application that had a highly regular pattern of communication based on a "Master/Slave" type of applications and an application based on "FFT (Fast Fourier Transformation)" that had a predictable but changing pattern of communications.

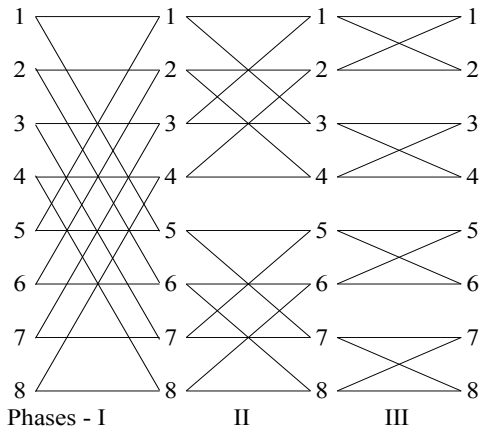

Figure 8.1: FFT Butterfly Communication

I first developed a synthetic dataset reflecting "Master/Slave" parallel application communication patterns. The code to generate this dataset takes in the number of processes as the input and designates the process with rank 0 as the master. The remaining processes, with ranks 1 to $n-1$, are considered to be slaves. Typically in a Master/Slave application there are three phases: the distribution phase where the master process distributes chunks of data to the slave processes, the computation

phase where all the slave processes compute and, if necessary, communicate with each other, and finally the gather phase where all the slave processes send their computed partial results to the master process. These three phases may then repeat to get results for several sets of data or to do iterative calculations on a single set. Rather than have no communication during the computation phase, I chose to have the communication that takes place between processes during the computation phase in my synthetic data structured such that each *odd* ranked process communicates with every other *odd* ranked process and each *even* ranked process communicates with every other *even* ranked process. This pattern of communication was chosen so there would be two distinct groups of communicating processes to ensure that TEIRESAIS would easily determine these groups. Further, since my data set reflects Master/Slave style communications there will be a burst of data sent out to all processes from the master in the distribution phase and a series of individual communications received from all the slaves by the master process in the gather phase. TEIRESIAS should be able to recognize these patterns as well.

My second program generates a synthetic dataset of FFT-like communication patterns. It takes in the number of processes and generates communications data that corresponds directly to the "butterfly" communication pattern that takes place during the computation of the FFT. This pattern is shown again in Figure 8.1, where the numbers represent the ranks of the processes and the lines connecting the numbers represent communications between the processes in each phase of the FFT (corresponding to each column of process ranks).

The two applications chosen have well structured, regular patterns of communi-

cation, which were sufficient for my initial assessment of my TEIRESIAS implementation and my initial predictors.

8.1.1 Example of Grouping and Prediction with Synthetic Data

In one experiment I fed TEIRESIAS the FFT-based synthetic data after dividing the generated dataset using a sampling period of size 37. The total number of sub-sequences fed to TEIRESIAS was $LengthOfSequence/samplingPeriod$ (i.e. $10000/37 = 271$).

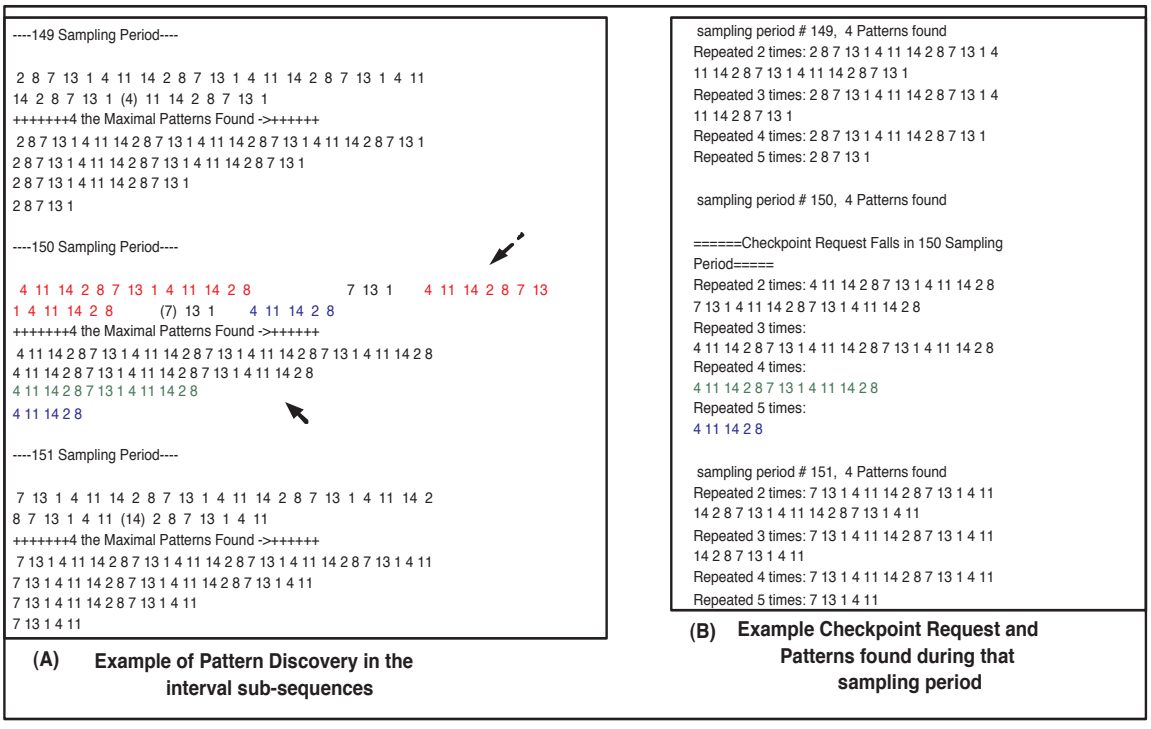

Figure 8.2: Example of Grouping Processes using TEIRESIAS

A snapshot of the patterns discovered using TEIRESIAS is shown in Figure 8.2.

Figure 8.2(A) shows some of the original sequence chopped into smaller sequences (determined by the sampling period) and the discovered patterns in the sub-sequences corresponding to intervals 149 through 151. The dashed line arrow shows the repeated patterns discovered in sub-sequence 150, the solid line arrow shows the prediction that my *basic initial predictor* made. Figure 8.2(B) shows the arrival of an arbitrarily selected checkpoint request at time "5542" (a randomly chosen number between 0 and 10000, the length of the sequence). Since the sampling period is "37" the checkpoint request falls in the $5542/37 + 1 = 150th$ sampling period (counting from 0). The patterns found in the $150th$ sampling period are shown in both Figure 8.2(A) and Figure 8.2(B). The most recent occurring pattern is used to make a prediction. The

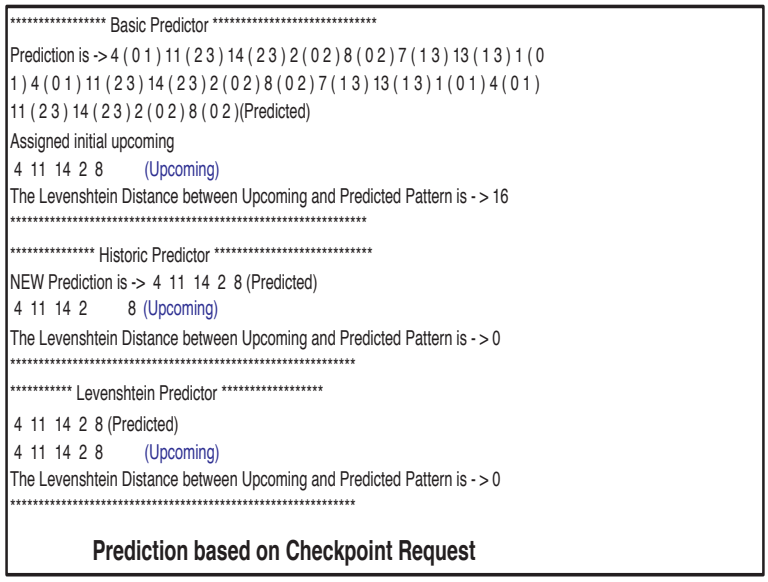

Figure 8.3: Predictions made by Initial Predictors

basic prediction is made by selecting the closest pattern (4 11 14 2 8 7 13 1 4 11 14 2 8) prior to the checkpoint arrival time, shown in Figure 8.2(A) in parenthesis. When choosing the pattern my predictor makes sure that the pattern is repeated at

least twice before the checkpoint arrival time and that it is the closest such recurring pattern to the checkpoint arrival time. As shown in Figure 8.3, the *basic* predictor, at the point where the checkpoint request arrives predicts the group of processes involved in communication corresponding to the selected pattern (4 11 14 2 8 7 13 1 4 11 14 2 8). I also used TEIRESIAS to identify the actual "upcoming" pattern to check whether the prediction I made using my *initial predictor* was accurate or not. Figure 8.3 shows the upcoming prediction discovered as well as the predictions made by each of my three types of *initial predictor*s.

The metric that I used for evaluation of the accuracy of my predictions is the Levenshtein distance between the predicted and the observed (i.e. upcoming) patterns. Figure 8.3 also shows the accuracy of prediction in terms of Levenshtein distance between the actual, observed pattern and the prediction made by each *initial predictor* (16 for the *basic* predictor, and 0 for the *historic* and *Levenshtein* predictors). The numbers in the braces, shown in Figure 8.3, 4(0 1) 11(2 3) 14(2 3) and so on, are the process ranks originally used to map to the corresponding TEIRESIAS symbol (e.g. $0 + 4 \times 1 = 4$ for communication from $P0$ to $P1$).

8.1.2 Original Synthetic Data Experiments and Results

I used my implementation of the TEIRESIAS algorithm together with my *initial predictors* (*basic*, *historic*, and *Levenshtein*, as described in Section 7.6) on the synthetic data that I generated. As can be seen in Figure 8.5, where I ran TEIRESIAS and my *initial predictors* over the synthetic communication data for the master/slave-type application, with input of $10,000$ as the length of the sequence, 8 as the number

of processes, 100, 200, 300, to 1000 as the sampling period sizes (each for a separate run) and 5541 as the checkpoint request arrival time, both the *historic* and *Levenshtein* predictors made accurate predictions. There was, however, unexpected behavior observed in the predictions made by my *historic* and *Levenshtein* predictors when the sampling periods sizes were 700 and 900.

The emulated checkpoint arrived in the $5541/700 = 8th$ sampling period, when the sampling period size was 700. But in an 8 processes Master/Slave application (used in this example), when the group of processes communicate with each other they produce 32 communication events occupying 32 consecutive positions in the collected communication information, before repeating this communication again. With 700 as the size of the sampling period in a sequence of length $10,000$ and with a pattern of communication completing in 32 consecutive positions, any pattern of communication that fully repeats itself occurs only after a gap of 7 sampling periods (i.e. in the $8th$ sampling period) after the pattern first occurred. For example, if a pattern 1 2 3 repeats itself and occupies 3 consecutive positions in the collected communication information, as shown in Figure 8.4, and if the size of the sampling

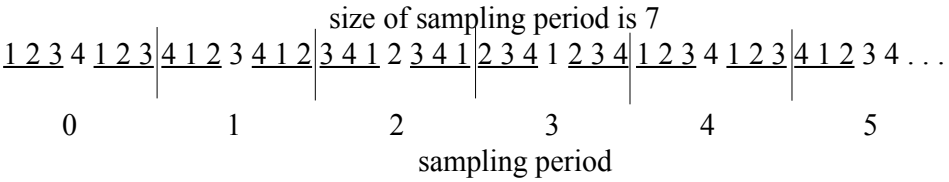

Figure 8.4: Unexpected Behavior Example

period is 7 then the pattern 1 2 3 will fully repeat itself only after a gap of 3 sampling periods. Further, If the checkpoint request arrives in sampling period 3 then if the

pattern, 2 3 4, becomes the basic prediction (since it is on going), the *historic* and *Levenshtein* predictors will fail because there is no match for the pattern available to be found in the previous sampling periods. Thus, my predictors will only work well when enough communication information has been collected so that the patterns have repeated themselves by the time of arrival of a checkpoint request. In my synthetic Master/Slave example, to have proper prediction we must have collected data for at least 8 sampling periods before any checkpoint request arrives. That is, the checkpoint request must arrive in the $9th$ or later sampling period, for the *historic* and *Levenshtein* predictors to be able to make accurate predictions.

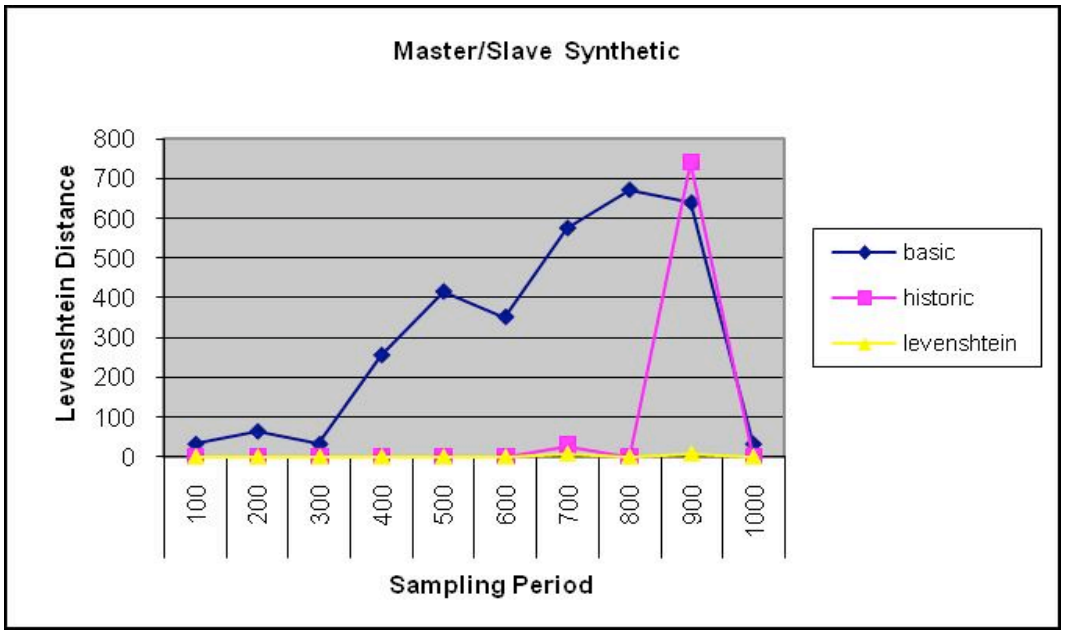

Figure 8.5: Prediction Accuracy for the Master/Slave Application Using Synthetic Data

I ran another experiment with the same basic scenario but this time using the FFT-like synthetic communication data and got similar results, shown in Figure 8.6, along with the same sort of unexpected behavior in predictions made by the *historic*

and *Levenshtein* predictors when the sampling period sizes were 700 and 900. The length of the sequence was the same, $10,000$, the sampling period size was again 700 and I set the checkpoint request arrival time in such a way that it arrived before the first pattern (that occurred in the first sampling period) repeated itself (3541 was the checkpoint arrival time). With the FFT-like data and 8 processes, the group of processes when communicating with each other produce 24 communication events occupying 24 consecutive positions in the collected communication data, before repeating this pattern of communication again. In this case, with 700 as the length of the sampling period, the checkpoint request arrived in the $3541/700 = 6th$ sampling period. But, with 700 as the length of the sampling period, $10,000$ as the length of the sequence and 24 as the pattern length, any pattern will repeat itself only after 6 sampling periods or in the $7th$ sampling period after the pattern first occurred. So I must collect data for at least 7 sampling periods before any checkpoint request arrives to permit accurate predictions.

We observe this unexpected behavior because the length of the sampling period divides the communication information sequence in such a way that the undiscovered repeated patterns are divided and placed in adjacent sampling periods resulting in loss of detection of patterns by TEIRESIAS. When repeated patterns occurred in one sampling period, then TEIRESIAS could discover them easily. Understanding this behaviour it is clear that, if we could have a mechanism to adapt the length of the sampling periods fed to TEIRESIAS at run time so that the length of the sampling period was an exact multiple of one cycle of communication events (for example, a multiple of 32 in the Master/Slave example), then my predictors would predict

accurately for all lengths of sampling periods.

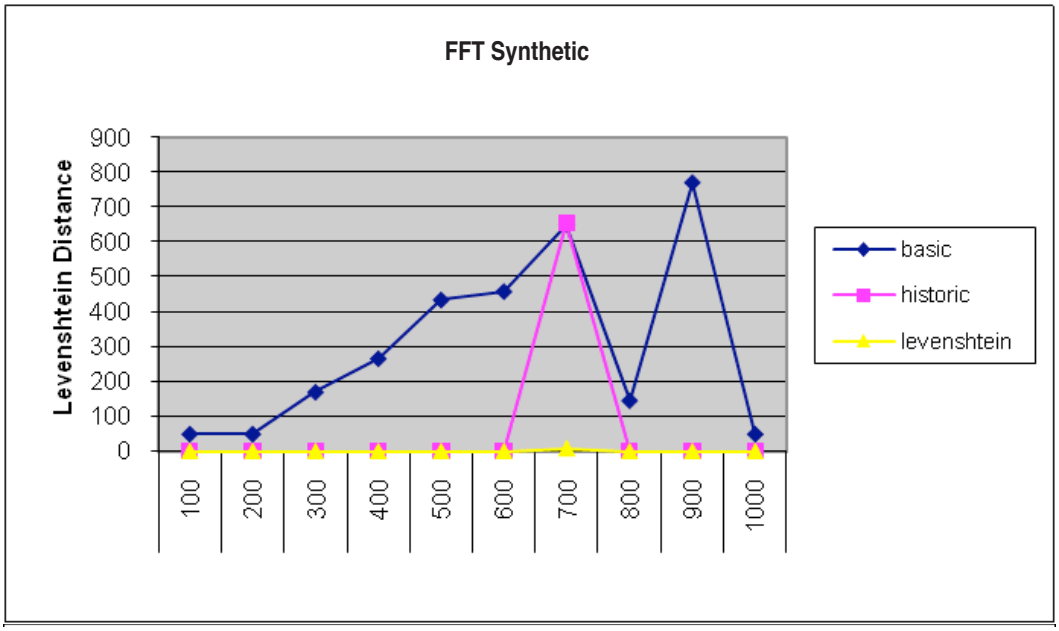

Figure 8.6: Prediction Accuracy for the FFT Application Using Synthetic Data

8.1.3 Adding Adaptivity to fix the Unexpected Behavior

To correct the unexpected behavior I observed in my first synthetic data experiments, illustrated in Figure 8.5 and Figure 8.6, I applied an approach of adapting the length of sampling period before making predictions. Theoretically, we know that, for example in an FFT application, if the number of processes involved is 8 then to complete one "butterfly" communication cycle the communication data collected will occupy 24 consecutive positions in the raw communication information before repeating the butterfly communication patterns for another set of data. Also, if the length of the sampling periods are set to a multiple of the length of one cycle of communication then the predictors will provide accurate predictions. Unfortunately,

we cannot know, a priori, what application we are running and how many processes we are using.

To try to avoid the problem of inaccurate prediction, I modified my predictor code to adapt the length of the sampling periods. Initially, my modified predictors start by using a sampling period length chosen randomly. They then adapt the length of the sampling period by matching a small part (length of sub-sequence equal to the number of processes involved) from the beginning of the collected communication information to find its next occurrence. This gives us the potential length of sampling period (which also reflects one cycle of communication events) because the point of next occurrence also signifies the point where next cycle of communication events for the application has started. Having an approximate length of sampling period I then calculate the adaptation factor for the initial sampling period by finding the difference between the initially chosen sampling period and the new sampling period found. To bring the adaptation factor in effect I then change (if required) the initially chosen sampling period to match the length of new sampling period found, by moving back a few (adaptation factor) communication events. The idea here is that Master/Slave and FFT application usually start with distribution of data in the beginning, and if the collected communication data represents the data collected from several runs of the application on several datasets then finding the beginning of the next distribution of data would give us the exact length of one cycle of communication events (communication data of one run of application on one dataset). This length is then used to determine how far back to move given a random sampling period length. An example of how the adaptation works can be seen by the following example, suppose one cycle

of communication of 16 process M/S application for a data size produces 45 communication events in the collected communication information (45 unique symbols), then a sequence of size 90 of such communication events would represent communications of 2 communication cycles of application and the communications will also have repeated themselves (assuming that the second dataset is of same size as for the first run). If the application is run several times on approximately the same problem size, then the size of the sequence of communication events will also grow by the multiple of 45 communication events. Lets suppose the application ran 5 times resulting the communication events sequence of length 225. If we choose 100 as the length of initial sampling period and adapt the sampling period then my adaptation algorithm will find a sub-sequences' next occurrence after 45 communication events giving 45 as the potential length of the new sampling period. Now 45×2 is 90 so the adaptation factor becomes $100 - 90 = 10$. The initial sampling period is then adapted by moving 10 position (communication events back) to match the exact multiple of one cycle of communication of M/S application to 90. Similarly, when the size of sampling period is chosen to be 125, the prediction algorithm adapts the length of sampling period 125 to 90 again since next multiple is 135 (size 150 would be adapted to size 135 and so on).

8.1.4 Fundamental Patterns

Another problem that I discovered in the first synthetic data experiments was my predictions commonly produced excessively long patterns consisting of repetitions of the same sub-patterns. I refer to such sub-patterns as "fundamental patterns" and

they should be recognized individually rather than as a long repeated sequence of them. For example, if there are patterns of the form A^2, A^3 and so on in a sampling period, then A is a fundamental pattern. To recognize fundamental patterns (i.e. the A's alone) in the patterns discovered by TEIRESIAS I incorporated the Knuth-Morris-Pratt (KMP)[1] pattern matching algorithm [KMP77] into my predictors to find the fundamental patterns in each sampling period. I use this algorithm to find any repeated occurrences of adjacent identical smaller patterns within a longer pattern discovered by TEIRESIAS in a sampling period.

I first run TEIRESIAS on the given sequence of communication information and get a summary of all the small and long patterns found in each sampling period. Then, in each sampling period I start from the smallest pattern and check if it is a fundamental pattern in the other, longer patterns (using the KMP algorithm, if a small pattern is A, I check if long patterns can be represented as A^2, A^3, and so on). I use KMP recursively to find all such small patterns in each sampling period.

Having found all the fundamental patterns in each sampling period, I get the frequently occurring smaller patterns derived from the TEIRESIAS results. I then run the same *initial predictors* (*basic*, *historic* and *Levenshtein*) to make a prediction but this time my predictors use the adaptivity technique to adapt the sampling periods and use the fundamental patterns instead of the direct results from TEIRESIAS. This results in what I call my *improved predictors*.

[1]KMP algorithm searches a given string for its occurrences in main string by by-passing the re-examination of the characters of the given string that were already examined earlier. The by-passing is done by using a partial-match table of observation containing information about the start position of new match in case of a mismatch. This table is built using a pre-search of the initial segments of the given string to match with some segments of the main string. Assuming the existence of partial match table, KMPs search time is O(n), where n is the length of the main string [KMP77].

The results that I present, from this point onwards, show the results of predictions using my *initial predictors* labeled "basic", "historic" and "Levenshtein" and the results of predictions using my *improved predictors* (with both adaptivity of sampling period sizes and recognition of fundamental patterns) labeled "basic(kmp)", "historic(kmp)" and "Levenshtein(kmp)".

8.1.5 Revised Synthetic Data Experiments

After incorporating fundamental pattern identification and adaptivity of sampling period, I ran both sets of predictors, the *initial predictors* as well as the *improved predictors* on my generated synthetic data for a number of test cases of modest, varying sizes and got improved results. I again fed my synthetic data based on master/slave and FFT applications to TEIRESIAS and to my predictors.

I ran the predictors by choosing various lengths of sampling periods each time and the results of predictions made by my predictors for each length of sampling period are shown in the graphs presented. Each "Sampling Period" on the X-axis represents the length of the sampling period chosen for one run of my predictors. The Y-axis represents the accuracy of my predictions in terms of "Levenshtein Distance" between the predicted and observed patterns. The Z-axis represents the prediction results obtained by running each kind of predictor.

The total length of the sequence for the M/S synthetic data was $10,000$ and a random checkpoint arrival time of 5541 was, again, chosen. Figure 8.7 shows the results obtained when I used the M/S application synthetic data for 4 processes. (The *improved predictor*s are shown with '(kmp)' suffix and the *initial predictor*s

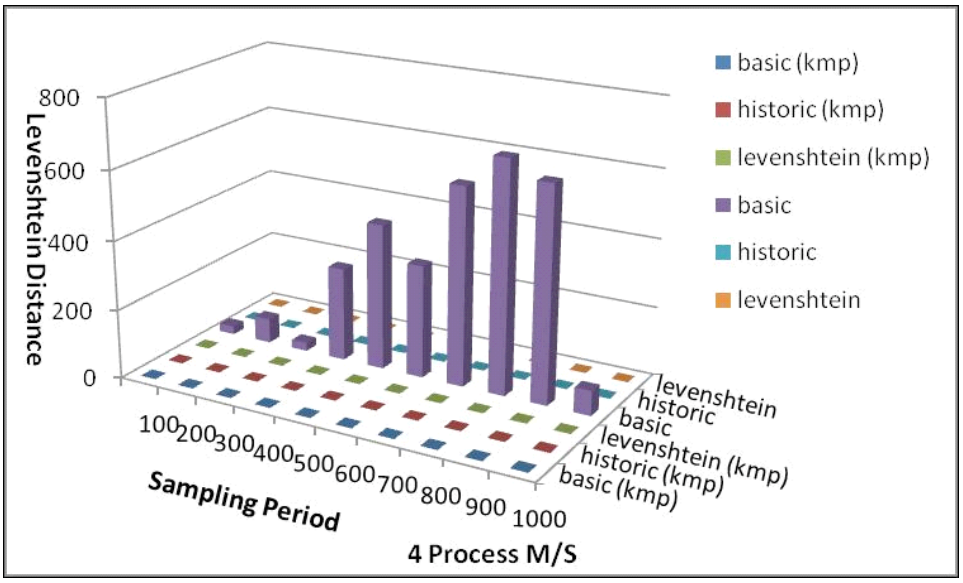

Figure 8.7: Results Using 4 Process Synthetic Data for the M/S type Application

are shown as before.) We can see that the Levenshtein distance between the prediction and observed patterns is 0 for all the *improved predictors* reflecting perfect predictions. Whereas, the *historic* and *Levenshtein* predictors performed quite well without adaptivity in sampling periods and recognition of fundamental patterns, the *basic* predictor performed very badly.

Figure 8.8, shows the results of running TEIRESIAS and my predictors on M/S synthetic data for 8 processes. The results again show that all the *improved predictors* predicted accurately. Using the *initial predictors* approach, my *historic* predictor behaved unexpectedly when the sampling period length was 900 in the synthetic data for the same reason as the case shown in Figure 8.5. The *basic* predictor from the set of *initial predictors* also performed poorly.

Figure 8.9 shows the results for 16 processes using the M/S synthetic data[2]. I

[2]I switched to a continuous data graph to highlight the variations in the data more clearly. The

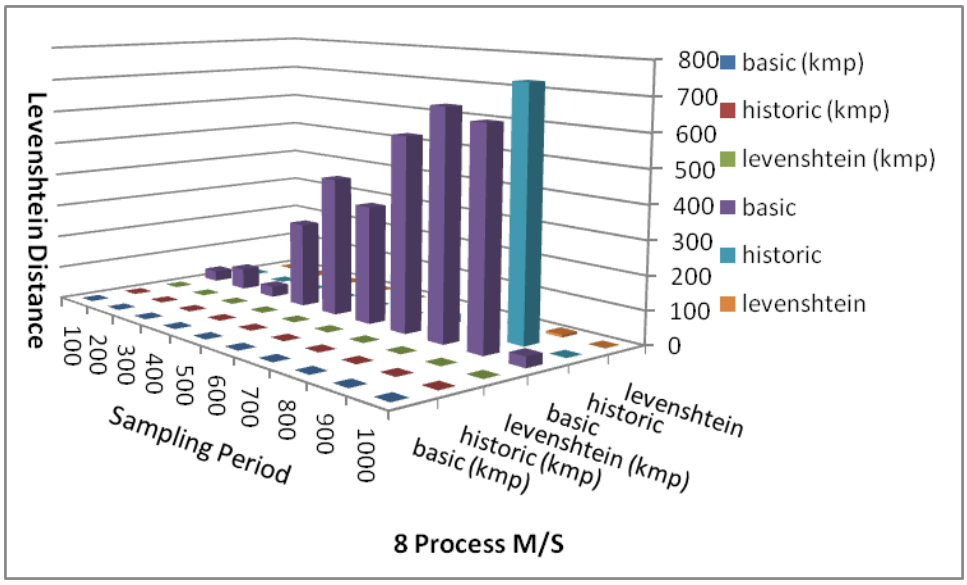

Figure 8.8: Results Using 8 Process Synthetic Data for the M/S type Application

sometimes got results signifying that no predictions could be made in the beginning of the experiment when the length of the sampling period is small, depending on when the repetition of the patterns start . All my *improved predictors* made accurate predictions but my *initial predictors* showed unexpected behavior. Of particular note in Figure 8.9, is that in the case when the sampling period length is 700 the *historic* predictor of the set of *initial predictor* failed to make any prediction because it could not find an exact match in the past, while the *Levenshtein* predictor could predict the pattern correctly. Thus, even when the initial *basic* and *historic* predictors fail, we may still see some reasonable predictions using the original *Levenshtein* predictor.

Figure 8.10 show the results obtained from running TEIRESIAS and my predictors on 4 processes using the FFT-like synthetic communication data. Both the *improved*

results, however, are not continuous.

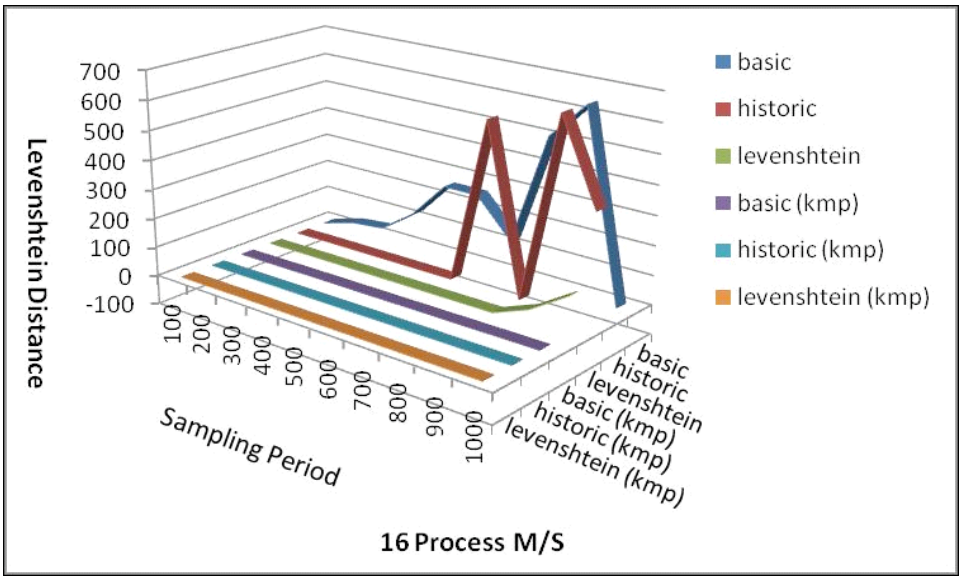

Figure 8.9: Results Using 16 Process Synthetic Data for the M/S type Application

predictors ((kmp) suffixed) and the *initial predictors* results are shown. Figure 8.10 clearly shows that both the *initial predictors* and the *improved predictors* could make accurate predictions with the exception of the *basic* predictor from the *initial predictors* set. It also suggests that the regular communication of FFT-like applications is more predictable when the number of processes involved are less.

The unexpected behavior in Figure 8.11 (showing the results for 8 processes running on the FFT-like synthetic data) when the sampling period length is 700 for my *historic* predictor from the set of *initial predictors* is because of the same reason as for the case shown in Figure 8.6. The *basic* predictor from *initial predictor* set also performs poorly as it is based only on the immediate (sampling period in which the checkpoint request arrived) sampling information.

Figure 8.12 show the results obtained by running TEIRESIAS and my predictors on a 16 process FFT-like synthetic data sequence. Again, my *improved predictors*

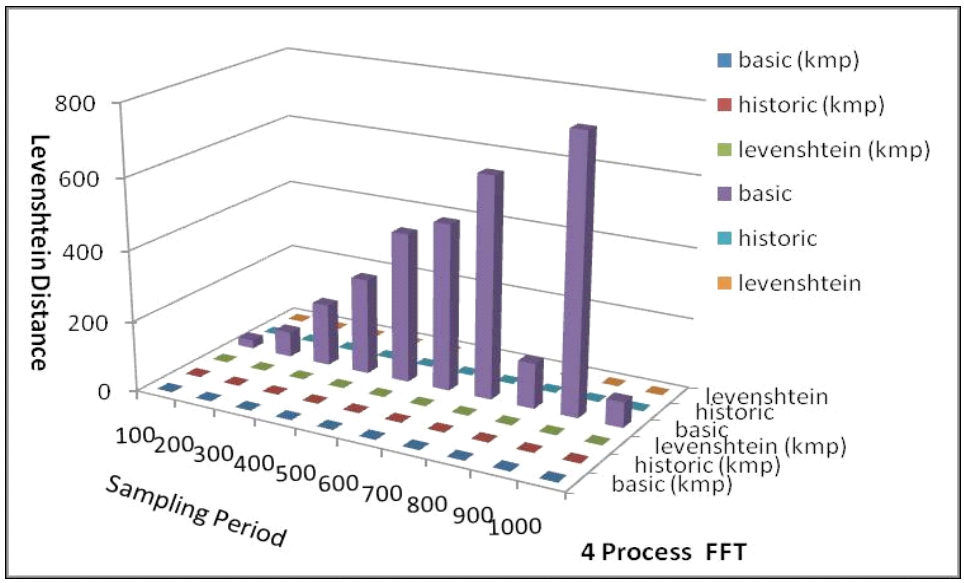

Figure 8.10: Results Using 4 Process Synthetic Data for the FFT type Application

made accurate predictions but my *initial predictors* behaved unexpectedly for some sampling period lengths.

I tested my predictors for fairly long running applications with large lengths of sampling periods (in terms of overall run time). With larger sub-sequences, TEIRESIAS gets to see more communication information representing significantly longer periods of application execution (which means the repetition of patterns is more likely to have occurred). But, I also wanted to test cases where the sub-sequences (lengths of the sampling periods) are small. Also, for the examples discussed, the checkpoint request always arrived after we had collected enough information for my *improved predictors* to perform well. With adaptivity of length of sampling period and fundamental patterns incorporated I wanted to test if my *improved predictors* will perform well or will produce results similar to what we have seen in Figure 8.5 and Figure 8.6 if the checkpoint request arrived early.

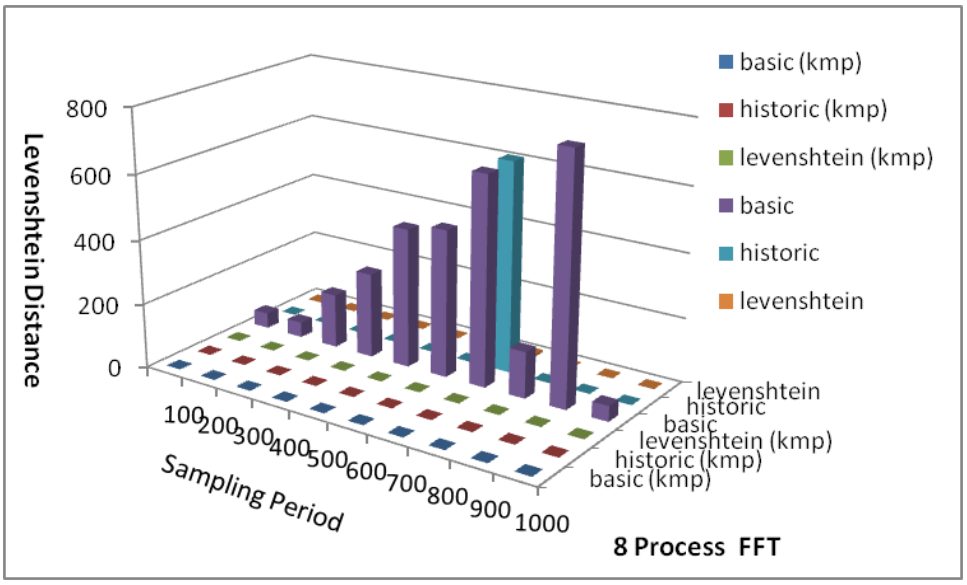

Figure 8.11: Results Using 8 Process Synthetic Data for the FFT type Application

I performed an experiment with small sampling periods using the M/S synthetic data to see how the different predictors would perform. The length of sequence (representing the communications of an entire run of the application), I chose was 100 with 4 processes and 41 was chosen as the checkpoint arrival time. Figure 8.13[3], shows the results obtained by running both my *initial predictors* and my *improved predictors*. We can see that the *improved predictors* still performed well as compared to the *initial predictors*. Also, with small (8, 12, 16) and relatively larger (50) sampling period lengths we can see that my *improved predictors* could not predict because, in the case of extremely small sampling period length (8, 12, etc.), enough information was not there to find fundamental patterns in each sampling period since the patterns had not yet repeated themselves. Similarly, with the sampling period length of 50

[3]Again a continuous data plot is used just to show the variations more clearly. The data is not continuous.

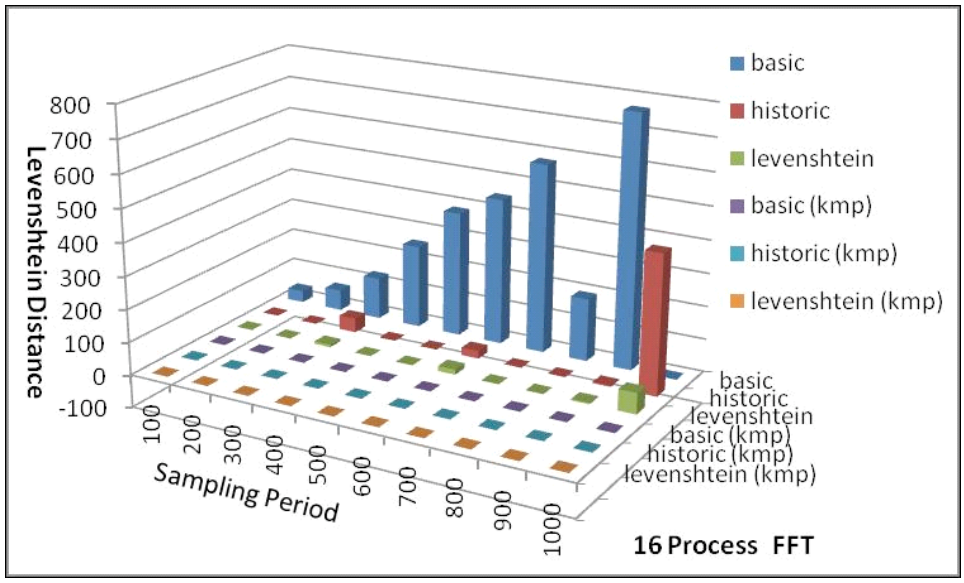

Figure 8.12: Results Using 16 Process Synthetic Data for the FFT type Application

and checkpoint arrival time of 41 and the total length of collected communication information sequence 100, the checkpoint arrived in the first sampling period itself, which was too early for any of my predictors to make any predictions.

The communication characteristics of the application types I used are regular and structured, so I tried an experiment where I varied the large-scale characteristics of the various communication patterns to test how my improved predictors tolerate variation in the characteristics of communication patterns for the application types used in assessment.

I performed an experiment where I modified the synthetic data generation code for the M/S-like application to produce data where the master had a prolonged distribution phase before any slaves could begin their computation/communications, and the results are shown in Figure 8.14[4]. We can see from the results that the *initial pre-*

[4]Continuous data plot is used, just to show the variations more clearly. The data used is not

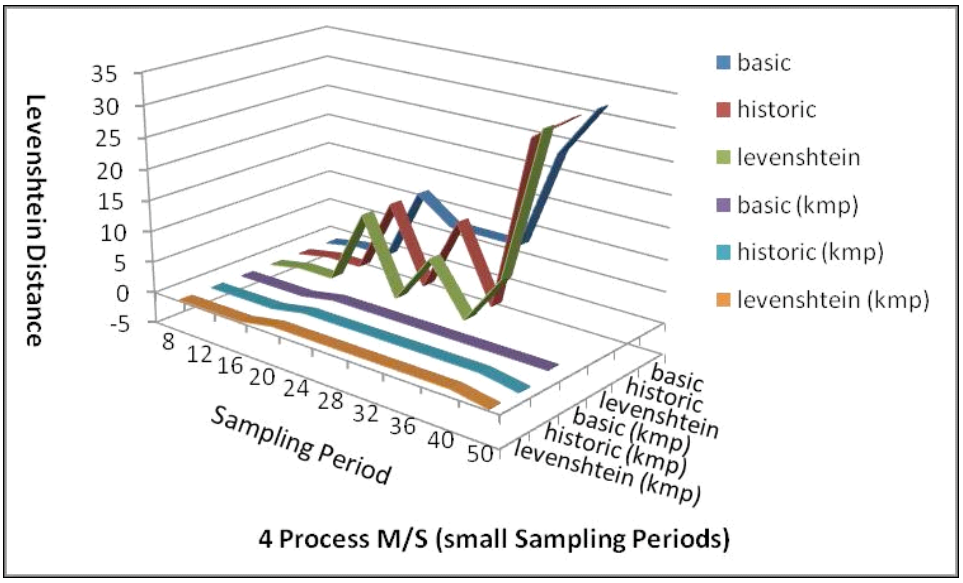

Figure 8.13: Results Using 4 Process Synthetic Data for the M/S type Application with small sampling period

dictors failed, whereas, my *improved predictors* ((kmp) suffixed) predicted accurately. This enforces our belief that there is value in determining fundamental patterns and adapting the length of sampling period and using them for predictions.

8.1.6 Real Data Experiments and Results

After establishing some basic results using the well-behaved synthetic data for the M/S and FFT type applications, I collected real world application communication data to further assess the accuracy of using TEIRESIAS and my predictor algorithms. The synthetic data was hand crafted and represented ideal patterns with regular repetitions. When applications are run in a real execution environment, the communication patterns generated may show less regularity due to the occurrence of communication events in an un-synchronized fashion and/or due to specific appli-

continuous.

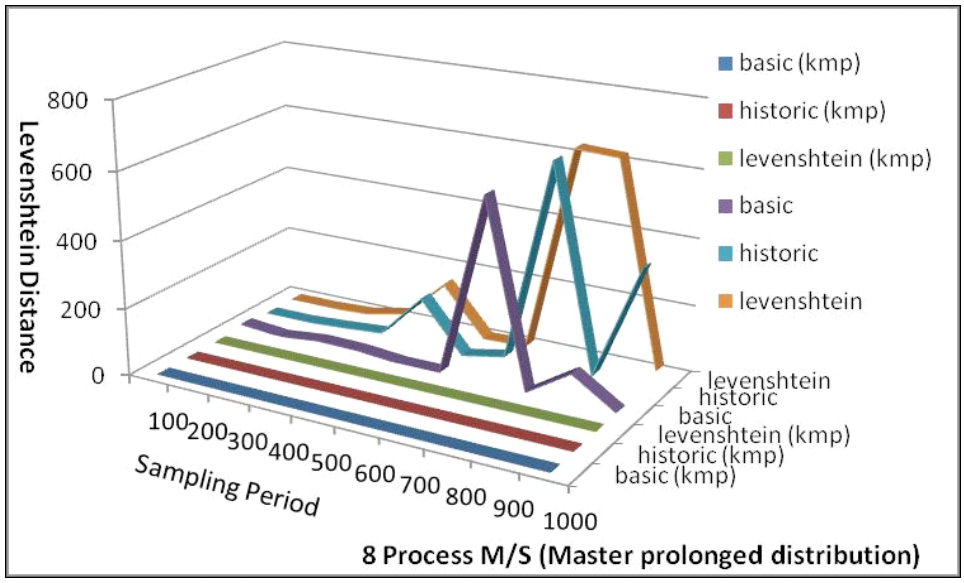

Figure 8.14: Results Using 8 Process Synthetic Data for the M/S type Application with Prolonged Distribution Phase

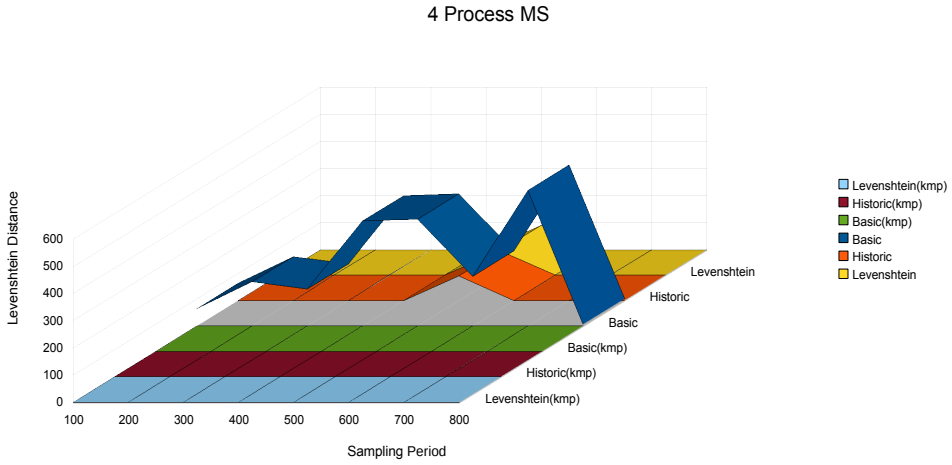

Figure 8.15: Results Using 4 Process Real Data for the M/S type Application

cation characteristics such as data-dependent communication behavior. This makes pattern discovery and prediction more difficult. I chose to use M/S, FFT and FEM (Finite Element Method) applications in my second set of experiments. All these

applications represent common communication patterns, and the first two will permit easy comparison with the first set of experiments (based on synthetic data). I collected the real communication data by running each application using my instrumented version of LAM/MPI with various numbers of MPI processes.

The data collected for a 4 process M/S application amounted to some 2400 communication events. The results, shown in Figure 8.15[5], are for various sampling period sizes and a randomly chosen checkpoint arrival time of 2000. We can see that all the *initial predictors* performed sub-optimally, especially my *basic* predictor which predicted very poorly when compared to my *historic* and *Levenshtein* predictors. All my *improved predictors* were able to predict accurately in this case.

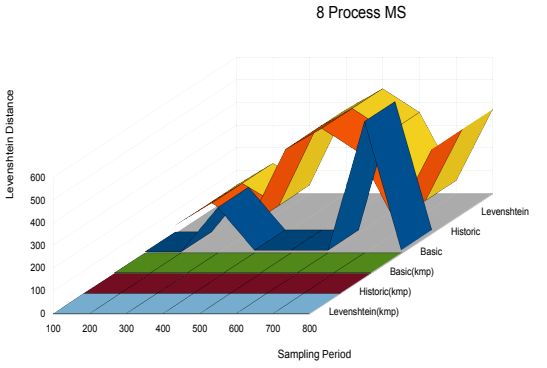

Figure 8.16: Results Using 8 Process Real Data for the M/S type Application

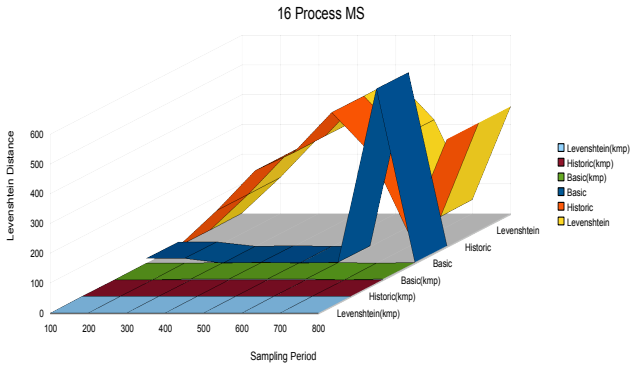

Figure 8.17: Results Using 16 Process Real Data for the M/S type Application

Figure 8.16, shows the results using real communication data captured from an 8 process M/S application. We can see that the *historic* and *Levenshtein* predictors from my initial set of predictors predicted poorly while my *basic* predictor was able to predict accurately in some sampling periods (for sizes 400, 500 and 600). On the other

[5]I again use a continuous graph to show the variations more clearly though the data is not continuous.

hand, all my *improved predictors* (*basic(kmp)*, *historic(kmp)* and *Levenshtein(kmp)*) could predict the upcoming patterns accurately. I got almost the same results with the 16 process M/S application real data, as shown in Figure 8.17. This set of experiments using real M/S data reinforces our results using the synthetic data and provides confidence in the effectiveness of using adaptive sampling periods and recognizing fundamental patterns.

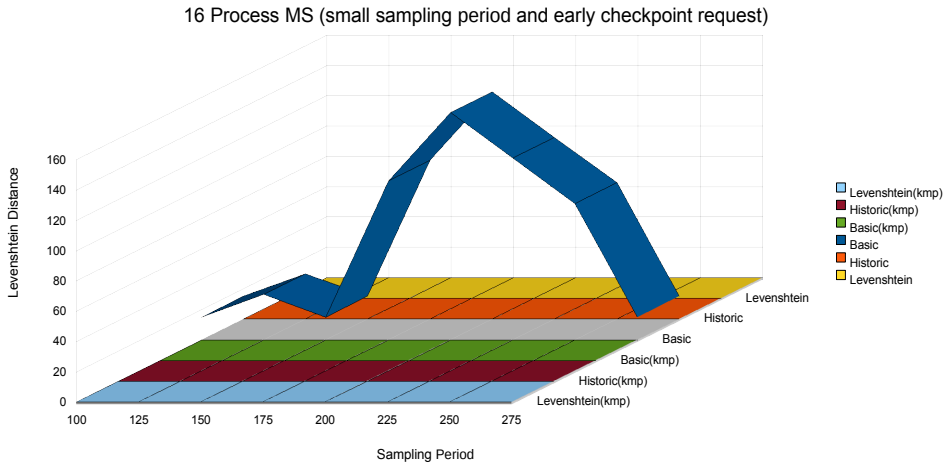

Figure 8.18: Results Using 16 Process Real Data for the M/S type Application with Small Sampling Period

I, next, performed another experiment that consisted of taking the 16 process case and varying the size of the sampling period so that the size of the sub-sequence fed to the prediction algorithm was small but enough for the patterns to have repeated themselves in one sampling period. Of course, when a size of sampling period is chosen it gets adapted by the algorithm by moving back few (as explained in Section 8.1.3) communication events in the sampling period to become an approximate multiple of one cycle of communication events of the application (working on equal sized chunks of data). Recall the example from Section 8.1.3, if one cycle of commu-

nications for a 16 process M/S application for a data size produces 45 communication events in the collected communication information (45 unique symbols), then a subsequence of size 90 would be a multiple of one communication cycle of application and the communications will also have repeated themselves (assuming that the second dataset is of same size as for the first run). Thus, when we choose 100 as the size of sampling period, it gets adapted by moving 10 position (communication events back) to match the exact multiple of one cycle of communication of M/S application to 90. The matching algorithms (TEIRESIAS, KMP) used in my predictions will still be able to see enough information in one sampling period of above sizes to produce reasonable matches because the size of the sampling period (after adaptation) still remains big enough for TEIRESIAS to discover repeated patterns (both small and long A^2, A^3 and so on type), for KMP to match the fundamental patterns in the TEIRESIAS results found in one sampling period, and for the predictors to analyze sufficient previous sampling periods because the checkpoint arrival time is 800. The idea of this experiment is to test the effectiveness of my adaptive and fundamental patterns approach when we have a significantly smaller window for making predictions as compared to the sizes of sampling periods and checkpoint arrival time in the previous experiments presented. Successful results in this experiment would show the effectiveness of my prediction algorithm when working with small sampling period size (which affects how frequently we can collect sampling data).

I chose the size of sampling periods to start from 100 and incremented it in successive runs of the experiment by 25. The checkpoint arrival time chosen was 800 (relatively early when compared to previous experiments). The results shown in Fig-

ure 8.18 suggest that my *improved predictors* can still make accurate predictions when the frequency of data sampling is increased (resulting in small sampling periods) and when a checkpoint request arrives early. As long as enough information is present (some communication patterns, small or long, have repeated themselves) TEIRESIAS can discover those patterns and my predictors can use them to make accurate predictions. This again confirms our confidence in the effectiveness of using adaptive sampling periods and recognizing fundamental patterns using the KMP algorithm.

Next, I collected the communication data of FFT application runs for various numbers of processes for the same problem size. Figure 8.19 shows the predictions for a 4 process FFT application based on real data. In a sequence comprised of 828 communication events, I randomly chose a checkpoint arrival time and made predictions using my *initial predictors* and *improved predictors*. All my *improved predictors*

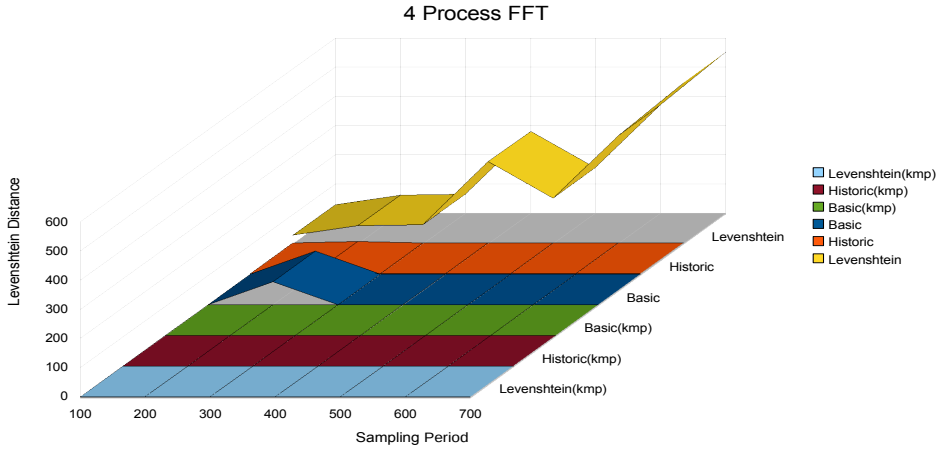

Figure 8.19: Results Using 4 Process Real Data for the FFT type Application

again predicted accurately. The *Levenshtein* predictor from my *initial predictors* performed quite poorly. This was somewhat surprising as it was otherwise the best of

my non-adaptive predictors. I could not find a potential reason for this behaviour. Similarly, the 8 process real FFT predictions, shown in Figure 8.20, suggest that my *improved predictors* can again predict upcoming patterns accurately. Consistent with other experiments, my *initial predictors* performed poorly.

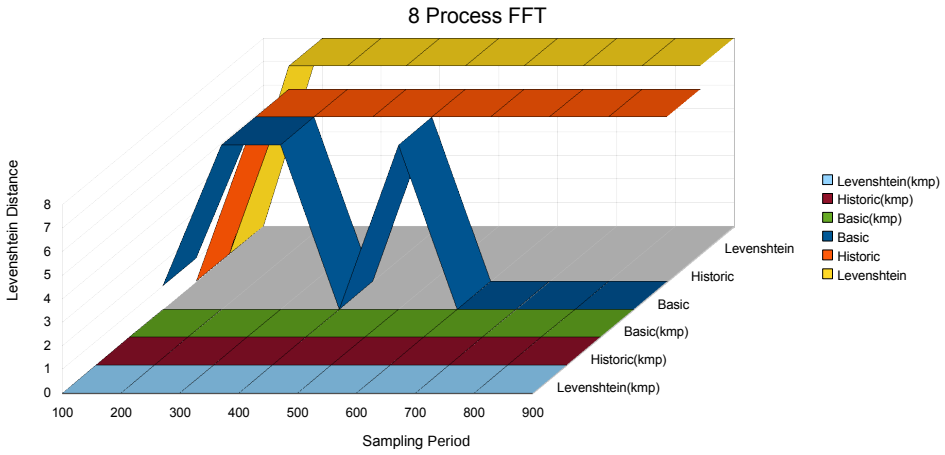

Figure 8.20: Results Using 8 Process Real Data for the FFT type Application

The real world FFT data is more irregular than the FFT-like synthetic data used in my first set of experiments because the data collected from a real run of the application lacks complete synchronization between all pairs of communicating processes. In the synthetic data I hand crafted the communication events knowing who communicates with whom and in what order and after what interval. This made the data highly structured. On the other hand, in a real run the algorithm describes who communicates with whom and after what interval, but the communicating processes reside on different nodes which may lack synchronization in communication leading to an apparent delay in actual completion of a communication event or unexpected orderings between unrelated "parallel" communication events. For example, if the

next communication event is between processes P0 and P1 and the next after that is between P2 and P3, and if one of the processes P0 or P1 is running a little slowly (both are not synchronized) then one of them will have to wait for the other one to reach the point of communication (either to receive acknowledgement or receive data and send acknowledgement) and if processes P2 and P3 are running in parallel with synchronized speeds then the communication between them would complete and get written to the communication data collected prior to the communication event between P0 and P1 resulting in an ordering that differs from the synthetic data.

Such short lack of synchronizations are dealt with by sorting the communication events in small chunks (i.e. small sets of communication events are sorted in increasing order by 1^{st} rank in the communication before the sampling period size sub-sequences are fed to my algorithms). Thus, after sorting, recurring patterns of communication can still be identified. For example, in Figure 8.21 A after sorting communication data with only short lack of synchronization, the pattern "1 2 3 4" is seen recurring in sampling periods a, b, and c.

Any lack of synchronization involving long waits leads to irregular communication events as shown in Figure 8.21 B. In the Figure, short and long lack of synchronization examples are shown followed by sorting. For an application with communication events 1 2 3 4, which repeat after regular intervals. In the case of longer lack of synchronization, the captured communication events might look something like those shown in Figure 8.21 B, left side. After sorting, the patterns no longer repeat, making recognition of the pattern 1 2 3 4 impossible. This is likely the reason why more variation is seen in Figure 8.20 for my *initial predictors* than for the corresponding

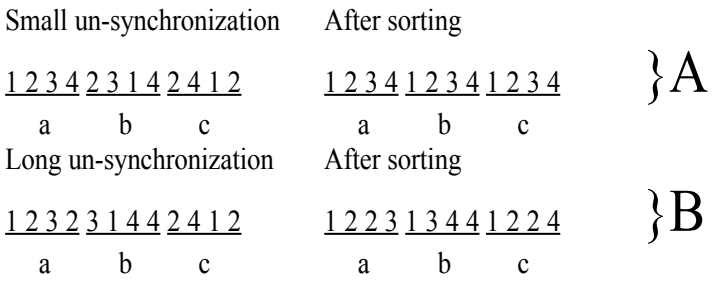

Figure 8.21: Example of lack of synchronization

synthetic data graph in Figure 8.11.

Figure 8.22, shows the prediction results for the 16 process real FFT case. Both my *initial predictors* and my *improved predictors* failed to predict accurately up to a sampling period size of 300 because of insufficient data (i.e. the communication patterns do not repeat themselves till a sampling period size of 300). The sequence for the 16 process FFT test case consisted of 4050 communication events and 1995 was the random checkpoint arrival time for the presented results.

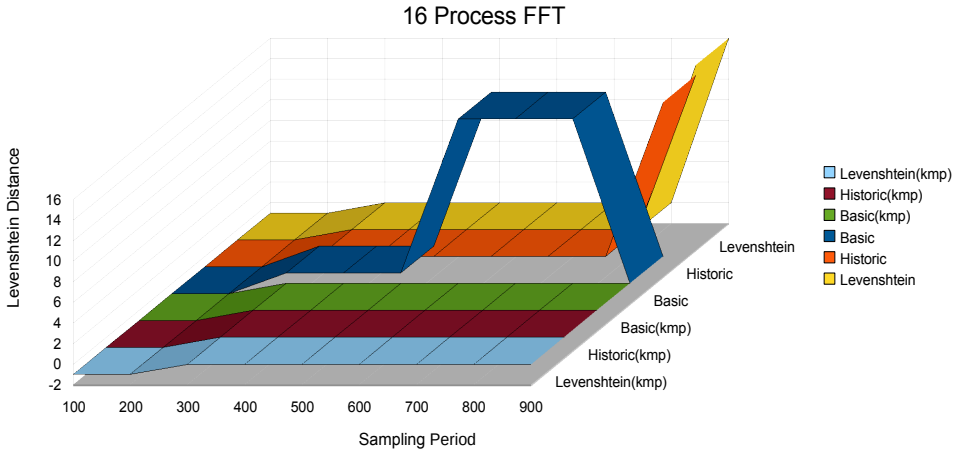

Figure 8.22: Results Using 16 Process Real Data for the FFT type Application

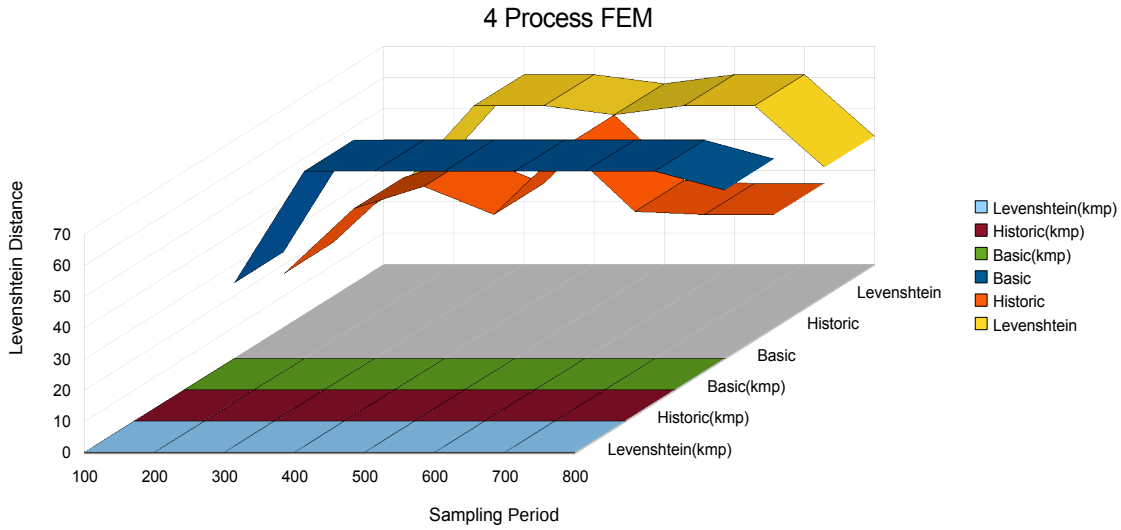

Figure 8.23: Results Using FEM type Communication Data for 4 Processes application

The adaptive approach that I use in the case of FFT and M/S applications does not work well for FEM application. This is because the M/S and FFT applications have long phases of computations followed by brief communications and the communication is well defined (in the case of FFT) or the communication is static (in the case of M/S). Irregularities in the occurrence/order of communication events due to lack of synchronization in my collected communication information for these applications can be easily dealt by my technique of sorting the communication events in small chunks. FEM, on the other hand, also shows regular communication events but the irregularities in the occurrences/order of communication events due to lack of synchronization can be "wide apart" so that sorting in small chunks does not help produce regularity. This kind of behavior in the FEM application is due to the nature of communication and the comparatively small computation times followed by additional communication between processes. When there is long computation time

following the communication (like in FFT and M/S), the irregularity in order of collected communication events due to lack of synchronization can be fixed by sorting in small chunks because we have a large enough sampling window for collecting any delayed communication events preventing the mixing of events from one communication cycle with those from the next. When the computation and communication both are short (like in FEM) then the problem can not be fixed by sorting in small chunks because we have shorter window for collecting any delayed communication events from one cycle of communication before another occurs. Due to small computation

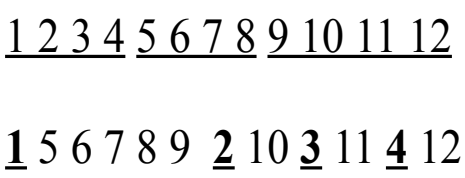

Figure 8.24: FEM and Synchronization

time followed by communication in the FEM application and each group of processes working on a cross-section (for example, $P0$ region, as shown in Figure 8.32) running in parallel it may also happen that a communication between one set of processes may be distributed and recorded in between another group of processes' communication events as shown in Figure 8.24. In Figure 8.24, let 1 2 3 4 is the communication events among a group of processes communicating with each other to exchange data and 5 6 7 8 and 9 10 11 12 are other communication events between other groups of processes. In an ideal scenario, the communication events ordering should be as shown in the top part of Figure 8.24. If, however, there is lack of synchronization between processes involved in the communication events 1 2 3 4, then the communication events might

get recorded as shown in the lower part of Figure 8.24 (where the bold underlined communication events are distributed between the other communication events in the collected communication information).

I modified my adaptivity algorithm to work for FEM in such a way that I run TEIRESIAS and predictors repeatedly by choosing a sampling period size and incrementing it. I choose an arbitrary but reasonable sampling period size first and increase it by small amounts in each size until I find a good prediction. I use Levenshteins distance to measure the accuracy of prediction with the actual observed (upcoming) pattern and also with each size I have seen the predictions, which becomes the basis for whether to adapt or not. I call this kind of adaptation of the sampling period forward-adaptation which differs from the one I used in FFT and M/S application.

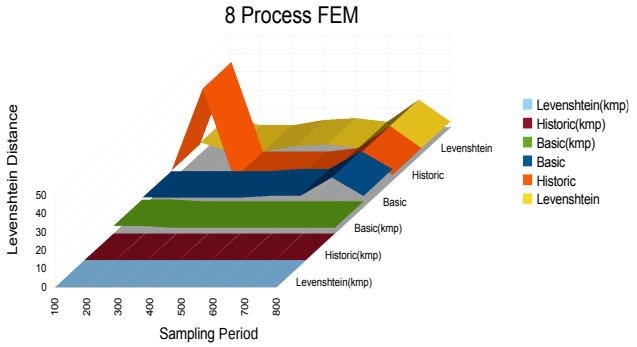

Figure 8.25: Results Using FEM type Communication Data for the 8 Processes Application

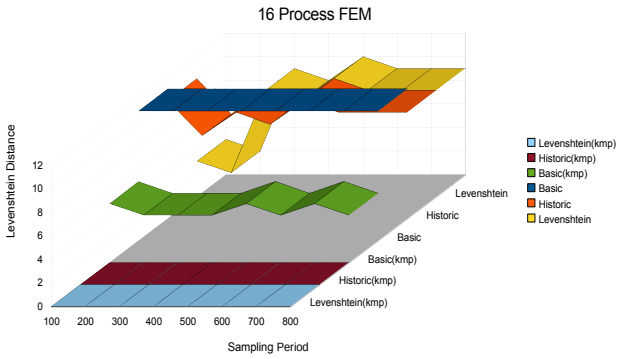

Figure 8.26: Results Using FEM type Communication Data for the 16 Processes Application

Figure 8.23 shows the results for a 4 process FEM run with real data. We can see that the *improved predictors* with forward-adaptation were accurate whereas the *initial predictors* predicted poorly. I do not compare the results of my *improved predictors* with forward-adaptation with my *improved predictors* with moving-back

adaptation because the latter involved sorting the sequences which produced either no predictions or resulted in extremely distorted graphs due to large Levenshtein distances between prediction and observed patterns.

In Figure 8.25 and 8.26, the *basic(kmp) improved predictor* with forward-adaptation fails to predict accurately for 8 and 16 process FEM because when the number of processes increases in FEM we see more irregularities in communication due to lack of synchronization of communication and increase in the number of process groups communicating with each other. This leads my *basic(kmp) improved predictor* with forward-adaptation to predict inaccurately, but my forward-adaptivity technique still helps my other *improved predictors* with forward-adaptation to perform well (in both the 8 and 16 process cases). This is not unexpected because my *basic(kmp) improved predictor* with forward-adaptation performs poorly because the *basic(kmp)* prediction is based only on the communication information that is immediate. The other

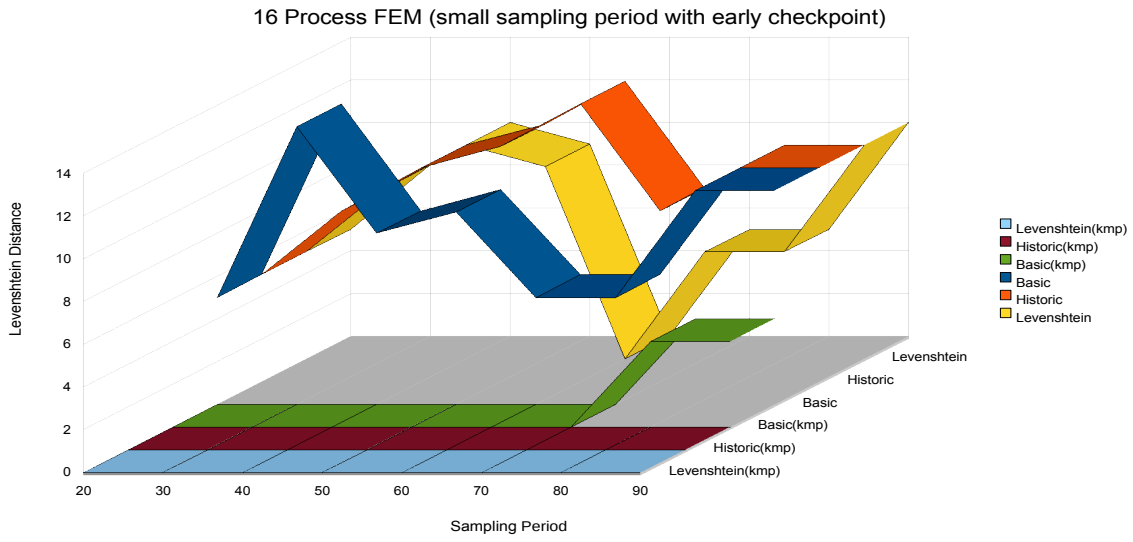

Figure 8.27: FEM Communication Data for 16 Processes application with small sampling period

improved predictors with forward-adaptation, that analyze the historical information still perform better and can be used effectively.

Finally, I performed another experiment for FEM application also, checking how my system performs when the sampling period is small and the checkpoint arrival is early. I expected reasonable results for FEM, in this case, as shown in Figure 8.27, because the communications are more repetitive in small intervals.

8.1.7 Summary of Experiments

The results shown for all the three applications answer the first question that accurate predictions of upcoming communication patterns can generally be made and used to group closely communicating processes. The applications chosen exhibit a fair amount of regularity in communication patterns and based on the results from running my prediction algorithms on the synthetic and real communication information for these kinds of application, I expect my predictors to work well for applications that involve reasonably regular communication during execution. Of course, for applications with random communication (without any regularity), my predictors will likely produce sub-optimal results.

8.2 Simulating Partial Checkpoint/Migrate

To fully assess the usefulness of the partial checkpoint and migrate system I also need to assess the partial migration of processes in different network environments ranging from fast local area networks to much slower wider area networks and under various node loading conditions. Due to the inherent complexity in deployment and

lack of access to widely distributed computing resources, I chose to assess this aspect of the usefulness of my system through simulation.

In a distributed high performance computing environment where the resources are located at geographically distinct locations connected with networks of various bandwidths and latencies, the usability of my system may vary particularly depending on application communication characteristics (e.g. frequency and volume of messages). To study a variety of such network possibilities I first had to select ranges of values for pertinent network characteristics for use in my simulations. Rather than select arbitrary values, I chose to gather actual network bandwidth and latency data for the Local Area Network (LAN) within the Computer Science department, for the University campus LAN (cLAN), between sites at Universities within Canada: the country wide area network (CWAN) and between sites in different countries: the World wide WAN (WWAN). The information was gathered using simple ping tests and FTP transfers and thus provides only a coarse estimate of actual network characteristics but the values obtained do represent a useful range of, for example, possible real world bandwidths. In what follows, the LAN numbers are labelled "CS umanitoba", the campus numbers are identified with "umanitoba" alone and other numbers are labelled descriptively (e.g. "Calgary" represents a site at the University of Calgary). Average results for the Canadian WAN sites are referred to as "CWAN".

Figure 8.28, shows the variation in bandwidth available to different locations. I used FTP connections to download large files from these locations and the values in Figure 8.28 represent simple averages of 10 such FTP downloads from each server providing a coarse estimate of a range of bandwidths. The values are used to study

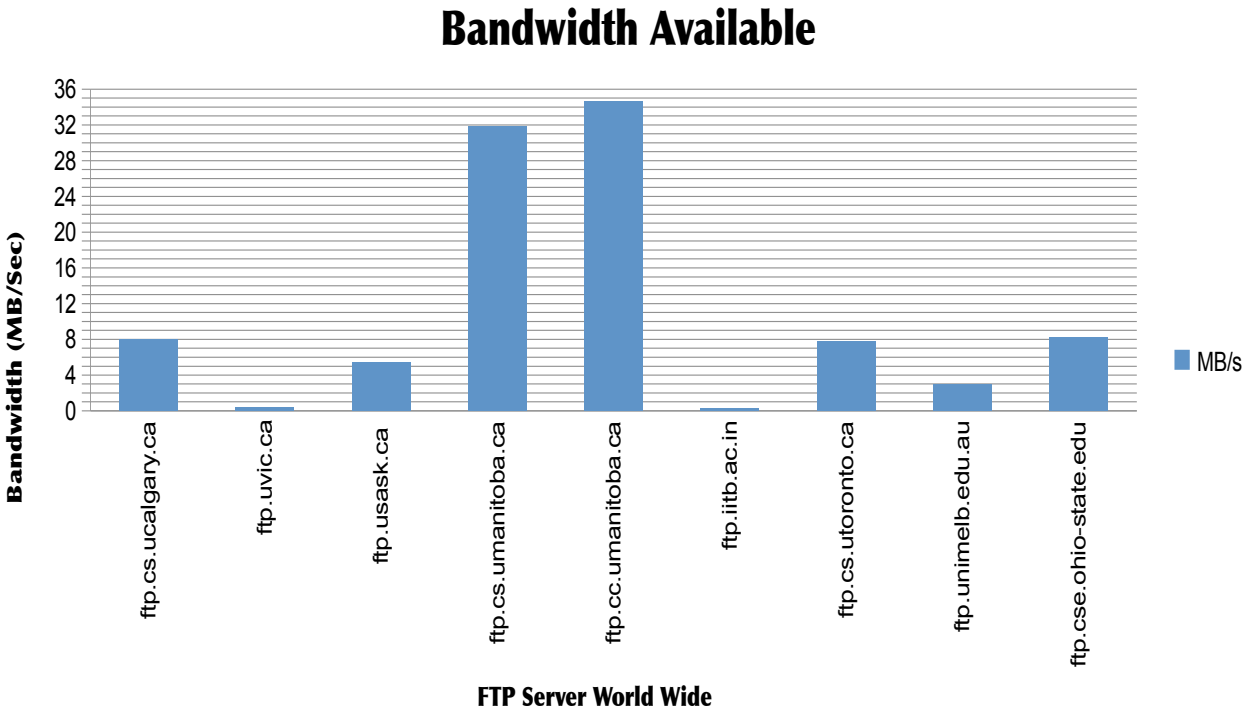

Figure 8.28: Network Bandwidth for various Locations

the impact of variation in network bandwidth on migration of parts of application processes to/from such locations so their precise values are less important than the range of values considered.

For each simulation scenario of interest, I evaluate and compare the performance (i.e. execution time) of an application for three cases: a normal run of the application (with no checkpoint and migration), a run of the application when there is an overloaded node that is making the application run some percentage slower than its normal execution time, and a run of the application when there is an overload and partial checkpoint/migrate is done to move the affected process(es) to various alternate execution locations.

An obvious question is how to accurately model overload in the simulation. The

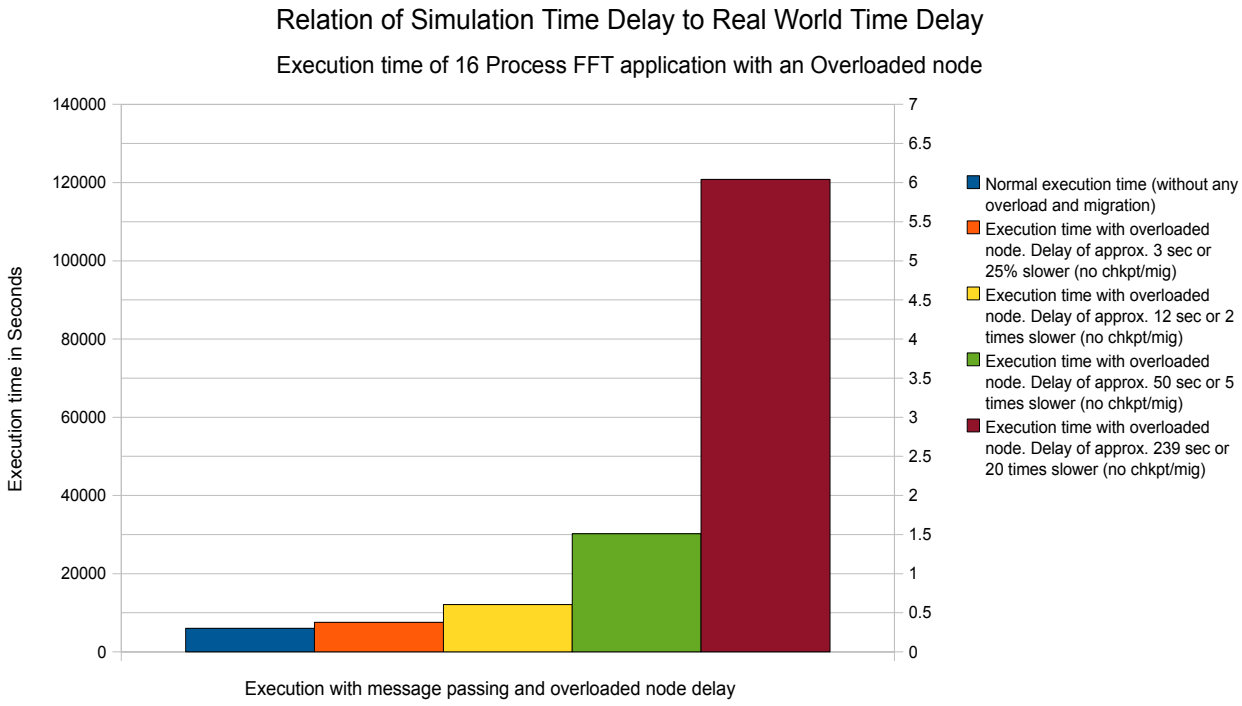

Figure 8.29: Delays due to Overload

communications data that I use in my simulations is data that I get from the actual runs of my test applications. I ran each application using my instrumented LAM/MPI and gathered the communication information including the time-stamps of each communication event. The time-stamp information gives me information about the execution of an application that I can modify to reflect slowdowns due to overloaded compute nodes. To assess the slowdown due to overload I simply run my simulation based on the communication information and time-stamps collected and adjust the timing of communication events based on how much time delay there would be introduced before sending/receiving a message from an overloaded node if it were running a certain percentage, X, slower. The effect of certain percentage slowdowns of one node in a 16 process FFT, is shown in Figure 8.29. As can be inferred from

the graph, increasing slowdown on one node has a non-linear effect on overall execution time as the delays compound across multiple communications between the slow process and others. For all the applications that I use in my simulations, I determine the overload factor to be used in adjusting simulation messaging events by relating them to real world execution times. For example, in the 16 process FFT application (Figure 8.29) if we have a normal execution time without any overload of 6042 seconds then if the application were to run 25% slower than the actual execution time due to an overloaded node, the messages sent to and received from that overloaded node would be delayed by approximately 3 seconds (25% delay is shown in Figure 8.29 as second bar from the left, the first bar being the actual execution time). For my simulations I use 4 overload scales to assess the performance of my system under various conditions corresponding to those shown in Figure 8.29. I take 25% slower as *low* overload or delay, 2 times slower as *medium* overload, 5 times slower as *high* overload and 20 times slower as *extremely high* overload.

The intention here is to compare various applications' execution times (thereby, performance) with and without overload along with the execution time when there is partial checkpoint/migrate involved. Naturally, checkpoint and migrate overhead are included in my simulations using the groupings of communicating processes obtained by running TEIRESIAS and my predictors on the collected communication information for each sampling period. Variation in sampling periods is not considered in my simulations as I have already established that my prediction algorithms make accurate predictions under variation in sampling periods based on my earlier results with small sampling periods and early checkpoint arrival times.

8.2.1 Simulation Design

I used SSJ (Stochastic Simulation in Java) [LB05] to develop my discrete event based simulator. My simulator uses inter-process communication information captured from application runs to drive the simulation. The actual computations done do not affect the simulations (which are focussed on inter-process communications) and are therefore abstracted away in the simulation. I begin each simulation by creating a simulated runtime environment where compute resources are distributed and connected with each other over networks having various bandwidths (based on the data that I obtained from my bandwidth collection exercise). I then "start" the execution of a given application with a number of processes each residing on a separate node (for example a 16 process FFT on 16 nodes), by choosing nodes that have a LAN connection with each other so that the processes are located near to each other (in terms of network proximity). I then read the communication and time-stamp information that I created during my instrumented sample run of the application using LAM/MPI and begin the simulation.

As described earlier, each pair of processes is mapped to a unique symbol that corresponds to a communication event between those processes. I schedule simulated communication events between processes based on the captured communication symbols at the corresponding time (possibly adjusted to reflect slowdown) based on the time-stamp information. The first time-stamp is translated to simulation time 0 (start of the application) and the time-stamps following the first one are then adjusted on-the-fly relative to 0.

To simulate having a process running on an overloaded node I choose a random

process from those associated with the simulated application. Any messages sent to and received from that process are then delayed by using a selected overload factor as described.

The core events in my simulation are *overload* and *checkpoint/migrate*. For simulating partial checkpoint/migrate I generate random numbers between 0 and a "frequency" of checkpoint and migrate and when the number generated equals 0, I treat it as a checkpoint request and trigger an associated overload event to determine which process(es) will need to be checkpointed and migrated. The frequency of checkpoint and migration can be varied to induce low, medium or high frequencies of simulated overload and hence partial checkpoint/migrate (described in detail later). The simulator then checks the process(es) on the overloaded node and determines the group of communicating processes (using the grouping information derived using TEIRESIAS and my *improved predictors* (*historic* and *Levenshtein*)) for when the checkpoint request has arrived (by running the prediction algorithm). The group of communicating processes is chosen and the process(es) on the overloaded node are checked to see if they occur in the predicted group. If so, a the checkpoint/migrate event is triggered for migrating the entire group of communicating processes together otherwise just the process(es) on the overloaded node are checkpointed and migrated and the simulation continues and waits for the next event.

In the event of checkpoint/migrate, the cost (time required to transfer data at the available bandwidth) of migration to migration locations for the migrated processes is computed. Once the processes are migrated to new locations I also factor in the subsequent delay of communication between the migrated and non-migrated processes

when computing the final execution time of application with checkpoint/migration.

For my simulator to give useful results, the data that I use to simulate application behavior must be accurate. There are two possible concerns related to this: the use of data collected from a pseudo-parallel (one compute node) execution of the algorithm and the use of small application runs to simulate larger ones. These issues are now discussed before reporting my simulation results.

8.2.2 Correctness of the Communication Data: Parallel vs. Pseudo-Parallel Execution

An obvious question that arises is the correctness of the time-stamp information collected because of the fact that I am running applications on a single processor rather than in an actual parallel execution environment where each process would be running on an independent processor. In a single processor environment when there is more than one process the processes are not running in parallel, but in an alternating sequential ("pseudo-parallel") manner. Each process gets a share of processor time for execution before giving up the processor to another process. Even though I have collected the time-stamp information for the communication events by running the applications in a single processor environment, I argue the information collected is still useful and that the simulation results that I present based on the data I have collected are still valid. This is because we are not interested in the exact time at which communication events occur but rather, primarily, in the order they occur relative to one another.

A parallel application is run on multiple nodes/cores to exploit parallelism in com-

putation and communication. An example of the computation and communication events that might take place in such a parallel application are depicted in Figure 8.30 (left side) and a similar pseudo-parallel execution is also depicted (right side). In

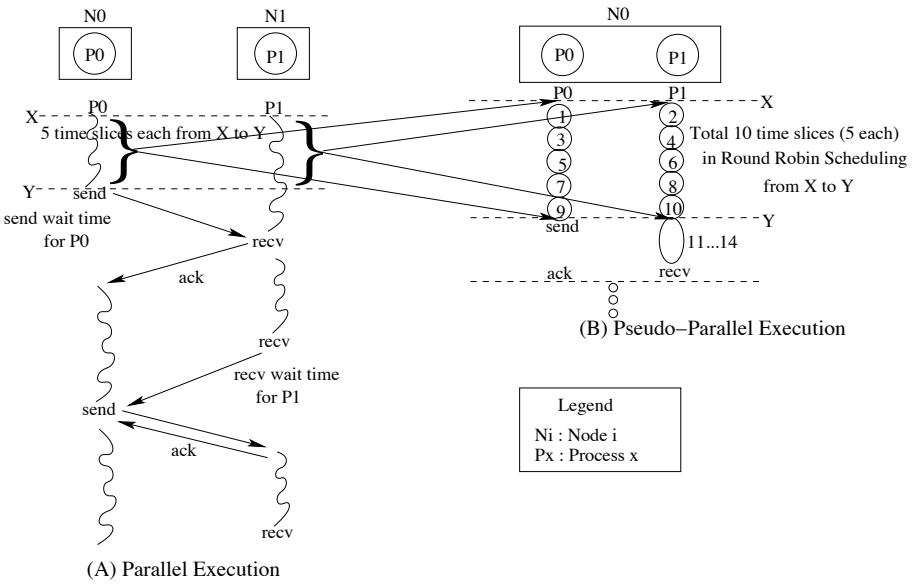

Figure 8.30: Parallel vs. Pseudo-parallel

the Figure, $P0$ and $P1$ are processes on execution nodes $N0$ and $N1$ in the parallel execution and together on execution node $N0$ in the pseudo-parallel execution. X and Y are the start and end times of a period of computation before a message is sent/received by the processes. Assume that the processes ($P0$ and $P1$) take 5 time-slices of CPU time to execute from time X to time Y. The numbers in circles in the pseudo-parallel execution (right side) depict the order in which time-slices might be given to processes $P0$ and $P1$ (in an alternating fashion). I will refer to this example when explaining the validity of my collected communication data.

A typical parallel MPI application consists of several processes each identified by a rank of its own which, typically, execute the same piece of code on different data

to solve a single parallel problem (i.e. SPMD model). If such an MPI application with, for example, 4 processes (ranks) is running on a single node/core then there is no parallelism and all the processes/ranks share the same processor for execution. Ideally, in a parallel environment, each process should be running independently on its own separate processor giving the potential for, in this example, four times speedup.

In a pseudo-parallel execution, multiple processes will compete for the CPU, and the operating system uses some scheduling algorithm (such as round robin) to allot the CPU to the processes in a fair way ensuring that they each get an approximately equal share of the CPU over relatively short time periods. In a parallel MPI application the processes are supposed to run in parallel when they are computing and only wait when they send/receive messages to/from other processes. If an application is running on a single node/core, each process also has to wait for its turn to proceed in computing. The key question is, what affect does waiting for the CPU have on the apparent parallelism of the application (as judged by the communication events) and whether I have to adjust the time-stamp for each communication event I collect to compensate.

There are two things that need to be demonstrated to show that the data that I collect by running the application on a single node/core (the pseudo-parallel case) is, in some sense, correct with respect to what it would have been if the application was run on more than 1 node/core (the parallel case). First, I need to show that the computation times of the processes in the pseudo-parallel case are "in equilibrium" with the parallel case. By "in equilibrium" I mean that the time taken by all the processes in the parallel execution to reach from point X to Y in the computation is

only a fixed factor different from the time taken by all processes in the pseudo-parallel execution to reach from point X to Y in the computation, given that the problem size is the same in both executions. That is, the duration of the computation times between communication events is in a fixed ratio between the parallel and pseudo-parallel cases. Second, I need to show that the overall speed ratio between the parallel and pseudo-parallel cases is consistent. That is, the speed ratio between the point a message is sent and the point that message is received in the parallel and pseudo-parallel cases are also "in equilibrium" (in the same sense as for computation times).

Computation Times are in Equilibrium

Consider an example MPI application which has only 2 processes, $P0$ and $P1$, run in parallel and pseudo-parallel as shown in Figure 8.30 (A) and (B).

In Figure 8.30 (A), $P0$ and $P1$ each take 5 time slices to execute from point X to point Y in their respective executions (computation only). Note that I collect time-stamp information for the communication events only and not the computation events. After reaching point Y, process $P0$ sends a message and waits to receive an acknowledgement from the recipient process $P1$ which still needs to compute (say 4 time slices more). Before it can receive any message, assuming blocking sends and receives (as are used in all my sample applications), process $P0$ has to wait for process $P1$ to receive the message and send an acknowledgement before it can proceed with its computation. This waiting occurs whether $P0$ and $P1$ are running on different nodes/cores (in parallel) or not. So the time-stamp of a communication event ($P0$ to $P1$) in Figure 8.30 (A) reflects the time when both processes have spent their 5 time

slices and reached point Y in the execution. Only then does $P0$ send a message to $P1$ and the time-stamp is recorded. Similarly, in the pseudo-parallel case, process $P0$ will not reach the point Y faster than process $P1$, because equal shares of the CPU are given to both processes and they normally take turns in proceeding in computation by gaining access to CPU. Thus, the order of communication events in both cases (parallel and pseudo-parallel) would still be the same and only the time-stamps in the pseudo-parallel case would differ by a fixed factor due to the wait time of processes for getting the next time slices. In Figure 8.30 (B), the pseudo-parallel case, it is just as if the processes are running slower. Because in Figure 8.30 (B) each process gets a roughly equal share of CPU for computation, when processes $P0$ and $P1$ are executing on node/core $N0$ we can think of the CPU as being half as fast because it is being given alternately to processes $P0$ and $P1$, as reflected in Figure 8.30(B). Recall that the circled numbers in Figure 8.30(B) represent the time slices. Each process needs 5 time slices and $P0$ is given slices 1, 3, 5, 7, 9 while $P1$ is given slices 2, 4, 6, 8, 10 Both $P0$ and $P1$ execute from point X to Y in lock step and arrive, approximately, together. Then $P0$ sends the message to process $P1$ (time-stamp recorded) and waits for an acknowledgement from $P1$. Since $P0$ is waiting, $P1$ gets all the CPU time and reaches the point where it receives the message at full speed (no CPU sharing) and sends an acknowledgement. Then they both proceed with computation again. As in the parallel execution case, both processes make progress then wait for each other when communicating. So, the time-stamp information that I collect in Figure 8.30 (B) is valid and useful. The compute times for the parallel and pseudo-parallel cases differ by a fixed amount and the processes execute from

point X to point Y in approximately the same time in the pseudo-parallel case as they would in a parallel execution running on a slower processor.

This argument generalizes to arbitrary numbers of processes. In the example shown in Figure 8.30 (A) and (B), where there are two processes, if the parallel processes take 5 time slices each to execute from point X to point Y in the computation then it will take 10 time slices in the pseudo-parallel case. If we increase the number of processes to four in both the parallel and pseudo-parallel cases, but keep the problem size the same, then because the problem size is the same, the data size distributed between two processes earlier will now be distributed among four processes. This smaller data size to each process will, in general, reduce the execution time of all four parallel processes to 2.5 time slices for each process to execute from point X to point Y. The same problem, when run in pseudo-parallel with 4 processes, will still compute in a total of 10 time slices as each of the 4 processes will now need only 2.5 time slices to execute from point X to point Y. In this case the pseudo-parallel execution is like running on parallel processors that are four times slower.

Another question arises if there are more processes since the wait time to get a time slice again by a process increases with the number of processes. I argue that the time-stamp information is still valid in this case by showing that the wait time of processes to get another time slice is small relative to the overall application execution time. With an increasing number of processes the correctness could depend on 1) the problem size, 2) the application communication characteristics, and 3) the duration of a time slice. Consider a fixed problem size. The communication characteristics of the application (*total number of communication events / total execution time*) will

determine the average time pairs of processes are involved in computation before communicating with each other This average time of computation will determine how to distribute the total number of time slices among all processes for computation between communication events. With this information, for a given number of processes I can then determine the number of time slices a process (rank) should get between each communication event and the wait time for one process to get another time slice. If I can show that the average wait time for a process to get another time slice is small relative to the application execution time, then I can show that the processes do not have to wait for long time before they can communicate with each other. So the collected time stamp information is still useful. To be fair to all the processes some variant of Round Robin scheduling is commonly used by Linux and the maximum time slice that a process can have is commonly hard coded to be approximately 5-10 milliseconds.

Given this information I can determine the average percentage of time that a process (rank) has to wait to re-gain access to the CPU in each case. As long as the wait time is not large (e.g. a small percentage of the execution time) I can confirm that increasing the number of processes (ranks) will not cause the execution time of the parallel and pseudo-parallel cases to be out of equilibrium.

I now present this analysis for some example execution environment sizes for the applications I used in my experiments.

1. Fast Fourier Transformation (FFT) Application

   The image size I used is 4K × 4K data points for my sample FFT application.

   - **4 Process FFT analysis:** Problem size = 4K × 4K

Run time of application (I repeated the entire application to solve the problem 46 times to collect enough[6] data) = 16390 seconds.

The total number of communication events = 828.

This gives, 16390/828 = 19 seconds between each communication event.

If we have a time slice of 10 milliseconds which is typical of the Round Robin based scheduling algorithm in Linux, then there are 19 × 1000/10 = 1900 time slices between communication events divided between the 4 MPI ranks being executed.

This gives 1900/4 = 475 time slices/rank (or 4750 milliseconds/rank) between communication events.

The average time a process/rank has to wait to get another time slice (using a 10 millisecond time slice) is 10 × 3 = 30 milliseconds.

This is 30×100/16390×1000 = 0.00018 percent of the total runtime, which is very small. Therefore the use of the data collected using pseudo-parallel execution should be safe.

- **16 Process FFT:** Problem size = 4K × 4K data points

  Run time of application (I repeated the entire application to solve the problem 15 times to collect enough data) = 6042 seconds.

  The total number of communication events = 4050.

  This gives, 6042/4050 = 1.5 seconds between each communication event.

  If we again have a time slice of 10 milliseconds, then there are 1.5×1000/10 = 150 time slices between communication events divided between the 16

[6] "enough" is determined by how long an application runs so that we have gathered sufficient communication information for TEIRESIAS and my predictors to analyze communication patterns in reasonable sampling periods.

MPI ranks being executed.

This gives 150/16 = 9 time slices/rank (or 90 milliseconds/rank) between communication events.

The average time a process/rank has to wait to get another time slice (using a 10 millisecond time slice) is $10 \times 15 = 150$ milliseconds.

This is $150 \times 100/6042 \times 1000 = 0.0024$ percent of the total runtime, which, again, is very small. Therefore the use of the data collected using pseudo-parallel execution should again be safe.

2. Finite Element Method (FEM) Application

The problem size I used is 1K × 1K data points for my sample FEM application.

- **4 Process FEM:** Problem size = 1K × 1K

  Run time of application = 34.14 seconds.

  The total number of communication events = 1986.

  This gives, 34.14/1986 = 0.0171 seconds between each communication event.

  Given a time slice of 10 milliseconds, there are $0.0171 \times 1000/10 = 1.71$ time slices between communication events divided between the 4 MPI ranks being executed.

  This gives 1.71/4 = 0.4275 time slices/rank (or 4.275 milliseconds/rank) between communication events.

  The average time a process/rank has to wait to get another time slice (using a 10 millisecond time slice) is $4.275 \times 3 = 12.825$ milliseconds.

  This is $12.825 \times 100/34.14 \times 1000 = 0.03$ percent of the total runtime, which

is still very small. So the use of the data collected using pseudo-parallel execution should be safe.

- **16 Process FEM:** Problem size = 1K × 1K data points

  Run time of application = 36.18 seconds.

  The total number of communication events = 14135.

  This gives 36.18/14135 = 0.0025 seconds between each communication event.

  Given a time slice of 10 milliseconds, there are 0.0025 × 1000/10 = 0.25 time slices between communication events divided between the 16 MPI ranks being executed.

  This gives 0.25/16 = 0.015625 time slices/rank (or 0.15625 milliseconds/rank) between communication events.

  The average time a process/rank has to wait to get another time slice (using a 10 millisecond time slice) is 0.15625 × 15 = 2.34375 milliseconds.

  This is 2.34375 × 100/36.18 × 1000 = 0.0064 percent of the total runtime, which is very small. So the use of the data collected using pseudo-parallel execution should be safe.

3. Master Slave (M/S) Application

   The problem size I used is 0.02K × 0.02K matrix size for my sample MS application.

   - **16 Process MS:** Problem size = 0.02K × 0.02K

     Run time of application (ran for 20 time to collect enough data) = 17.38 seconds.

The total number of communication events = 11988.

This gives 17.38/11988 = 0.00144 seconds between each communication event.

Using a time slice of 10 milliseconds, there are $0.00144 \times 1000/10 = 0.14$ time slices between communication events divided between the 16 MPI ranks being executed.

This gives 0.14/16 = 0.00875 time slices/rank (or 0.0875 milliseconds/rank) between communication events.

The average time a process/rank has to wait to get another time slice (using a 10 millisecond time slice) is $0.0875 \times 15 = 1.3125$ milliseconds.

This is $1.3125 \times 100/17.38 \times 1000 = 0.0075$ percent of the total runtime, which is very small. Therefore the use of the data collected using pseudo-parallel execution should be safe.

Speed Ratio between Send and Receive

I have shown that the computation run times for the parallel and pseudo-parallel cases are in "equilibrium" and that the wait time of processes to get another time slice in the pseudo-parallel case is small relative to the overall execution time of the application. This means that the time-stamp when a message is sent after some computation is done is in a consistent ratio between the parallel and pseudo-parallel cases. I now show that the speed ratio between the parallel and pseudo-parallel cases is, in some sense, in "equilibrium" as well and that increasing the number of processes (ranks) for a truly parallel execution on many processors does not invalidate the use

of my collected data in my simulations.

Consider the communication data collected from my sample runs of applications using pseudo-parallel executions and assuming that the data collected is from regular SPMD parallel applications. This means that each process (rank) works on the same algorithm and a fairly equal amount of data in any given compute period between consecutive communication events. This means that the processes (ranks) work roughly synchronously with each other so if there is a delay in receiving a message then that delay is due to a lack of synchronization between processes (which should be small because the code is regular and SPMD) rather than due to unexpected communication delay.

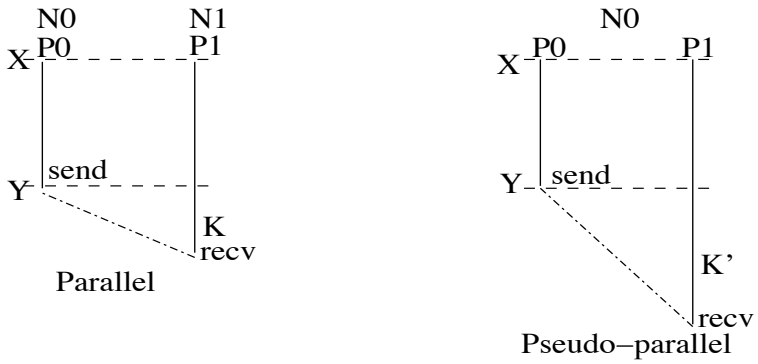

Figure 8.31: Speed Ratio between Parallel and Pseudo-parallel cases

Consider the scenario shown in Figure 8.31. Let N be the total number of ranks in an application run. In the parallel case (left side of the Figure) the application is running on two nodes/cores $N0$ and $N1$ with one process (rank) on each node/core, say $P0$ on $N0$ and $P1$ on $N1$. When $P0$ reaches the point Y in its execution and sends a message to $P1$ it waits for acknowledgement from $P1$. If $P1$ gets delayed due to lack of synchronization between the processes, it takes K time units to reach the

point where it receives the message and sends back an acknowledgement. Similarly, on the right side of Figure 8.31, where processes $P0$ and $P1$ are running on the same node $N0$ in pseudo-parallel fashion, when $P0$ reaches the point Y and sends a message to $P1$ it waits for an acknowledgement from $P1$. $P1$ might be delayed in receiving the message and sending back an acknowledgement by K' time units due to the wait time to get another time-slice as well as any lack in synchronization. Then the question of concern is how much time the parallel case will take as we increase the number of processes (ranks) (N) relative to the pseudo-parallel case. I wish to show that for reasonable N the difference in apparent message delivery time ($K' - K$) is relatively small.

In the pseudo-parallel case, processes/ranks share the CPU approximately equally. Thus, after a message is sent, the receiving process may have to wait while other processes/ranks execute their timeslices before receiving the timeslice it needs to actually execute its receive operation. If, as in Figure 8.31 (left), we expect a delay between send and receive of K time units due to lack of synchronization between sender and receiver, then, in the worst case, executing in a pseudo- parallel fashion, we would expect $(N-1) \times K$ time delay due to the need for the still executing $N-1$ processes/ranks to share the CPU (i.e. all but the sending process who is blocked waiting for an acknowledgment). Thus, as the number of processes increases so too does the delay between the send and receive events in a message exchange. Since we assume regular, SPMD applications, however, we expect K to be small and the number of timeslices required by each rank/process to complete its execution before the receiving process can execute its receive operation to also be small (perhaps 1)

so even if we have a relatively large N (e.g. hundreds of processes) the time delay between send and receive will be quite small in comparison to the actual message delay in the network[7]. Thus, the message event times collected are reasonably indicative of the times we could expect in a true parallel execution and can be simply scaled to reflect slower communications links as required. Since both the computation and communication times are reasonable, it is safe to use the collected communications data in my simulations.

Further, consider the following:

- N=2: If $N = 2$ we have K''=$N \times K = 2 \times 5 = 10$ in the parallel case and K'=$(N - 1) \times K = 1 \times 5 = 5$ in the pseudo-parallel case. The ratio between these two numbers (that is the amount of difference in time to receive a message and send an acknowledgement between the parallel and pseudo-parallel cases) is $10/5 = 2$ time units.

- N=1000: if $N = 1000$ we have $K'' = N \times K$=$1000 \times 5 = 5000$ in the parallel case and $K' = (N - 1) \times K = 999 \times 5 = 4995$ in the pseudo-parallel case. Now the ratio between the two is $5000/4995 = 1.001$ (instead of 2) time units. Hence for larger values of N, given a fixed, small value of K the difference in time to receive message and send an acknowledgement between the parallel and pseudo-parallel cases is much smaller.

Therefore using the time-stamps collected from the pseudo-parallel case in the simulation is sufficiently accurate compared to using time-stamps collected from parallel

[7]When I did my pseudo- parallel test runs, no MPI messaging optimization was used so the time to deliver a message from one process/rank to another was determined by the network access time – essentially a LAN network delay, rather than the much smaller time that would have been incurred if in-memory message delivery had been enabled.

runs as N gets larger.

8.2.3 Correctness of the Communication Data: Extrapolation of Problem Size

As it was not possible for me to increase the problem sizes (number of processes) in my test runs due to the implementation environment used (as described earlier), I had to use extrapolation from the communication data obtained for smaller problem sizes to represent communication data for larger problem sizes. The correctness of this extrapolated data for each application type must be addressed.

Master Slave (M/S) Data

The extrapolation of M/S data is the trivial case. An M/S application has three phases, distribution, computation and collection. If I have communication data from a 16 process runtime environment I can extrapolate this data to represent, say, a 4096 processes runtime environment for solving a bigger instance of the problem simply by duplicating the communication data from the small case (16 processes) to represent the data in the large case, renumbering communicating processes as needed. (e.g. a message from $P0$, the master, to $P1$ might be renumbered as a message from $P0$ to $P16$, $P32$, and so on in the duplicated data sets). This comes out to be a total of (4096/16=) 256 such smaller execution environments in the example. This duplication of data is safe to use because the communication exhibited in the extrapolated data is structurally the same as in a real (larger) application run.

Finite Element Methods (FEM) Data

Suppose that I have an $N \times N$ matrix of data points each allocated to a separate process with a total of P processes. In FEM the communication between processes is data induced. To compute a new value for a point, the process computing the point value needs data from its corresponding neighbor processes. In a 2×2 problem (as shown in Figure 8.32, left side) with 4 points, one point per separate process, P_0, P_1, P_2, and P_3, P_0 and P_1 exchange values, P_1 and P_3 exchange values, P_0 and P_2 exchange values and P_2 and P_3 exchange values. The number of communications

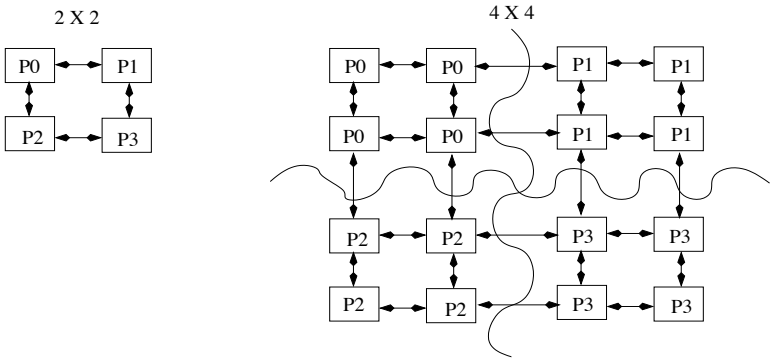

Figure 8.32: Communication in FEM and increase of Problem Size

between processes is N^2. If we increase the problem size simply by duplicating and updating the communication data from the 2×2 case to represent the data in the 4×4 case on the same P processes I lose the inter-process communications at the boundaries between the replicated 2×2 sub-problems (shown as crossing the wavy lines in Figure 8.32, right side). The loss in communication events is $2N$ which is asymptotically small compared to the total N^2 actual communication events, which suggests that the projected data can be used for larger size problems with only limited

loss of accuracy. As such I did not attempt to modify the extrapolated data to try to reflect the additional communication events.

Fast Fourier Transformation (FFT) Data

The 2-d FFT application that I use in my testing distributes the data, row wise to the available processes, a 1-d FFT is then performed by each process on its local data. Then the data is exchanged between processes to transpose the rows to columns followed by a 1-d FFT performed on the columns by each process. The results are then collected. Thus, the communication is involved only in distribution of data to processes, transpose of rows to columns and collection or results between processes.

Based on how the FFT algorithm I use was designed, no matter what the size of the problem is, as long as the number of processes used is the same (which is what I did) the patterns of communications are still the same as the problem size changes (no loss of communication). The reason there is no loss of communication with the change in problem size is because, the communication is algorithm induced and so what ever the size of problem given to each process is, the pattern of communication is still going to be the same as long as the number of processes used is the same. The extrapolation of data from a small size problem to represent a large size problem communication data is done by running a certain size problem for a given number of processes on my instrumented LAM/MPI and the communication information is collected. Then the collected communication information is duplicated to represent a bigger problem size to be run on the same number of processes in the simulated execution environment . Thus, my simulations address scaling in the size of problem,

not in the environment size.

To argue the correctness of my extrapolated data when the size of the execution environment size increases, consider for example, the FFT communication data for a problem size of 256 points in a 16 process runtime environment. If I want to simulate a bigger size problem say 512 point FFT on 32 processes, I can divide the 512 points problem into two parts of 256 points each and run each on (32/2=) 16 processes. In this case there is no communication lost when the program is working on the 256 points problem size using my extrapolated data. The only communication data lost is when the results from both the execution environments are combined to get the final result. Similarly, for an execution environment of 4096 processes, I can solve a problem size of 65536 (65536/256 = 256 points) points FFT by extrapolating the 16 process 256 point FFT communication data by simply duplicating it 256 times to represent data collected from 256 sub-execution environments each of size 16 processes solving 256 points FFT. The only communication data lost is the collection of data from all the 256 sub-execution environments when computing the final result.

I certainly, lose communication at the end of the execution when combining the results from all the 256 (16 process) execution environments but, generally, checkpoint/migrate is not desirable during the end of an algorithm execution because the bulk of the execution has already been done so there is little potential benefit (i.e. it is probably more effective to complete the execution despite one or more slow running nodes than to pay the cost of partial checkpoint and migrate followed by nothing more than a few communications to gather final results).

Based on how my 2-d FFT application works, MPI_scatter() takes O(n) time (n

is number of processes) + row FFT takes O(N log N) time (N is the problem size) + transpose with MPI_alltoall() takes O(n^2) time (n is number of processes)+ column FFT takes O(N log N) times (N is the problem size) + MPI_gather() (batch collection of results) O(n) (n is number of processes).

Thus, for a 16 process environment and 16 point FFT (ignoring the constant factors for illustration purposes) the approximate time would be:

$16 + 16 \log 16 + 16^2 + 16 \log 16 + 16$

$16 + 16 \times 4 + 256 + 16 \times 4 + 16$

$16 + 64 + 256 + 64 + 16 = 416$ is the total execution time

Losing O(n) gather events mean 16 events, and gathering 16 events takes

$(16 \times 100)/416 = 3.8\%$ of total execution time (which is lost and is small compared to the total execution time of 416).

For a 4096 processes environment of 4096/16=256 (16 process environments) and 1,000,000 point FFT we have 1000,000/256=3906 point FFT on 256 execution nodes. Again, ignoring constant factors for illustrative purposes:

$256 + 3906 \log 3906 + 256^2 + 3906 \log 3906 + 256$

$256 + 46604 + 65536 + 46604 + 256 = 159256$

Losing O(n) gather events mean 256 events, and gathering 256 events take

$(256 \times 100)/159256 = 0.16\%$ of total execution time (which is lost and is small compared to total execution time 159256).

We can see that the lost time (and communication events) is small compared to the total execution time, particularly as the problem size increases, so the use of the extrapolated data also seems acceptable for FFT.

8.2.4 Simulation Results: Master/Slave Application (M/S)

I now present the results from my simulation runs for all the example applications. I also present the results by scaling the data size for each application to see the effect of larger problem size on the performance of my partial checkpoint/migrate system. In the results that follow I show multi-graphs with "bars" representing the execution time for different checkpoint/migrate scenarios (such as normal execution, execution including checkpoint/migrate of various frequencies) and horizontal "lines" representing execution time without checkpoint and migration but with an overloaded node causing the overall execution time of the application to slowdown by a certain amount (such as 25% slower than normal execution, 2 times slower, 5 times and 20 times slower). I take the frequency of checkpoint/migrate into account as well by simulating runs with *low* (5 checkpoint/migrate events), *medium* (50 checkpoint/migrate events) and *high* (100 checkpoint/migrate events) frequencies of checkpoint/migration in the simulated execution of the application.

I used Matrix Multiplication as my example Master/Slave application which has the normal three phases, distribution, computation and collection. I gathered the communication information by running the application, first, for a small problem size of $0.02K \times 0.02K$ and repeated the execution several times to gather enough communication information to make predictions (as described later). I use the results of my simulation to determine if the performance of the application increases by doing a partial checkpoint/migrate as compared to when the application is running with an overloaded node causing the performance of the application to decrease.

The problem size that I used in M/S application is small and could be easily run

on a single computer. The reason for keeping the size small was because the M/S communication pattern is highly structured and static and running a realistic, larger problem takes much longer on the limited, single processor, testbed hardware I had available to me (as can be seen from the timings of an actual run of 16 processes on 30 MB data size, presented later). I present the smaller problem size results first for 16 and 8 process runtime environments and then I present some 16 process M/S application results for a bigger problem size (30 MB and 1024 MB of total Matrix Multiplication data). Regardless of problem size, the number of communications events involved in one run of the M/S application is limited. For this reason, gathering communication information from a single run does not give enough communication data to allow TEIRESIAS to find patterns and thus to allow my code to make predictions. I therefore had to run the application several times to collect enough communication information for TEIRESIAS to recognize the patterns of communication and for my predictors to make meaningful predictions. The results shown in Figures 8.33 and 8.37 show similar trends for scenarios where the network bandwidth is reasonable.

16 Process M/S Matrix Multiplication Application-small problem size

The results of my simulation runs for a 16 process M/S matrix multiplication application with total application data size of 0.0045 MB using the collected network characteristics described earlier are shown in Figure 8.33. For multiplication of two matrices, we need three matrices in total. Matrices A and B to be multiplied and the result is stored in matrix C. In my case if both A and B are $0.02K \times 0.02K$ then the result matrix is of size $0.02K \times 0.02K$. A $0.02K \times 0.02K$ matrix has 400 entries

and if each entry is represented by 4 bytes then the total bytes in the matrix is 1600 bytes. This is about 0.0015 MB. So, the total size of all the three matrices A, B and C would be $0.0015 \times 3 = 0.0045MB$. The total size is of the process image (both the data as well as code).

The horizontal lines in the graphs presented represent the simulated execution times with overloaded node(s) (but no migration) causing 25%, 2 times, 5 times and 20 times slowdown in execution, starting from x-axis and up, respectively. The vertical bars from left to right for each migration scenario represent the normal execution time (without any overload and migration), execution time with a checkpoint and migration frequency of 5, execution time with a checkpoint and migration frequency of 50 and execution time with a checkpoint and migration frequency of 100, respectively.

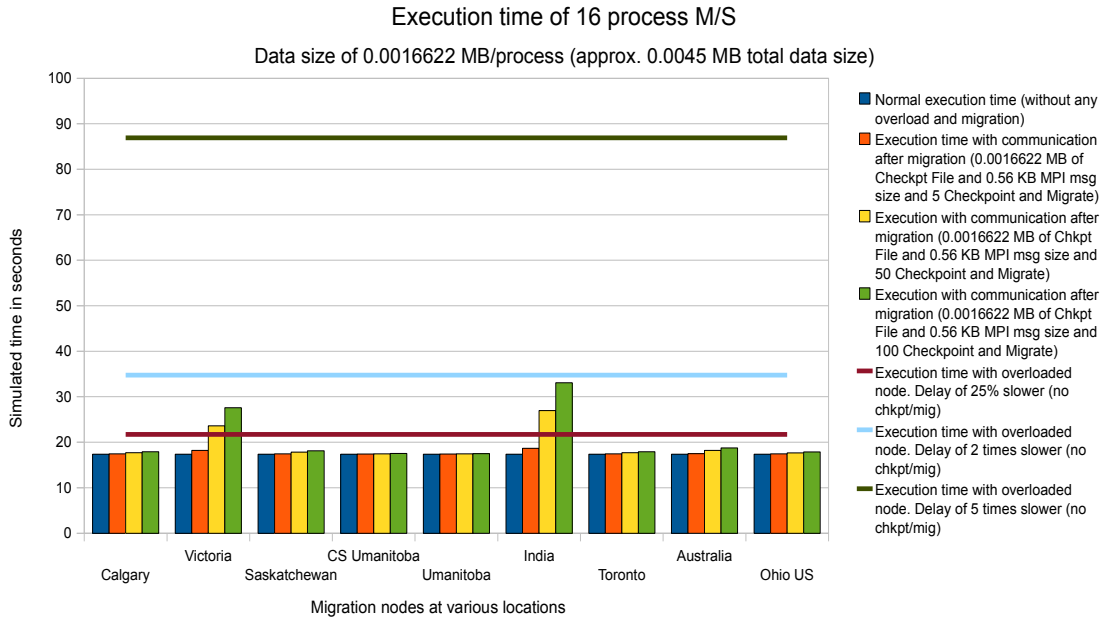

Figure 8.33: 16 Process M/S for $0.02K \times 0.02K$ Problem Size

We can see from Figure 8.33, that it is beneficial to partially checkpoint/migrate for all the considered checkpoint/migrate frequencies[8]. This applies to nodes at all locations when the overload is *low* (first horizontal line from x-axis) except for the pathological cases of India and Victoria in which case when the checkpoint/migrate frequency increases to *medium* (third vertical bar from left for a migration node) and *high* (fourth vertical bar from left for a migration node) then it is only beneficial to checkpoint/migrate when the overload is at least *medium* (second horizontal line from x-axis). This behavior can likely be attributed to the relatively low amount of communication between processes and more computation involved during the execution (i.e. the slaves are relatively compute bound).

The simulation results for an extrapolated problem size where the total data size is 1MB are shown in Figure 8.34. These results suggest that it is still beneficial to do a partial checkpoint/migrate of the application even when the problem size is larger in a 16 process M/S application for all frequencies of checkpoint considered.

8 Process M/S Matrix Multiplication Application-small problem size

With the same problem sizes as for the 16 process M/S matrix multiplication application I ran simulations for 8 process M/S matrix multiplication data and the results are shown in Figure 8.35. The result trends shown in Figure 8.35 show that it is beneficial to checkpoint/migrate in all frequencies of checkpoint/migrate and a *low* overload case. But the execution time bars have moved closer to the *low* overload line.

[8]For example, in the case of migration to a node in Calgary the simulated execution time with communication after migration and checkpoint and migration frequency of 100, it is beneficial to partially checkpoint and migrate when the overload on execution node(s) is causing 25% slowdown (because the 4^{th} vertical bar from the left is well below the 1^{st} horizontal line representing the 25% slowdown due to overloaded node(s)).

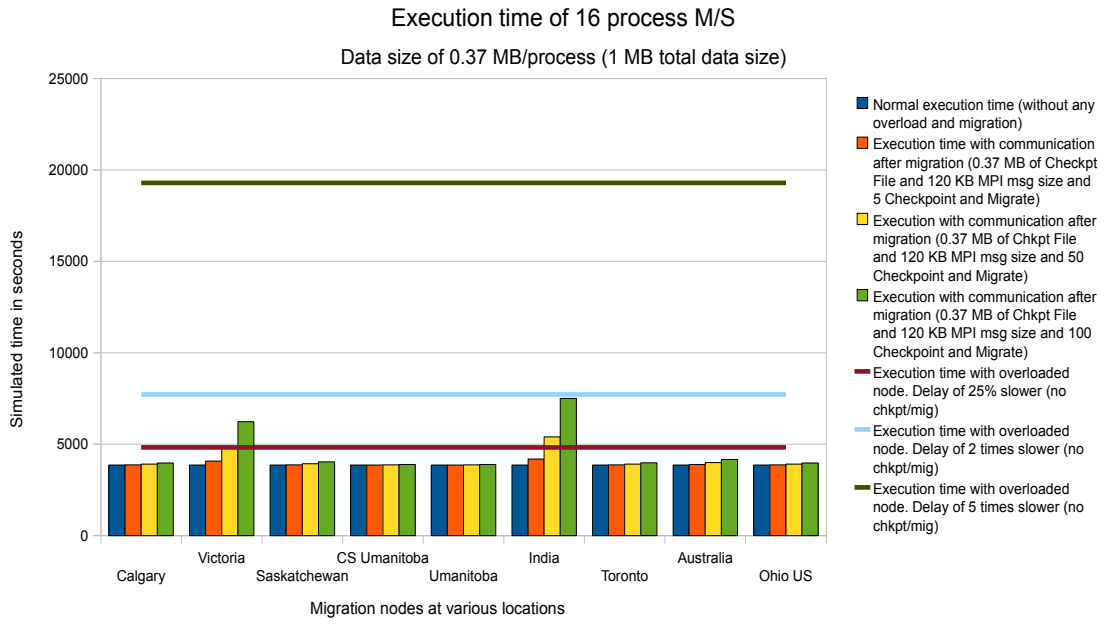

Figure 8.34: 16 Process M/S for 1 MB Problem Size

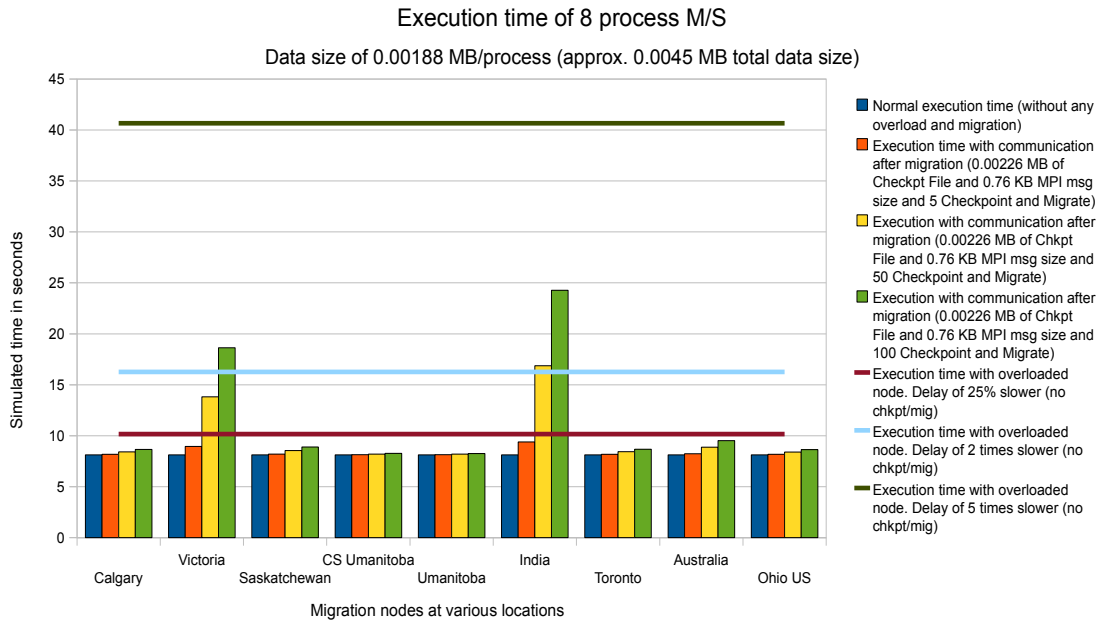

Figure 8.35: 8 Process M/S for $0.02K \times 0.02K$ Problem Size

The bars for Victoria and India have moved up and crossed the *medium* overload line which means that it is no more beneficial to checkpoint/migrate with *high* frequency of checkpoint/migrate when the overload is making the execution 2 times slower (*medium* overload).

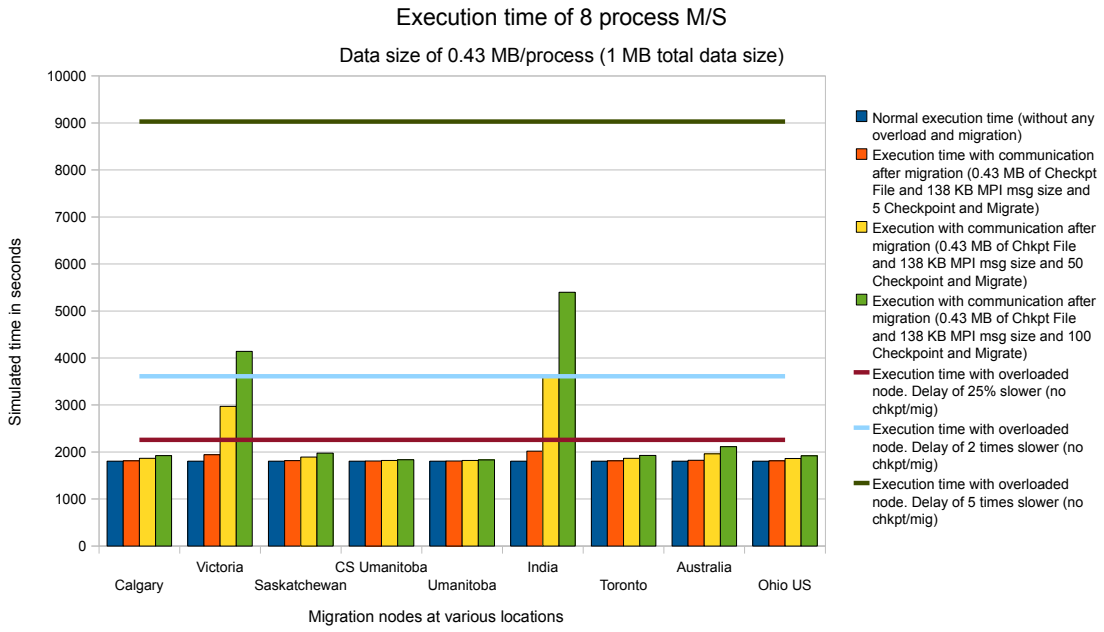

Figure 8.36: 8 Process M/S for 1 MB Problem Size

The results shown in Figure 8.36 are for extrapolated data of 8 process M/S matrix multiplication application for a problem size where total data size is 1 MB and they show us that partially checkpointing and migrating M/S matrix multiplication application is beneficial for all the migration nodes when the overload factor is causing 25% slowdown.

16 Process M/S Application: large problem size

I further ran a bigger problem size (2000 × 2000 matrix) of matrix multiplication approximately 30 MB of total data and ran the application several times on the same size of matrix to collect enough communication data to analyze patterns of communication. Figure 8.37 shows the results of my simulation run on the collected communication data of the application.

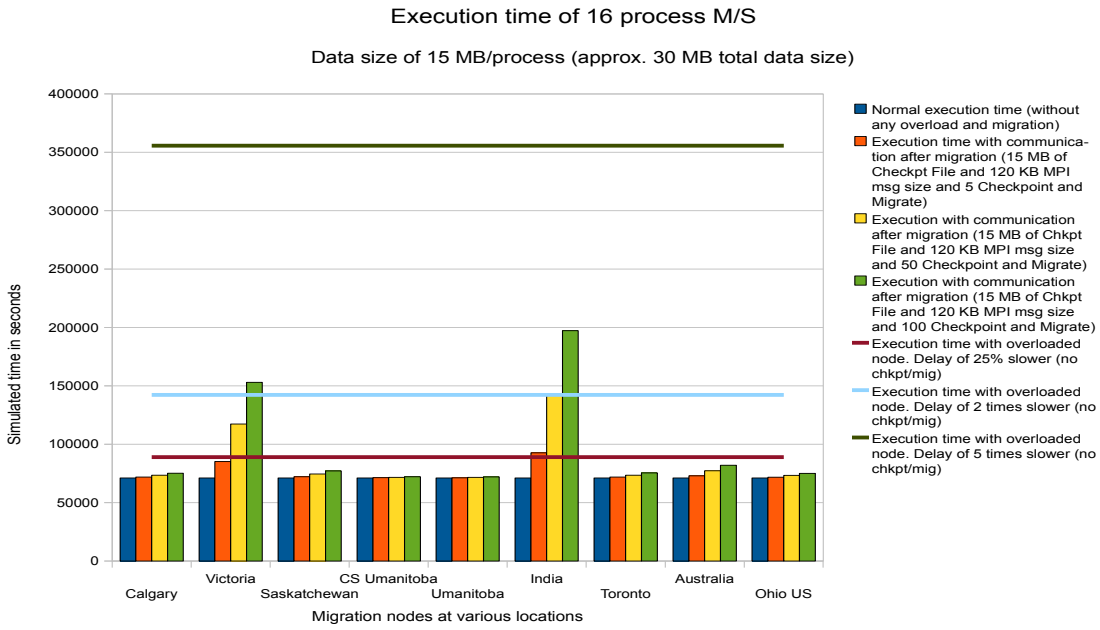

Figure 8.37: 16 Process M/S for 30 MB Data Size

We can see from the Figure that it is beneficial to partially checkpoint and migrate to all the LAN and CWAN nodes when the overload is causing 25% slowdown (represented by first horizontal line from x-axis) and the checkpoint and migrate frequency is *low*. Also, we can see that for almost all the migration locations except for Saskatchewan and India it is beneficial to partially checkpoint and migrate when the overload is causing 25% slowdown, which concurs with the results obtained from

the smaller problem sizes above. I further extrapolate this collected data from 16 process runtime environment and 30 MB of data size to generate communication data of application with 1024 MB data size and present the results next. Due to the long run times of actual run for collection of communication data I chose not to run the application for 8 processes with larger size because the results are expected to be very similar due to highly structured and static communication patterns of the application.

Further, due to the highly structured and static patterns of communication of M/S application I expect that the results of my simulation for a bigger problem size of M/S application on a larger execution environment will be consistent with the results presented above. I will, however, present the simulation results for bigger problem sizes on larger execution environment for the other two applications.

8.2.5 Simulation Results: Finite Element Method (FEM)

For my second simulation application I chose a classic heat transfer problem where each process is given a certain cross-section of a heated $2-d$ area. The initial temperature at the middle of the $2-d$ area is at a high temperature and the boundaries are held at a temperature of zero degrees throughout. Using a Finite Element Method (FEM) technique (as was done in my prediction experiments), each process then computes the temperature of its cross-section over time depending on what the temperature is for its surroundings. For calculation purposes each process has to get temperature values from its neighboring processes which involves communication.I simulated the FEM application on 4, 8 and 16 processes with the same problem size

and the results from the simulation are now presented.

16 Process FEM Application

The first problem size for the FEM application I chose was $1K \times 1K$ which turns out to be approximately 4MB (which is the process image (both code as well as data)) of total application data size and 0.23MB per process (checkpoint data size per process). A cross-section of $1K \times 1K$ has 1,000,000 entries and each entry of 4 bytes makes it 4,000,000 bytes in total. This is approximately 4 MB in size. The MPI message size that is exchanged between processes is constant for all the environment sizes (4, 8, 16 processes) because each process receives a message equal to the size of data immediately adjacent to it. That is, the boundary values only from neighboring processes.

The simulation result for 16 process FEM are shown in Figure 8.38 We can see that frequent communication affects the checkpoint/migrate for various overload conditions. Clearly in the case of LAN and cWAN connected nodes it is beneficial to checkpoint/migrate for all frequencies of checkpoint/migrate if the overload is slowing down the performance by 25% (*low* overload). In the case of CWAN nodes, however, we can see that for almost all cases it is beneficial to partially checkpoint/migrate when the checkpoint/migrate frequency is *low* and the overload is causing a slowdown of only 25% in the performance. When the checkpoint/migrate frequency is increased to *medium* then partial checkpoint and migrate is beneficial only when the overload is at least causing the performance to decrease by 2 times (*medium* overload). These results hold except for Victoria for the reason already described. In WWAN cases

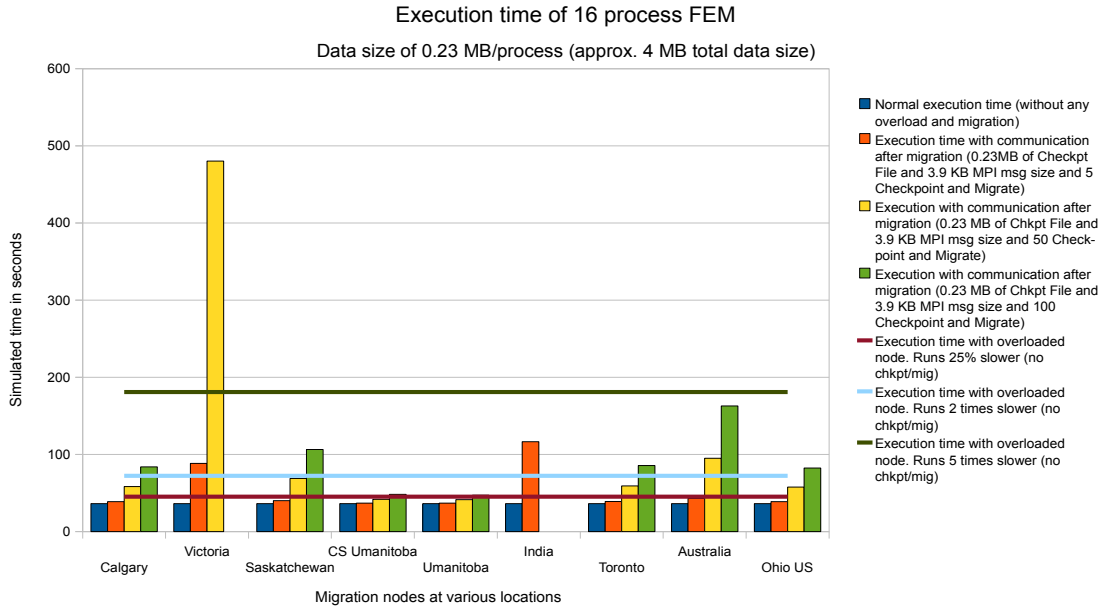

Figure 8.38: 16 Process FEM for $1K \times 1K$ Problem Size

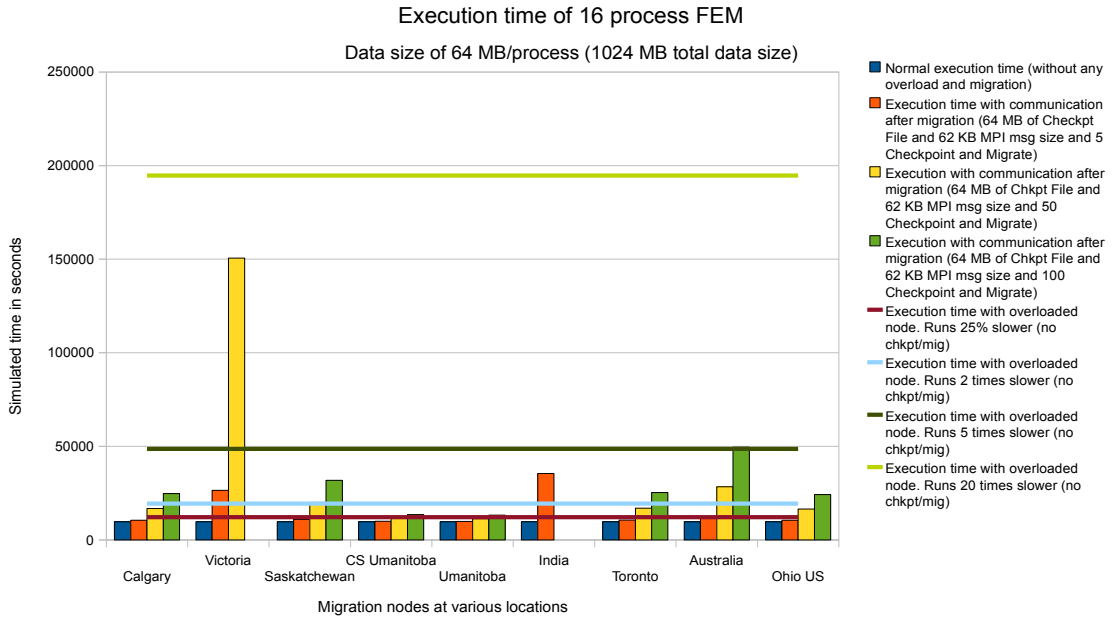

Figure 8.39: 16 Process FEM for $16K \times 16K$ Problem Size

only migration to Ohio US performs comparably to CWAN nodes. This is likely due to the existence of a high speed link from the University of Manitoba to Chicago providing high bandwidth and reasonable latency. Note that I removed the execution times for partial checkpoint/migrate with *high* checkpoint/migrate frequency for India and Victoria to avoid skewing the graph and making differences between the other results difficult to see. Migrating processes to Victoria and India will not preserve performance even in the *low* checkpoint/migrate frequency when the overload is causing only a 25% slowdown (*low* overload), where as the Australia nodes still perform well, probably because of a reasonably high bandwidth connection between Canada and Australia.

With the increase in problem size to $16K \times 16K$ on the 16 process FEM, shown in Figure 8.39, the performance degrades a bit (e.g. the WWAN Australia nodes). The checkpoint data size is now 64MB per process and the MPI message size increases proportionately. Saskatchewan nodes gave good performance with smaller problem size for *medium* checkpoint/migrate frequency and *medium* overload but for bigger problem sizes it is not beneficial to migrate in the same cases. Clearly, the increased communication has an impact of the usefulness of partial checkpoint/migrate.

8 Process FEM Heat Transfer Application

For the same problem size ($1K \times 1K$) the results of simulation runs are shown for an 8 process FEM heat transfer application in Figure 8.40. The total data size is approximately 4MB and the checkpoint data size per process is 0.47MB. The MPI message size remains the same as in the 16 process case, as described earlier. From

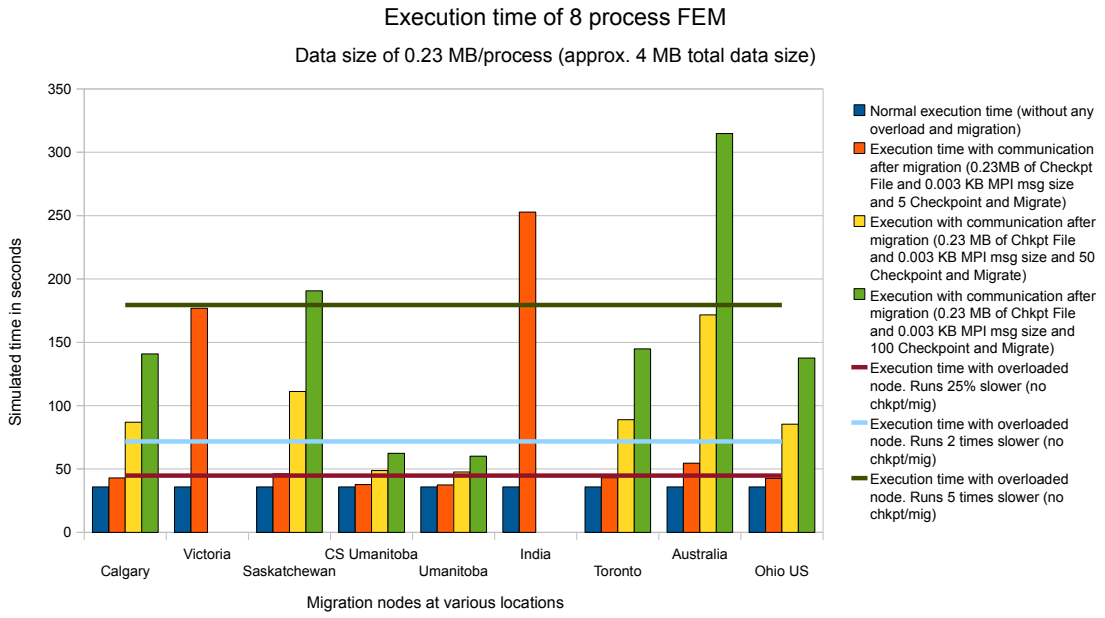

Figure 8.40: 8 Process FEM for $1K \times 1K$ Problem Size

Figure 8.40, we can see that it is no longer beneficial to partially checkpoint/migrate if the checkpoint/migrate frequency is *medium* or above for overload causing even 25% slowdown in performance in the LAN and cWAN nodes. This is likely due to the increase in the checkpoint data size per process. Even though the computation time increases because of using less processes, the size of checkpoint plays a determining role (with fewer processes and the same size of data as with 16 processes, the data size distributed per process increases) when we would choose to partially checkpoint/migrate to various geographically distributed nodes. I have removed the *extremely high* overload (20 times slower) line and, again, the bars for India and Victoria to avoid a skewed graph. It is clear from Figure 8.40 that for *medium* and *high* frequencies of checkpoint/migrate and CWAN connected migration nodes, it is beneficial to partially checkpoint/migrate when the slowdown is at least 5 times or

higher in the 8 process FEM heat transfer application.

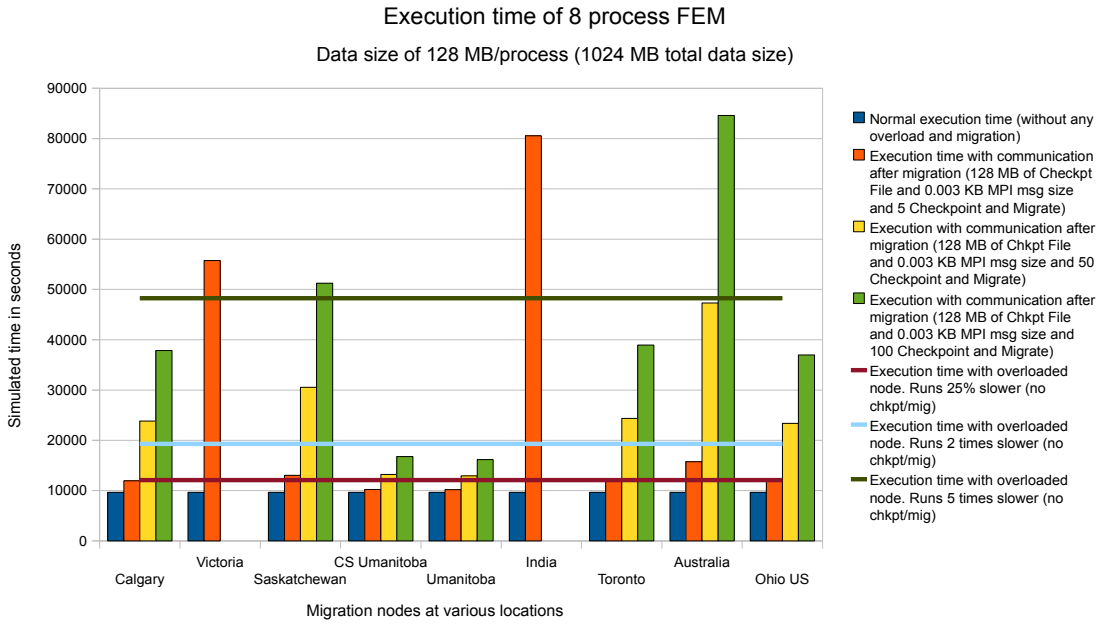

Figure 8.41: 8 Process FEM for $16K \times 16K$ Problem Size

I extrapolated the data from $1K \times 1K$ 8 process FEM to $16K \times 16K$ with a checkpoint data size of 128MB per process and MPI message size proportionately increased. The simulation results are shown in Figure 8.41 and show that the performance deteriorates when the problem size is increased, as the checkpoint data size also increases per process resulting in greater migration overhead.

256 Process FEM Heat Transfer Application

I extrapolated the data I collected from 16 process FEM application to represent communication data for a 256 process FEM. For extrapolation I simply picked up the communication events representing communication between, for example, $P0$ and $P1$ and changed it to represent communication between $P16$ and $P17$, etc. Then

I replicated the data to make the problem size sufficiently large to represent a real world large application running on 256 processes environment. The total data size turned out to be approximately 12 GB with approximately 50 MB of data per process. The simulation results for 256 processes working on approximately 12 GB of data are shown in Figure 8.42.

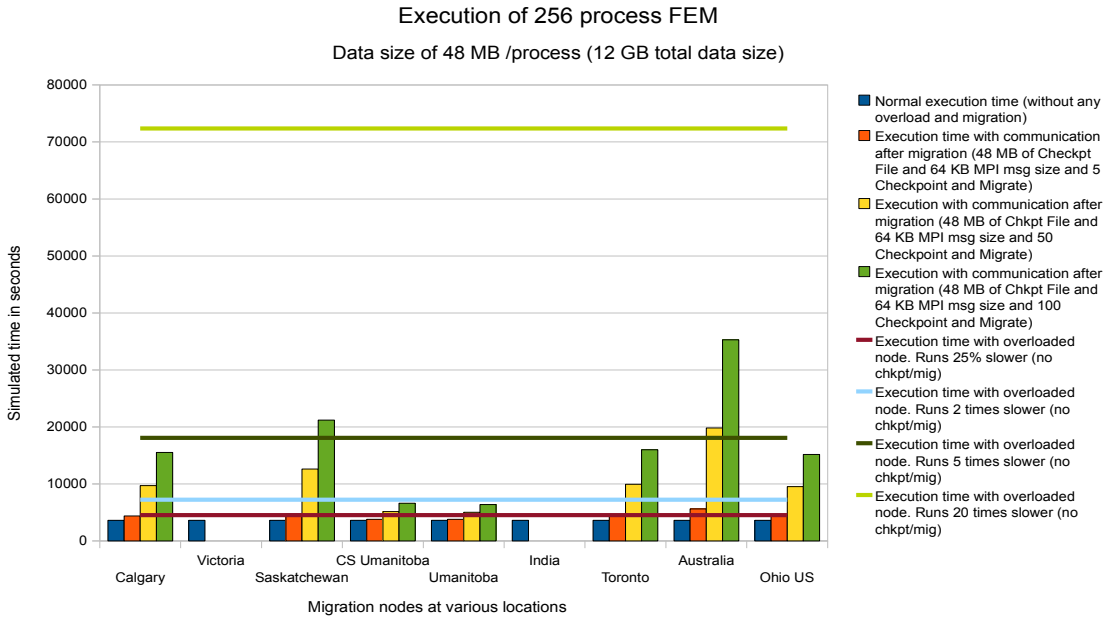

Figure 8.42: 256 Process FEM for $56K \times 56K$ Problem Size

We can see from the Figure that when overload is causing a 25% slowdown (represented by the 1st horizontal line from x-axis) then it is beneficial to partially checkpoint and migrate in almost all the North America nodes (LAN, CWAN and Ohio) when the checkpoint and migrate frequency is *low*. For the overload causing slowdown by 2-times (represented by the 2nd line from the x-axis) and for a *medium* checkpoint and migrate frequency it is only beneficial to partially checkpoint and migrate for LAN nodes. Again, I have removed the bars for India and Victoria to

prevent skewing of graph.

8.2.6 Simulation Results: FFT Application

I used a $2-d$ FFT MPI program and executed it on 16, 8 and 4 processes. To collect sufficient communication information, I ran the same program several times on the same problem size. As in the case of M/S application, due to restriction in running a bigger problem size on my test machine, I had to run a do-able problem size several times to collect the data. Collecting the data from one run does not give enough communication data and execution time to analyze data to make predictions of patterns. Collecting the communication data through multiple runs of the application gives sufficient communication data to analyze the communication patterns and enough window to simulate a checkpoint request and make predictions of upcoming patterns.

This is reasonable since it reflects the case where a large FFT problem is solved using a sequence of parallel FFTs done on subsets of the problem data with the final final result being computed sequentially from the partial results obtained from the parallel runs. This approach is preferable to a single large parallel run as it permits overlap of computation with the I/O required to load FFT data points from storage. It also provides a natural source of recurring communication patterns that would not be present in a single parallel FFT run.

16 Process FFT Application

The results of my simulation runs for the 16 process FFT case are shown in Figure 8.43. The problem size I chose was $4K \times 4K$, which results in approximately 4MB of data per process, which dominates the checkpoint data size associated with each process. (In terms of points a $4K \times 4K$ image corresponds to approximately 16,000,000 points.) We can see in Figure 8.43 that if the migration nodes are in

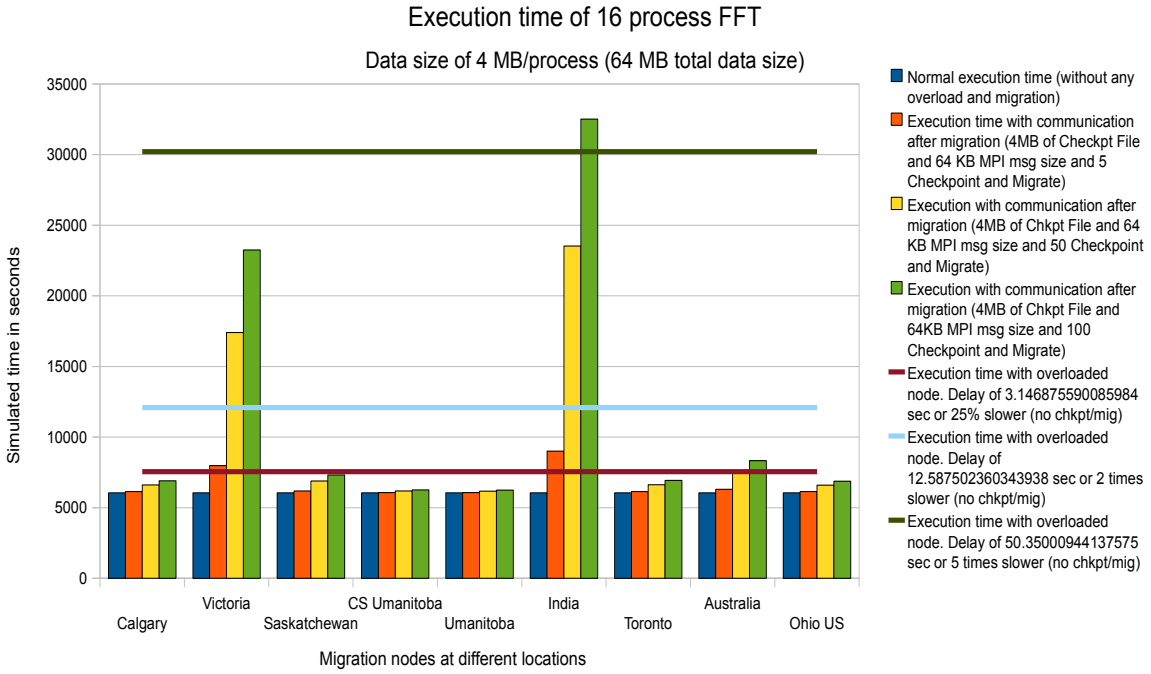

Figure 8.43: 16 Process FFT for $4K \times 4K$ Problem Size

a LAN (CS Umanitoba) or cWAN (Umanitoba) environment, even when there were 100 checkpoint/migrate events (*high* frequency) it is still beneficial to do a checkpoint and migrate when the overload is 25% or more. In the CWAN case (Calgary, Victory, Saskatchewan and Toronto), it is beneficial to checkpoint/migrate in all cases for Calgary and Toronto but it is probably not beneficial for Saskatchewan when the

checkpoint/migrate frequency is *high* (The slow link is possibly due to the fact that the FTP download, while performing the bandwidth test, was done by downloading a big file from Chemistry department server which could be connected over a slower link). For the migration nodes in Victoria it is not beneficial even in the case of only 5 checkpoint/migrate events (*low* frequency). In the WWAN cases (Ohio USA, Australia and India), we can see that migrating to Ohio USA is much better than Saskatchewan and as good as Calgary and Toronto. Australia nodes in the *low* frequency of checkpoint/migrate case performed better than Saskatchewan with *medium* frequency of checkpoint/migrate and it appears to be beneficial to checkpoint/migrate in the *low* checkpoint/migrate frequency case when the overload is causing 25% slowdown. Surprisingly, even for India it seems beneficial for the *low* checkpoint/migrate frequency case if the overload is *medium* (causing the application to run 2 times slower) or higher. One reason for such behaviour might be that the size of the predicted group of processes is very small leading to a small checkpoint data size to be migrated. This is because in the FFT application there are several communication phases and each time we proceed in execution from one communication phase to the next, the butterfly communication happens among a smaller group of processes (recall Figure 8.1, Phases I, II and III). It might have happened that at the checkpoint arrival time the simulation was executing the smaller phases of communication.

I have communication data for 16 process FFT for a problem size of $4K \times 4K$ and to study a problem size of $16K \times 16K$ for a 16 process environment, I extrapolated the data by simply replicating the communication data 16 times. Because $16K \times 16K$ for 4 bytes of float will become 1024 MB and since I have communication data for

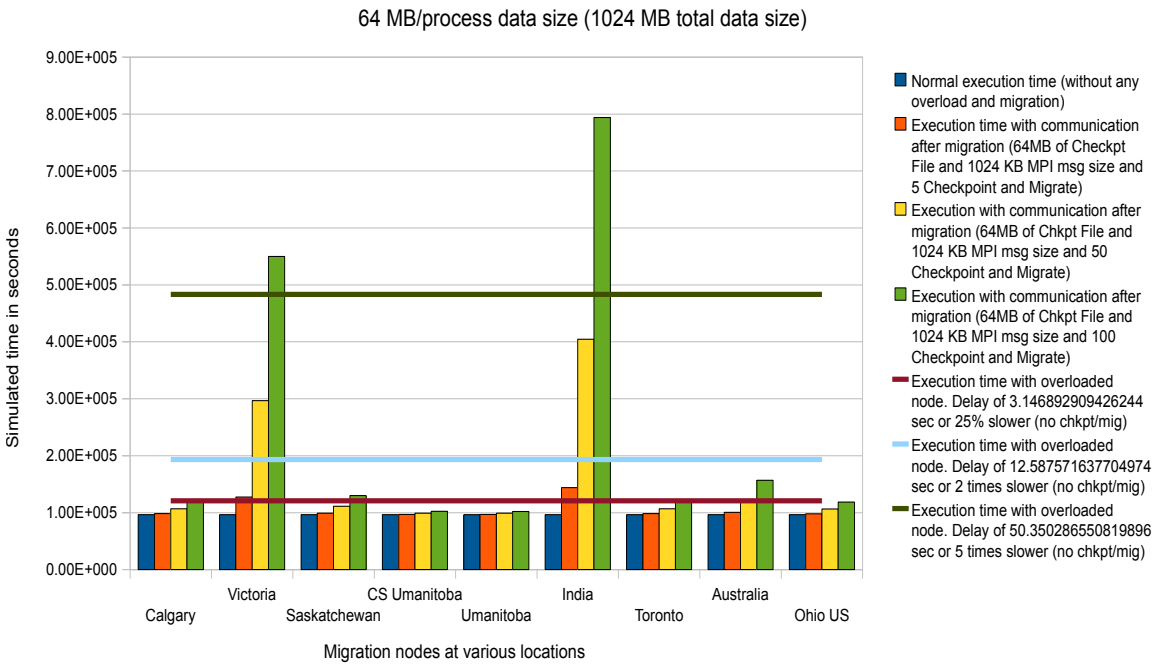

Figure 8.44: 16 Process FFT for $16K \times 16K$ Problem Size

$4K \times 4K$ for 4 bytes of float equals 64MB problem size, I just have to replicate the 64MB problem size data by 1024/64=16 times.

The results for the $16K \times 16K$ problem size with 16 processes for FFT (64MB of checkpoint data per process) are shown in Figure 8.44. With a problem size of $16K \times 16K$ we can see that for nodes in Calgary, Saskatchewan, Toronto and Ohio there will be no benefit if the checkpoint/migrate frequency is *high* and the overload is *low*.

Results for another problem of size $64K \times 64K$ (which was extrapolated 256 times from the $4k \times 4K$ data for the 16 process FFT) with 1GB checkpoint file size is shown in Figure 8.45. We can see that it is still beneficial to partially checkpoint and migrate in LAN connected nodes for all checkpoint/migrate frequencies when overload

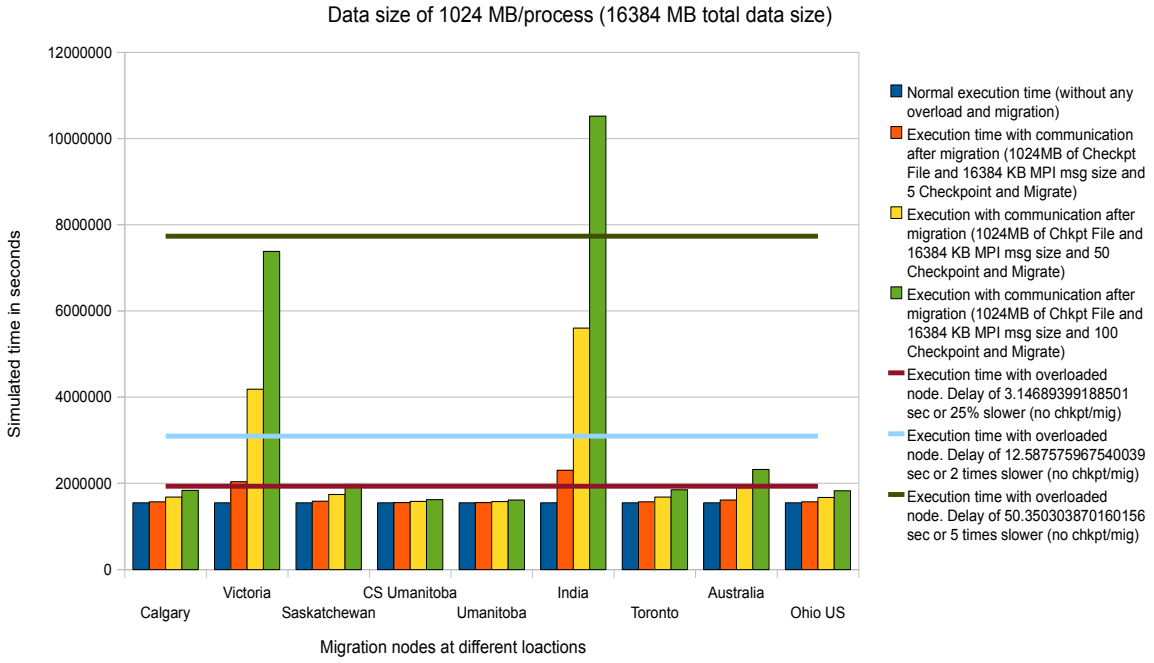

Figure 8.45: 16 Process FFT for $64K \times 64K$ Problem Size

is causing only 25% slowdown. In the case of CWAN connected nodes, Toronto, Calgary and Saskatchewan, when the checkpoint/migrate frequency is *high* and overload is causing 25% slowdown then partial checkpoint/migrate is not beneficial. Victoria nodes are an exception for the reasons already known. Ohio nodes perform equally well as CWAN nodes because of the connection between Manitoba and Chicago.

8 Process FFT Application

The first problem size I chose for the 8 process FFT application was $4K \times 4K$, which makes the checkpoint data size 8MB per process and the total data size 64MB for the application. The simulation results for my 8 process FFT case are shown in Figure 8.46. We can again see that in the case of LAN and cWAN nodes it

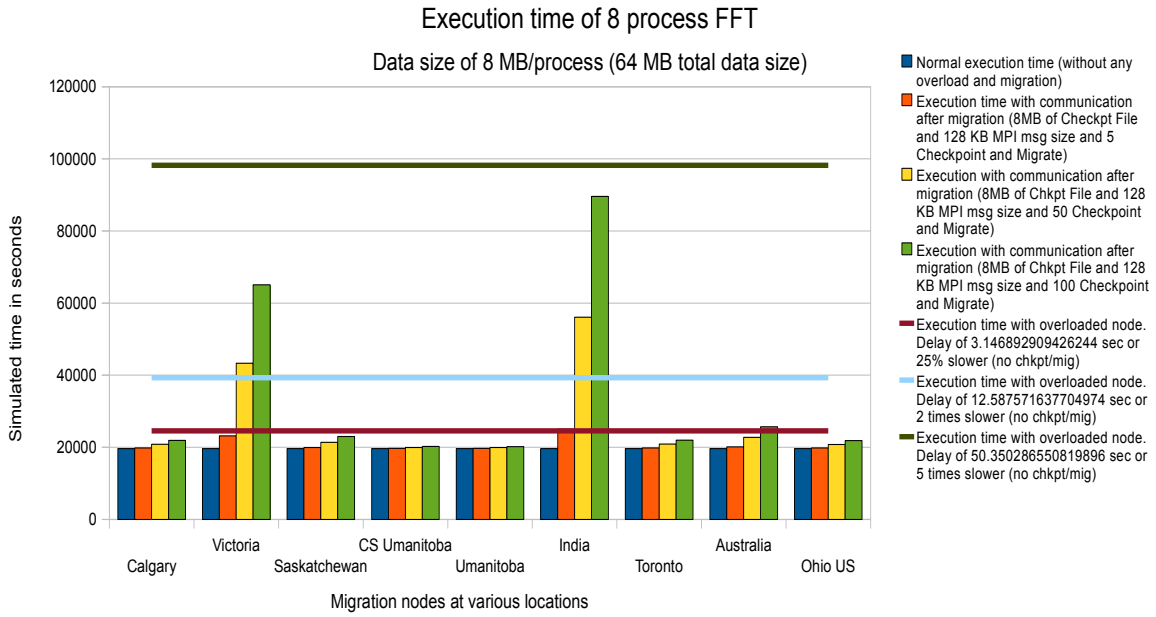

Figure 8.46: 8 Process FFT for $4K \times 4K$ Problem Size

is beneficial to checkpoint/migrate for all checkpoint frequencies (*low*, *medium* and *high*) even if the overload is slowing down the execution by 25% (*low* overload). In the case of CWAN it is also beneficial to checkpoint/migrate for all checkpoint/migrate frequencies when the overload is slowing down the performance by 25% except for Victoria (for the reason already explained), which is only good in the case of *low* checkpoint/migrate frequency for all overloads. For WWAN (India, Australia, Ohio) we can see that Ohio meets the performance requirement but India fails even in the *low* checkpoint/migrate frequency case for overload slowdown of 25% (*low* overload). In *medium* and *high* checkpoint/migrate frequencies, it is only beneficial to migrate processes to nodes in India if the slowdown is 5 times or more.

Figure 8.47 shows an extrapolated simulation run of the 8 process FFT on a problem size of $16k \times 16K$ resulting in 128MB checkpoint file size per process and a

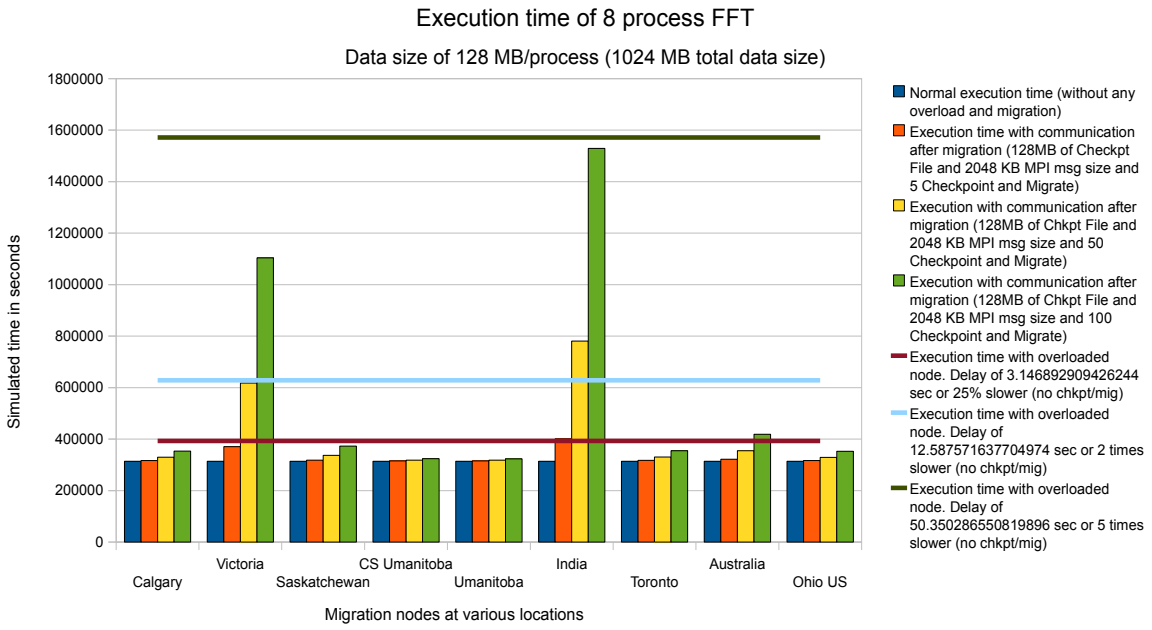

Figure 8.47: 8 Process FFT for $16K \times 16K$ Problem Size

total data size of 1024MB for the application. The results for the 8 process FFT for a problem size of $16K \times 16K$ are similar to the $4K \times 4K$ problem size.

I also extrapolated the problem size to $48K \times 48K$ by replicating the data for $4K \times 4K$ to make a checkpoint data size of 1024MB per process and the results for the simulation run are shown in Figure 8.48. We can see that it is now beneficial to checkpoint and migrate for all the LAN and CWAN nodes, except for Victoria, in all the cases of checkpoint/migrate frequency when the overload is causing 25% slowdown. Ohio and Australia still perform reasonably well with the exception of Australia performing poorly in the case of *high* checkpoint/migrate frequency.

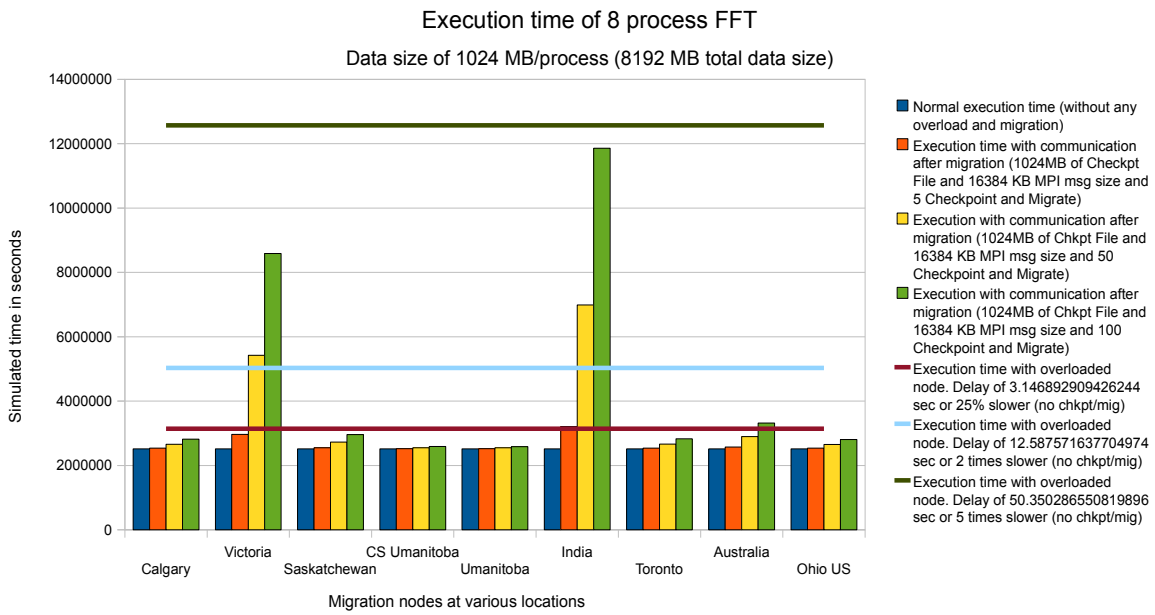

Figure 8.48: 8 Process FFT for $48K \times 48K$ Problem Size

4 Process FFT Application

For the same problem size ($4K \times 4K$) I ran simulations for a 4 process FFT and the results are shown in Figure 8.49. The checkpoint data size per process was 16MB and the total application data size was 64MB. We can see that the trend has changed a bit because of fewer processes being involved which increases the checkpoint data size. For the LAN and cWAN cases we can still benefit by partially checkpointing and migrating the processes in all cases of checkpoint/migrate frequencies when the overload is *low* (25%) or higher. But, in the case of CWAN we can see that for almost all locations it is no longer beneficial to checkpoint/migrate when the checkpoint/migrate frequency is *high* and the overload is *low* (25%). The reason for this behavior can be attributed to the way the data is distributed and how much communication is going on between the processes, which eventually decreases the execution time of the application for

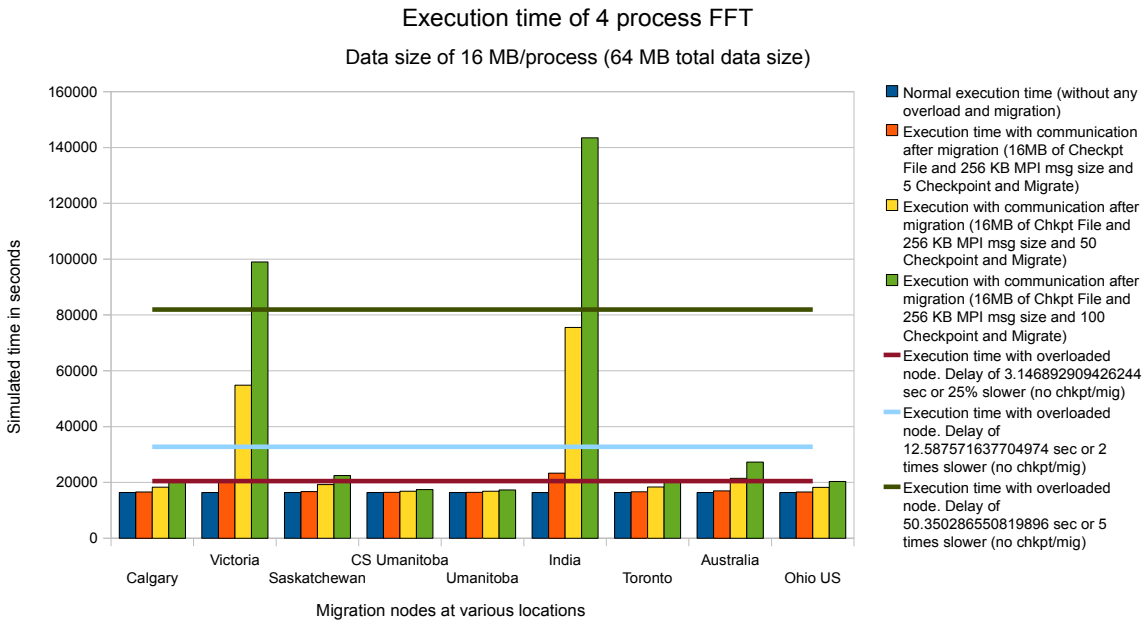

Figure 8.49: 4 Process FFT for $4K \times 4K$ Problem Size

the same problem size as compared to the 8 and 16 process FFT cases. When there are less processes involved, the data size on which each process is working increases and the number and frequency of the communication steps in the "butterfly" pattern decreases (due to locally available data), so there is more computation involved per process than communication with other processes (which has the effect of speeding up the computation per process).

Finally, Figure 8.50 shows the results from my simulation runs for a 4 process FFT with a problem size of $16K \times 16K$. The data from the $4K \times 4K$ was extrapolated to make the checkpoint data size per process be 256MB (by replicating the data 16 times). Figure 8.51 shows the results of the simulation runs for a problem size of $32K \times 32K$ which is extrapolated by using data for $4K \times 4K$. The trends in both cases are the same as for the $4K \times 4K$ results.

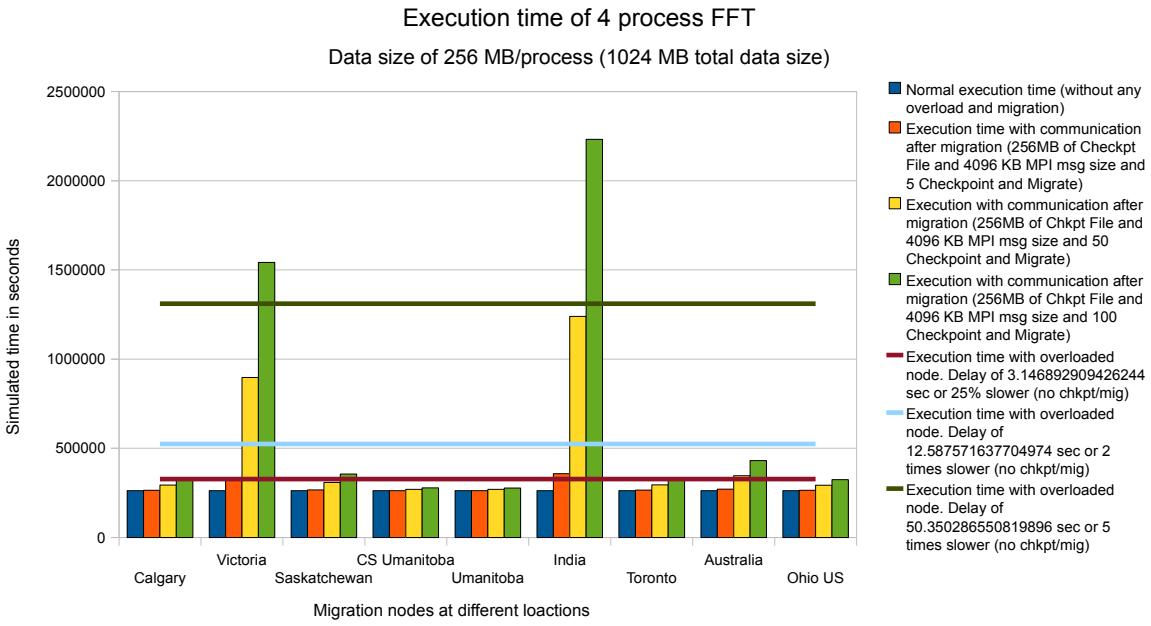

Figure 8.50: 4 Process FFT for $16K \times 16K$ Problem Size

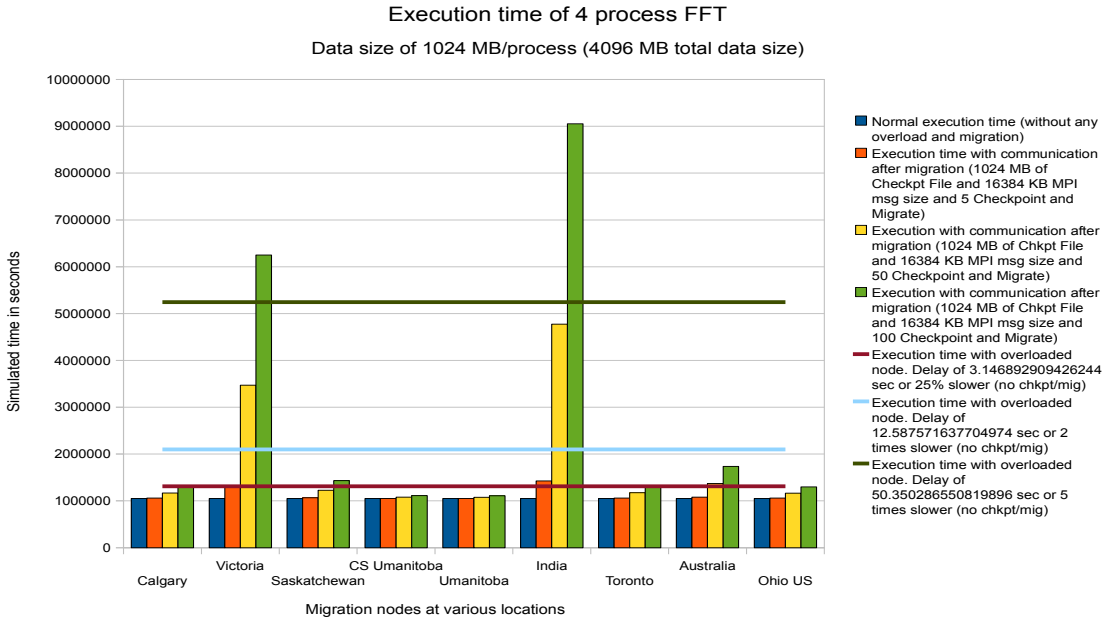

Figure 8.51: 4 Process FFT for $32K \times 32K$ Problem Size

256 Process FFT Application

I, again, extrapolated the data I collected from the 16 process FFT application to represent communication data for a 256 process FFT. For extrapolation I, again, simply picked up the communication events representing communication, for example, between $P0$ and $P1$ and changed them to represent communication between $P16$ and $P17$, etc. Then I replicated the data to make the problem size sufficiently large to represent a real world large application running on 256 processes. The total resulting data size was approximately 60 GB with approximately 256 MB of data per process. The simulation results for 256 processes working on approximately 60 GB of data are shown in Figure 8.52.

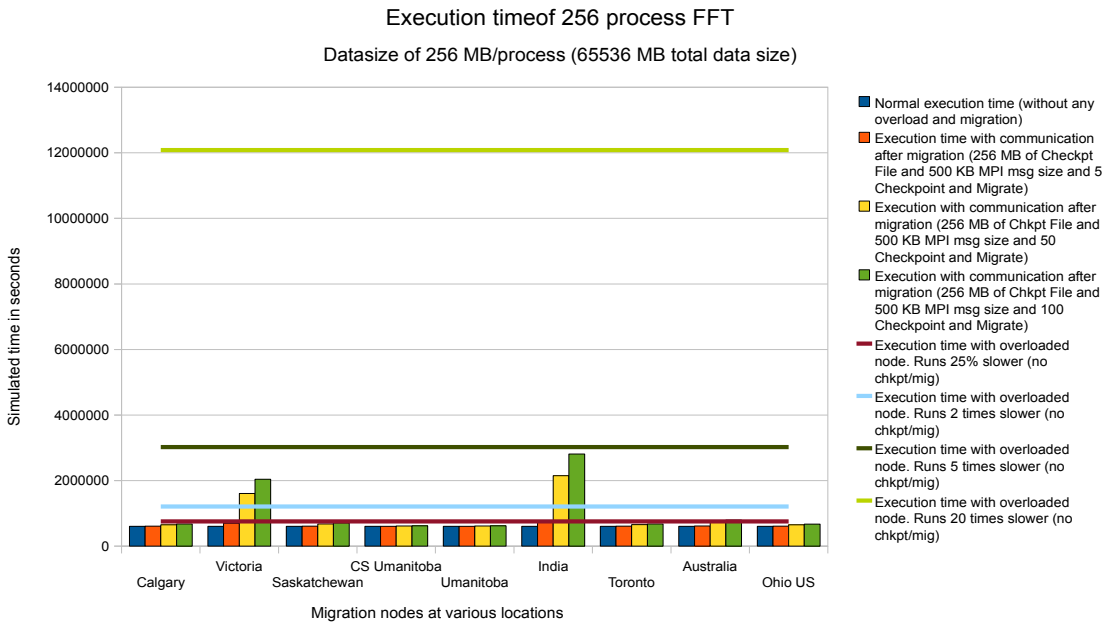

Figure 8.52: 256 Process FFT for $130K \times 130K$ Problem Size

We can see from that on a larger execution environment for a bigger problem size the behaviour of the partial checkpoint and migration is similar to the smaller

problem sizes on smaller execution environments. This behaviour is likely due to the highly structured communication pattern of the FFT application. We can also see that with 25% overload it is beneficial to migrate to LAN and CWAN nodes for all the frequencies of checkpoint and migration. I chose not remove the 20 times slowdown horizontal line in this graph to show how the graphs get skewed. I prevented this skewing in earlier results presented by removing the extreme values.

8.2.7 Effect of Aggressive Communications on Partial Checkpoint/Migrate Performance

After studying the real-world applications and their performance behavior using my partial checkpoint/migrate system, where I found all the applications that I consistent benefits for higher bandwidth networked nodes, I decided to create applications that exhibited very aggressive (frequent and larger volume) communications that I expected to not benefit from my system even when the checkpoint/migrate frequency is *low* and the overload is *low*. Essentially, I wanted to explore the limits of usefulness of my system and also to reinforce my confidence in my results. From the FEM results, where the communication is more aggressive than the other two applications, we have seen variations in the trends by varying the checkpoint/migrate frequency and overload on the nodes. In FEM the data size that is communicated between processes is quite small as compared to the data size each process works on, which suggests that the MPI message size and the number of communications between processes must play a major role in determining whether my system is useful or not. With this in mind I created an unusual Matrix Multiplication family of MPI

applications with various communication intensities.

A well written Matrix Multiplication is designed in such a way that the Matrix is divided into fairly equal parts and the parts (if decomposed by row, then rows) distributed to each process. Each process then works on its part of the data independently and sends the result back to the process that manages the computation. The amount of data that is exchanged between the master and each slave is an equal number of rows of matrix A, matrix B and then the result row of Matrix C is returned.

I re-designed this application to increase the communication frequency and amount of data that is exchanged between processes, first by increasing the communication intensity by sending 1 row at a time and the entire matrix B to each process for various problem sizes. The size of Matrix A in this test was 15000×15 and, B was 15×7. For the total data size of approximately 1MB the results of running the simulation are shown in Figure 8.53. The average MPI message size exchanged between processes was 0.26KB and the checkpoint data size per process was 0.0005MB. We can see from Figure 8.53 that the LAN and cWAN nodes still perform well but CWAN nodes and WWAN nodes start to touch the overload line of 25% slowdown.

Increasing the problem size to 5MB of data for which the results are shown in Figure 8.54, we can see that LAN and cWAN nodes also perform poorly for the *low* overload (25%) case. The CWAN nodes (Calgary, Saskatchewan and Toronto) now touch the 2 times slower mark and the WWAN Australia node now crosses the 2 times slower mark for all the checkpoint/migrate frequencies.

The results shown for approximately 20MB of problem size in Figure 8.55 confirm that the volume of communication play a major role in determining whether my

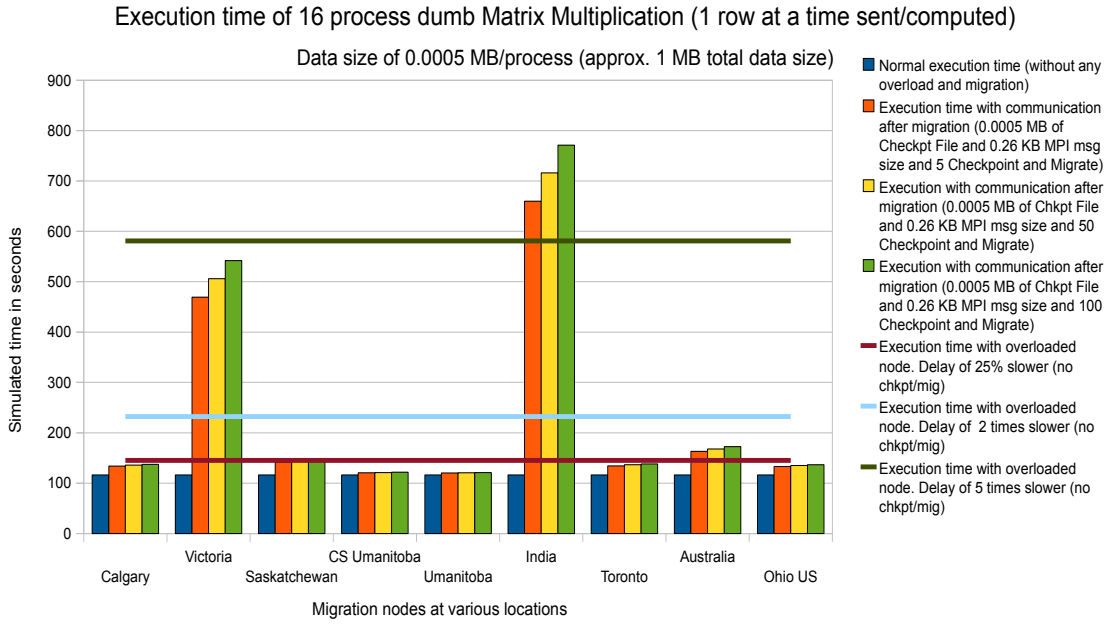

Figure 8.53: 16 Process Matrix Multiplication for 1 MB Problem Size

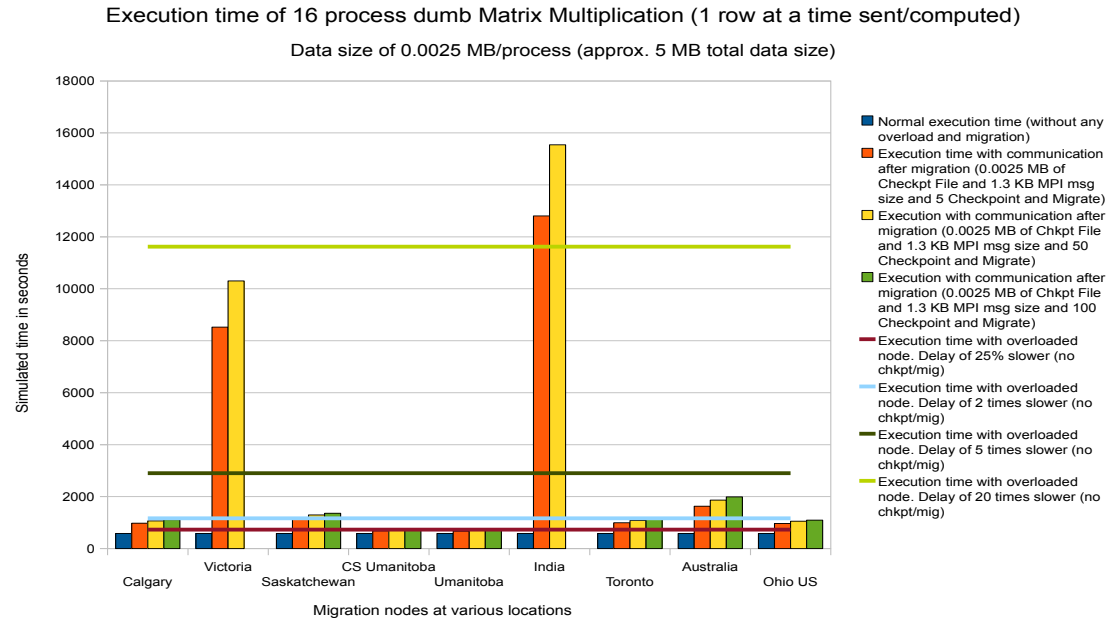

Figure 8.54: 16 Process Matrix Multiplication for 5 MB Problem Size

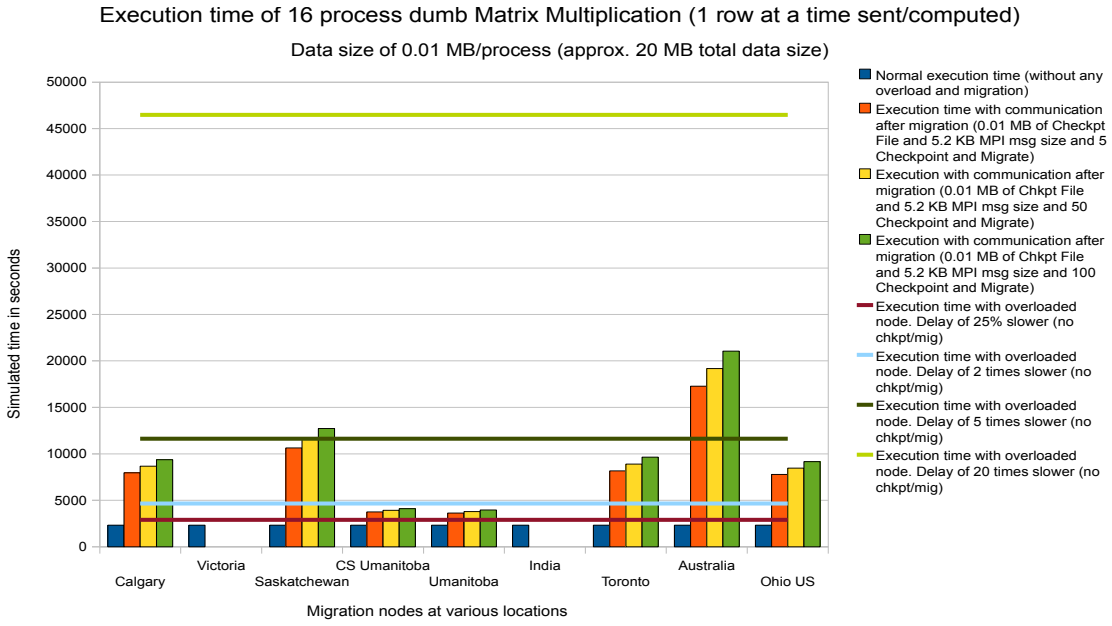

Figure 8.55: 16 Process Matrix Multiplication for 20 MB Problem Size

system is useful of not.

I performed another experiment, this time sending even more (unnecessary) data when communicating but computing just one row at a time. Rather than sending just one row of Matrix A and Matrix B as in the previous case, I now send a quarter of the Matrix A along with Matrix B to each slave process, but slave processes just compute one row and return the result. So if there are 15000 rows I send 3750 rows to each process along with the 15×7 B matrix. The results after running the simulation on the data collected are shown in Figure 8.56. The total application data size is approximately 1MB and the checkpoint data size per process is 0.21MB with an MPI message size of 110KB (average message size). The bars for Victoria and India are removed to avoid a skewed graph. We can see that even for the LAN and cWAN nodes my system now fails to give performance benefit. I again increased the size

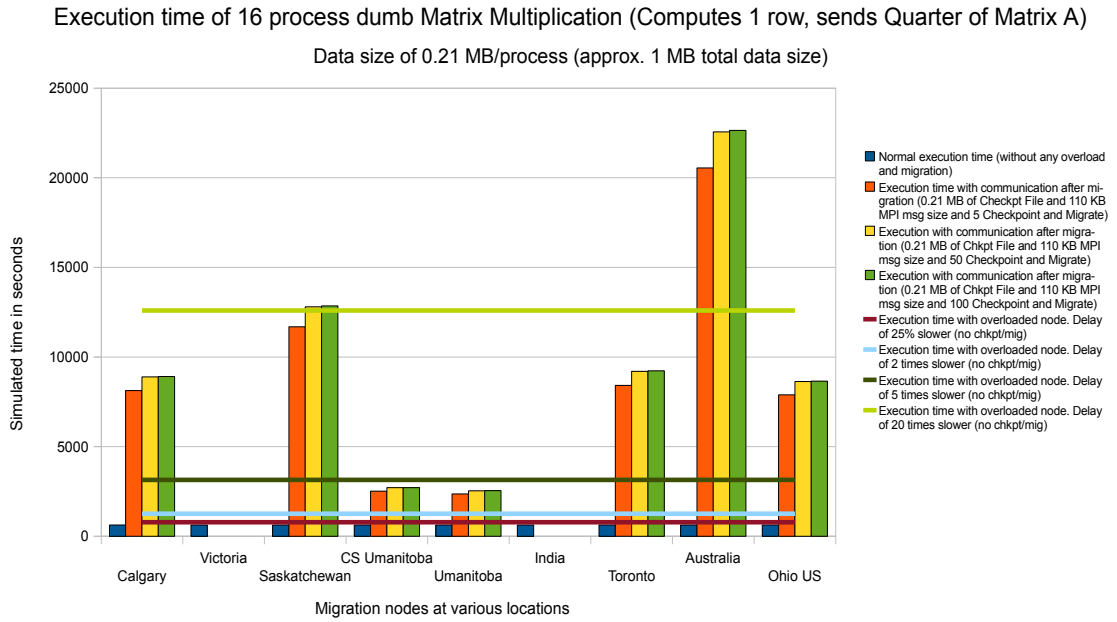

Figure 8.56: 16 Process Matrix Multiplication for 1 MB Problem Size

of problem this time to 2MB and ran the simulation and the results are shown in Figure 8.57. With the increase in data size the message size and checkpoint data size increases and performance deteriorates rapidly as evident from Figure 8.57.

Furthermore, I performed another experiment. This time I increased the MPI message size drastically by sending the entire Matrix A along with Matrix B to each slave process though each slave still computed only one row at a time. The results for this horrific version of Matrix Multiplication with a total data size of 1MB, checkpoint data size of approximately 0.86MB and MPI message size of approximately 500KB are shown in Figure 8.58.

We can see from Figure 8.58 that even in the case of LAN and cWAN nodes the partial checkpoint/migration is not beneficial even when the checkpoint/migrate frequency is *low*. The performance clearly degrades as we move to a bigger problem

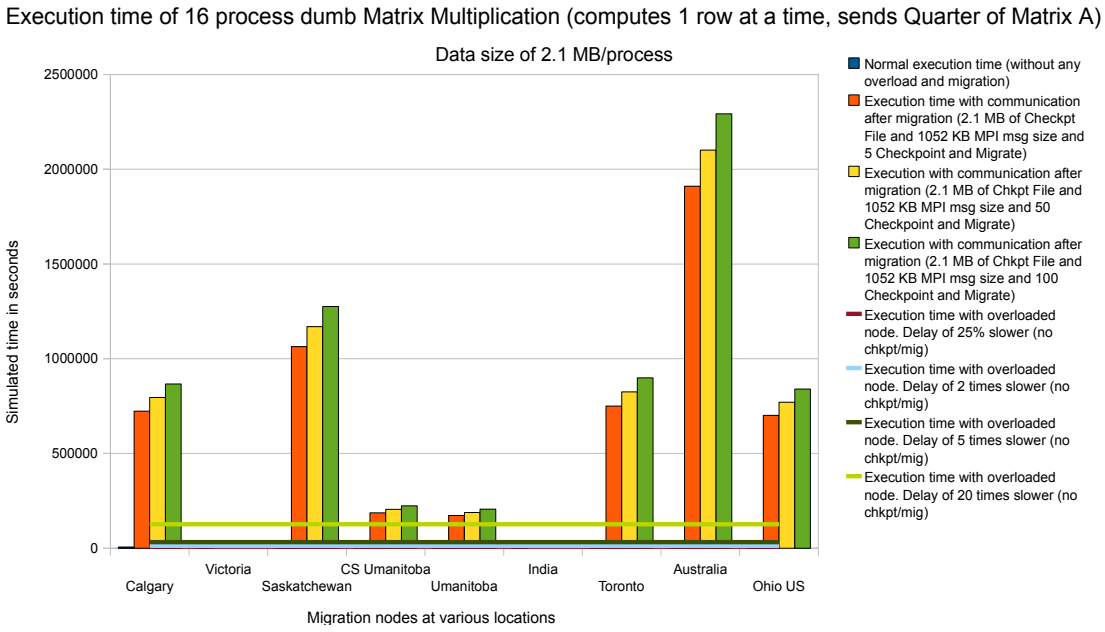

Figure 8.57: 16 Process Matrix Multiplication for 2 MB Problem Size

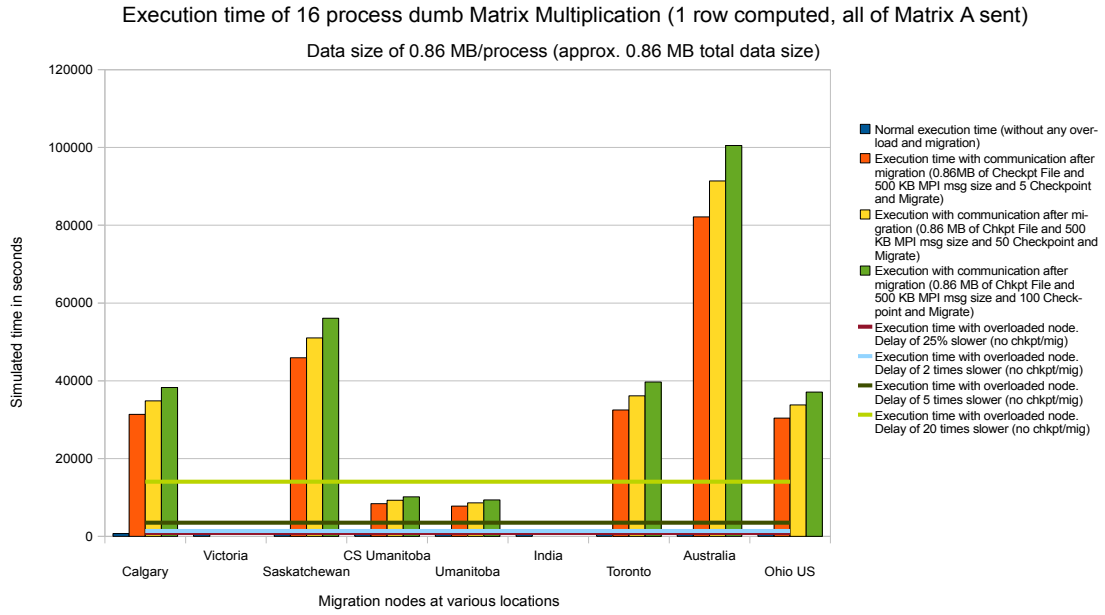

Figure 8.58: 16 Process Matrix Multiplication for 1 MB Problem Size

size as can be seen in the following Figures.

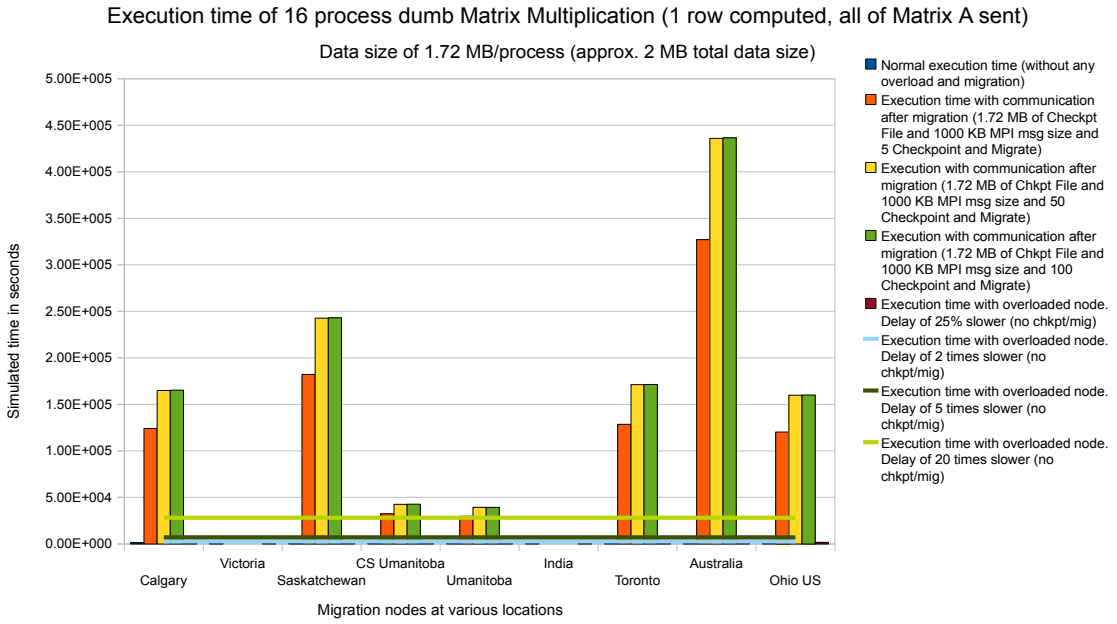

Figure 8.59: 16 Process Matrix Multiplication for 2 MB Problem Size

Figure 8.59 shows the simulation results for a problem where the total application data size is approximately 2MB and the checkpoint data size per process is 1.72MB with approximately 1000KB of MPI message size. The performance deteriorates even more and the bars for LAN and cWAN crosses the 20 times slowdown line. Figures 8.60 and 8.61 show the results for problems where the total application data size is approximately 4MB and 64MB respectively.

The Figures show skewed graphs where the execution times of the application with partial checkpoint/migrate cross the 20 times slower execution of application line.

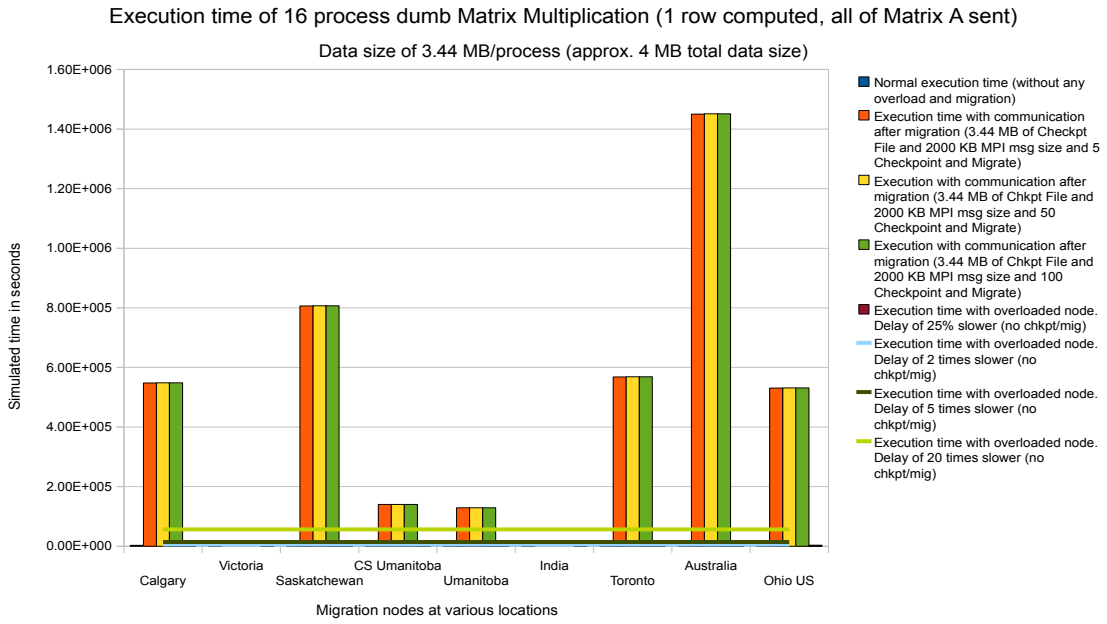

Figure 8.60: 16 Process Matrix Multiplication for 4 MB Problem Size

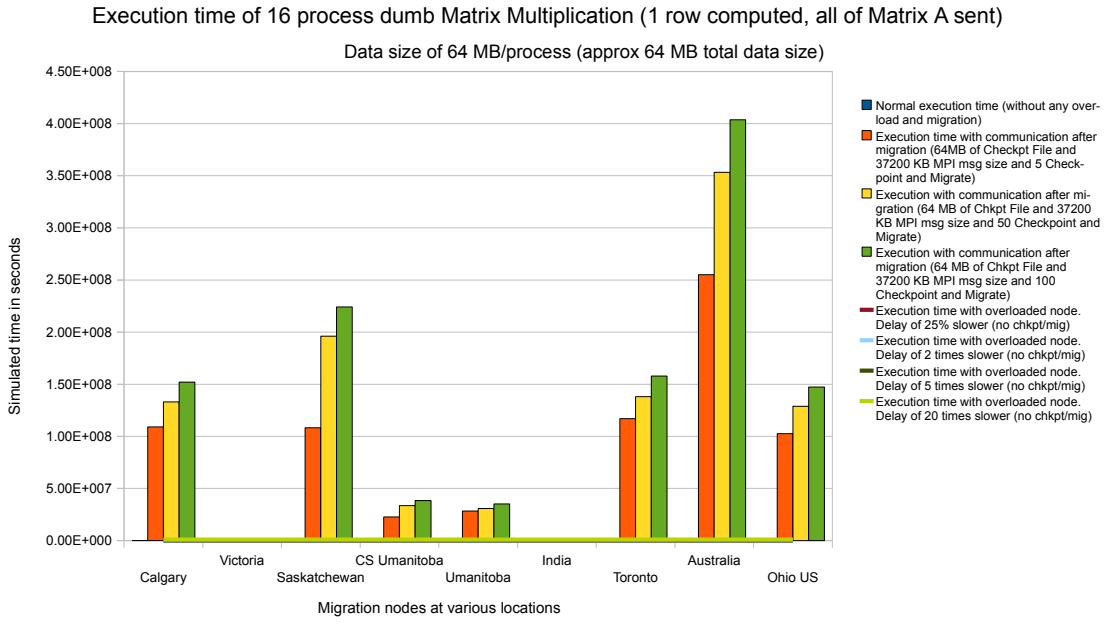

Figure 8.61: 16 Process Matrix Multiplication for 64 MB Problem Size

8.3 Usefulness of Partial Checkpoint/Migrate based on Experiment Results

From the results presented above and the analysis of the results for chosen applications and problem sizes, we can see that all the application types benefited by using my partial checkpoint/migrate system. In FEM and M/S when I decrease the number of processes to solve the same size problem, there is a performance deterioration, which suggests that with both small and larger number of processes in the execution environment it is beneficial to do a partial checkpoint/migrate of processes when the nodes are connected with significantly higher bandwidth networks, such as the LAN. cWAN and CWAN cases which benefited in all the cases. The WWAN nodes such as Australia and Ohio perform reasonably well in the case of *low* checkpoint/migrate frequency when the overload is *low*. This suggest broader applicability for partial checkpoint/migrate than might be expected but interpretation of the results must be tempered by the limitations of my experiments. Further, the trends in M/S and FFT are quite similar because of the fact that both the applications have quite regular but not overly aggressive communication patterns. FEM, on the other hand, as explained in Section 8.1.6 in detail, has frequent communication and computation events happening in parallel among separate groups of processes which causes the recording of communication events in an irregular fashion. This irregularity in occurrences of patterns of communication events could lead to sub-optimal predictions and hence increased after migration communication. But the results for the FEM application show that with an increase in the number of processes in the execution environment,

my prediction algorithms made good predictions and the application (and similar ones) should be able to benefit by using my partial checkpoint/migrate system for at least some scenarios.

The final experiments that I performed to evaluate the effects of communication intensity on partial checkpoint/migrate suggest that if the application is heavily communication oriented then, with the increase in the data being communicated the performance of the application deteriorates significantly when using partial checkpoint/migrate. This is not surprising.

The overall experimental results suggest that my Partial Checkpoint/Migrate system should be generally useful for the three types of applications tested which have regular communication patterns induced by programming style (e.g. Master/Slave), by data (e.g. Finite Element Method) and by algorithm (e.g. Fast Fourier Transformation). Highly irregular algorithms are, naturally, unlikely to benefit.

8.4 Overhead of Grouping and Prediction

As described in Chapter 7, Section 7.4.1, the worst case running time of TEIRESIAS is exponential in the number of patterns it generates. My implementation of TEIRESIAS works on a single sequence and discovers the patterns of communication occurring within that sequence. Regular applications with frequently recurring patterns should therefore be efficient. Algorithms with rapidly changing patterns, will incur higher overhead. My collection of communication data is done in sampling periods resulting in small sequences of communication information rather than just one very large sequence of communication information holding all the communication

information from the application. Also, the frequency of sampling and hence the overhead of doing pattern prediction can be adjusted. (Dynamic tuning of sampling frequency would be an interesting area for future work.)

My system is designed for long running applications with regular and repeating communication patterns. In such long running application I expect the communication patterns to be steady for long periods of time (relative to the sampling period). Having such steady patterns of communication in sampling periods definitely reduces the number of patterns generated by TEIRESIAS and hence reasonable performance can be expected. I also expect the running time of the KMP algorithm to be reasonable because of the fact that I only run KMP on the TEIRESIAS summarized pattern information rather than on the much larger raw collected communication information of sampling periods.

Chapter 9

Conclusions and Future Work

I have designed, implemented and tested a system for *partial* checkpointing in LAM/MPI. In particular, I have described the key aspects of my implementation of a prototype system which correctly supports the checkpointing and restart of a subset of the processes in a running MPI application while the other processes continue to run. My prototype also gathers information about inter-process communications and uses it as a basis upon which to determine which processes are communicating frequently so they may be grouped for checkpoint and migration.

I have also developed a family of predictors that can be used to group frequently communicating MPI processes for checkpointing and co-migration to improve their performance in a shared computing environment. My predictors are based on inter-process communication patterns discovered by the TEIRESIAS pattern discovery algorithm run against a sequence of pair-wise communication events captured by my LAM/MPI code enhancements. My initial results using synthetic data that mimics known communication patterns were very positive. Somewhat surprisingly, I saw rel-

atively little difference between the various algorithms for different application types. This suggests that it may be possible to create a reasonably good "default" predictor that could be used where little is known about an application's communications characteristics or where novice programmers cannot pick a specific predictor. Even if the consistent performance is partially an artifact of the synthetic data, it seems possible that a range of message passing applications can probably be supported by only a few predictors.

After having tested my predictors using the synthetic data I created, I collected actual inter-process communication data and used this to verify the results obtained and to increase my confidence in the ability of my system to handle the small changes in communications behaviour that occur in real programs. Using a modest but useful set of test applications I also identified optimizations that better suit particular application communications characteristics. I explored this in an attempt to ensure that consistent prediction accuracy can be provided and found that, with some minor algorithm modifications, for my tested applications consistent accuracy was obtained.

While TEIRESIAS outperforms other partial match algorithms using its "convolution" approach [RF98], its worst-case running time is still exponential for complex patterns with many variations. Therefore, it is not practical to run TEIRESIAS too frequently. Further, my predictor performance is sensitive to the size of the sampling periods. For both the synthetic data and real-world data used and for dozens up to hundreds of processes, my predictions are accurate and my predictor runtimes were all sub-second. As the number of communication events increases so too does the runtime. So this may have to be considered for applications with aggressive com-

munications. I have attempted to balance computational overhead and prediction accuracy and did experiments to determine how few pattern discoveries will suffice before missing important changes in inter-process communications for different types of applications (ultimately resulting in my use of the fundamental pattern approach). I also created a mechanism to adapt the sampling period dynamically for applications such as FEM where inaccuracies might otherwise occur.

Due to the non-deployable state of my software, I simulated the entire system using a simulation system written in SSJ (Stochastic Simulation in Java). My simulation uses the real world data that I collected from running applications and extrapolates it as needed. The useful patterns are summarized using the TEIRESIAS algorithm and the simulation uses the timestamp information and the raw data collected to simulate the execution of applications for different network and loading situations. The summarized data is used to make predictions when a simulated checkpoint request arrives.

I assessed the usefulness of my system by simulating various network conditions and assessing the cost of migration of the parts of applications to various locations having various network delays. I also assessed the impact of frequency of node overloading on my technique. I found that my system is practical to improve the execution time of long running LAM/MPI applications in the presence of a moderate level of node overload for the available applications in various networks.

I plan to explore several directions for future work. First, and foremost, I will expand the assessment of my system already done to better characterize the effectiveness of my approach. In particular, I will introduce baseline measurements (e.g.

moving a single MPI application process from an overloaded node and moving all MPI application processes at once) to more clearly indicate the performance benefit of partial checkpoint and migration. Further, the system that I developed is not fully deployable. Thus, a reasonable second goal in the future would be to configure the required environment to deploy a fully working implementation of the system. After the deployment of my system there are several things that can be done. For collecting communication characteristics I ran the MPI application on a single node by simulating a parallel environment, which limited the size of the problem that I could run to collect the communication information. Further, to get the communication data for a larger size problem I extrapolated the data collected from the smaller problem size to reflect the data for larger problem size. Once the system is deployed on a cluster of nodes, I can actually run the larger problem sizes and collect the communication information to verify the results that I got using the extrapolated communication information.

I also want to integrate my partial checkpoint/migrate prototype with the grouping mechanism I developed to actually perform a real runtime test for applications and verify the correctness of the groupings and study the effect of partially checkpointing, migrating and restarting the MPI application in a real execution environment.

To study the cost of migration and improve the system two possible general mechanisms for the actual transfer of checkpoints to the new location could be used, fast transfer and lazy transfer. The purpose behind using two mechanisms would be to achieve fast migration of jobs at reasonable cost (in terms of network overhead). For example, on a given machine running four processes the system might determine that

one of the processes which is heavily communicating with processes running on different machines is performing poorly due to unexpected load on the machine. While the other three local processes communicate only sporadically with the process which is performing poorly. It then makes sense to migrate the poorly performing task first and eventually (lazily) migrate the remaining three processes later. For the poorly performing process the checkpoint data should be sent immediately via the fast transfer mechanism, because the poorly performing process needs to be up and running again at the new location as soon as possible so it does not adversely affect the other processes (running on different machines) which may be waiting to communicate with it. The checkpoint and migration of the other three processes could then be done at an opportune time using the cheaper lazy approach making for more efficient use of the network resources.

In the long-term, I also would like to explore tools from grids, such as a resource management system (resource discovery and application manager) to incorporate with my system to automate the entire process of selection of alternate compute nodes and partial checkpoint/migrate.

Finally, my system has currently been tested only for a few, key MPI application types. As future work I plan to test more application types (based on well known parallel programming design patterns) and, if needed, come up with support for various different communication patterns in my prediction algorithms.

Appendix A

MPI Reference

A.1 MPI Overview

The typical organization of an MPI program is shown in FigureA.1. The call to MPI_Init must be the first MPI routine call any MPI program makes and $MPI\_Finalize$ must be the last call that any MPI program makes. An MPI program that does not reach an $MPI\_Finalize$ call is considered to be hung and the MPI runtime environment is not properly "cleaned". MPI_Init basically sets up the runtime environment for execution of the MPI application using the underlying LAM virtual machine consisting of all the nodes in the cluster. It basically invokes remote procedure calls to load the program executable into the memory of the remote/local machines. Once the program executable is loaded, an MPI "communicator" is setup and the ranks of the processes are allotted. A programmer can then write the code that the application as a whole or parts of the application must execute. A very simple example is shown in FigureA.1 where the if statement compares the *taskid* against a value *blah*.

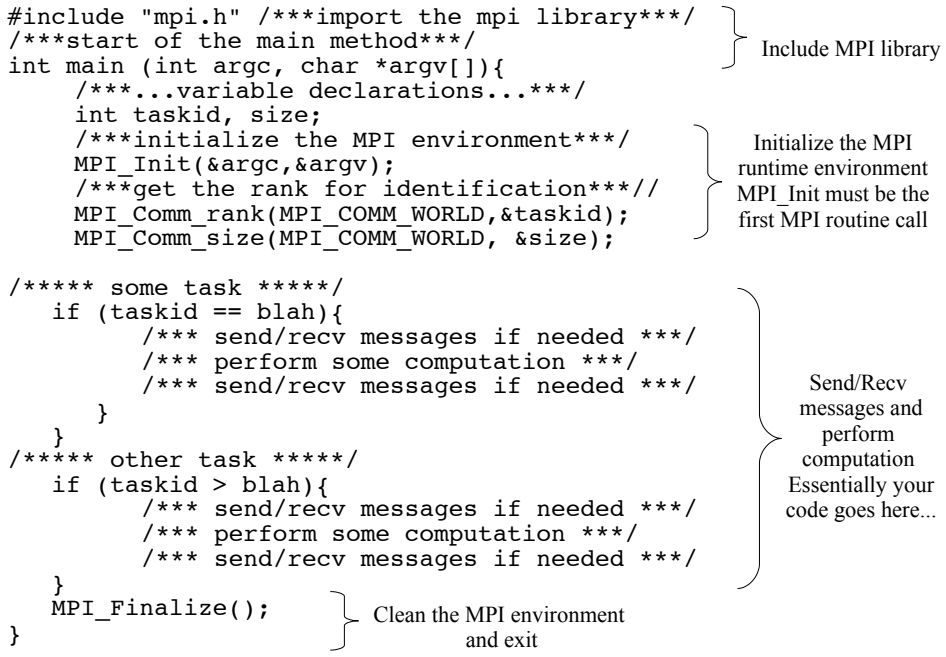

Figure A.1: MPI Program Organization

Depending on whether the *taskid* is equal or greater than *blah* the corresponding ranked processes will execute their designated part of the code. MPI processes can communicate with one another using either "point-to-point" (one process to another) or "collective" (groups of processes) communication operations.

A.1.1 Point-to-point communication

Some common and relevant MPI point-to-point communication routines are summarized here for the readers' reference:

MPI_Send():

*int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,*

MPI_Comm comm)

MPI_Send() is a blocking send. A process sends the message in 'buf' to another process with rank 'dest'.

MPI_Recv():

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status*)*

MPI_Recv() is a blocking receive. A process receives a message from another process of rank 'source'.

A.1.2 Collective communication

Some common and relevant MPI collective communication routines are now summarized as well:

MPI_Bcast():

*int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm*)

This routine broadcasts a common piece of data to all process ranks.

MPI_Scatter():

*int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, introot, MPI_Comm comm*)

If a, b, c, d is the data sent from the process with rank 1, then if there are 4 processes with ranks 0, 1, 2, 3 the data will be scattered so that process 0 receives a, 1 receives

b, 2 receives c and 3 receives d.

MPI_Gather():

*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*

The "root" process gathers values from each of the process ranks in the 'sendbuf' buffer.

MPI_Allgather():

*int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

Each process rank gets values from all the process ranks in the 'sendbuf' buffer. If values a, b, c, d are in ranks 0, 1, 2, 3 send buffers, respectively, then after MPI_Allgather all the processes (with ranks 0, 1, 2, 3) will have all the values a, b, c, d.

MPI_Alltoall():

*int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)*

If a, b, c, d is at rank 0, e, f, g, h is at rank 1, i, j, k, l is at rank 2 and m, n, o, p is at rank 3 then MPI_Alltoall will result in a, e, i, m being sent to rank 0, b, f, j, n being sent to rank 1, c, g, k, o being sent to rank 2 and d, h, l, p being sent to rank 3.

Again, there are some additional types of collective communications routines but they are not discussed here.

A.2 MPI Communicators

A communicator in MPI is a channel for communication between multiple processes in a group. A communicator consist of a group and a context. Processes are collected into groups associated with a communicator and given unique ranks in the group (By default all processes also belong to group/communicator 'MPI_COMM_WORLD'). Every communicator has a unique context and processes belonging to a group with in a communicator communicate with each other in the same context. Multiple communicators can be created in MPI applications to isolate communication between processes belonging to one group from processes belonging to another group. There are two types of communicators that can be created in an MPI application: intra-communicators and inter-communicators. An intra-communicator is used by the processes of a group to perform collective communication, whereas, an inter-communicator is used to exchange messages between processes belonging to separate intra-communicators.

MPI communicator routines:

MPI_Comm_create():

$int\ MPI\_Comm\_create(MPI\_Comm\ comm, MPI\_Group\ group, MPI\_Comm\ *new\_comm)$

creates a new communicator where, '*new_comm*' is the new communicator with a context and *group* is a sub-set of processes belonging to an existing communicator, '*comm*'.

MPI_Comm_split():

$int\ MPI\_Comm\_split(MPI\_Comm\ comm, int\ color, int\ key, MPI\_Comm\ *comm\_out)$

partitions a group of processes which are associated to the communicator '*comm*' into separate sub-groups each with a separate value of '*color*' (same color marked processes belong to one sub-group). '*key*' is used to define the rank of the processes belonging to one sub-group.

While my system can support the use of communicators, for simplicity, in this thesis, only applications using 'MPI_COMM_WORLD' are considered.

Appendix B

TEIRESIAS in More Detail

B.1 The TEIRESIAS Algorithm

As described earlier in Section 7.4.1, TEIRESIAS works in two phases. I now describe these phases in greater detail. The first phase is the scanning phase, the second phase actually consists of two parts, pre-convolution and convolution. Before discussing the working of TEIRESIAS we need some definitions (taken from [RF98]).

- Σ: is a set of residues or alphabet (example a set of all amino acids or nucleotides).
- Σ^*: is a set of strings that are constructed using the symbols in alphabet.
- $L(P)$: is a language defined by a pattern P is a set of all strings that can be obtained from P. This set of strings is obtained by substituting each don't care by a residue from alphabet.

- Pattern P: any string that begins and ends with a residue and contains an arbitrary combinations of residues and "don't care's" (represented by "." (dots)).
- $< L, W >$ pattern: A pattern P is an $< L, W >$ pattern if $L \leq W$ and every sub-pattern P with length $\geq W$ consists of at least L residues.
- Sub-pattern of P: Any substring of pattern P which is a pattern itself is called as sub-pattern of P.
- S: is a set of sequences (e.g. $s_1, s_2, \ldots, s_n$).
- s: is a sequence and an element of Σ^*.
- Offset list of P: For a pattern P and a set of sequences S the offset list of pattern P is defined as $Ls(P) = \{(i, j) | \textit{sequence } s_i \textit{ matches } P \textit{ at offset } j\}$.
- *maximal pattern*: for a set of sequences S, a pattern P is said to be *maximal* with respect to S if there is no pattern P' that is *more specific* (see next definition) than P such that $|Ls(P)| = |Ls(P')|$. Here, cardinality refers to the number of sequences containing pattern P or the count of the elements in offset list of pattern P.
- *more specific*: A pattern P' is more specific than pattern P if it can be obtained from P by changing one or more "."s to residues or by appending an arbitrary string of "."s and residues to either side of pattern P. Also, $|Ls(P')| \leq |Ls(P)|$.

The scanning phase of TEIRESIAS constructs an EP (Elementary Pattern) List, which has all the <L,W> patterns consisting of exactly L residues. The support[1] for

[1]Support of any pattern P is the number of occurrences of P in the given set of sequence(s).

each elementary pattern is at least K (i.e. each elementary pattern occurs in at least K input sequences or at least K times if the set consists of a single sequence).

Scanning Phase

```
EP = null
for all σ ∈ Σ
    if support (σ) ≥ K then
        Extend (σ)
    end-if
end-for

Extend (Pattern P)                          (1)
offset list counts[|Σ|]

A = # of regular characters in P
if A = L then
    Add P to EP                             (2)
    return
end-if
for i = 0 to (W - |P| - L + A)              (3)
    for all σ ∈ Σ                           (4)
        counts[σ] = empty
    end-for

P' = P concatenated with i dots
for (x,y) ∈ L_S (P)                         (5)
    if (y + |P| + i) < |s_x| then
        σ = s_x[y + |P| + i]
        Add (x,y + |P| + i) to counts[σ]
    end-if
end-for
for all σ ∈ Σ
    if support (P' σ) ≥ K then
        Extend (P' σ)
    end-if
end-for
end-for
```

Figure B.1: Scanning Phase (adopted from [RF98])

The Scan phase (shown in Figure B.1, for declarations of variable and functions refer to Figure B.2) starts by picking up each σ belonging to the alphabet Σ and trying to find occurrences of σ in the sequences. If the support for σ is less than K, then σ is ignored. Otherwise, σ is extended (by suffixing each residue from the alphabet in each step as long as the extended pattern is still found in at least K sequences) by calling the *Extend* routine at line (1) in Figure B.1 until it can no longer be extended. The extension results in an elementary pattern that has exactly L residues and whose maximum length is W. The *Extend* function basically takes a residue, σ, and tries all the possible suffixes including dots[2] and residues and checks if the support for the new string (generated by adding suffixes to σ) is at least K. If

[2] "dots" represent any symbols ("don't cares").

the support is K or more, the resulting string is again extended until it can not be extended further, thus creating an elementary pattern.

An example of the results obtained from the Scanning Phase is shown in Figure B.3(A). Each EP is associated with an Offset List, which denotes the position of the *EP* in a particular sequence. For a working example of the scanning phase please refer to Section B.2.

$S = \{ s_1, s_2, \ldots, s_n \}$: set of input sequences
$s_i[j]$: the j-th character in the i-the sequence
EP : set of elementary patterns
DirP(w) : subset of *EP* containing all the elementary patterns starting with w
DirS(w) : subset of *EP* containing all the elementary patterns ending in w
counts[i] : Offset list corresponding to the i-the character in Σ (Σ is arbitrarily ordered)
Maximal : set of maximal patterns
IsMaximal(R) : returns 0 if R is subsumed by a pattern in *Maximal*, and 1 otherwise

Figure B.2: Declaration in TEIRESIAS (adopted from [RF98])

The pre-convolution phase is where, preparations are made and data-structures built to speed up the convolution phase. The TEIRESIAS algorithm is designed so that, in the convolution phase, the *maximal* patterns are generated first before the non-maximal patterns that are subsumed by the *maximal* patterns. To accomplish this the pre-convolution phase is used, which essentially orders the *EP* List in a *prefix-wise* less ($<_{pf}$) order and also creates two orderings for each EP of those *patterns* that can be convolved with it on the *left* and *right* sides, respectively. The definitions of *prefix-wise* less ($<_{pf}$) and *suffix-wise* less ($<_{sf}$) are shown in Figure B.3(B), as it is easier understood with the accompanying example. The *left* convolution and *right* convolution lists for each *EP* are stored in lists labelled *Left* and *Right*. The patterns in *Left* for an *EP* are in ($<_{sf}$) ordering and the patterns in *Right* are in ($<_{pf}$) ordering. An example of such an ordering is shown in Figure B.4 where the

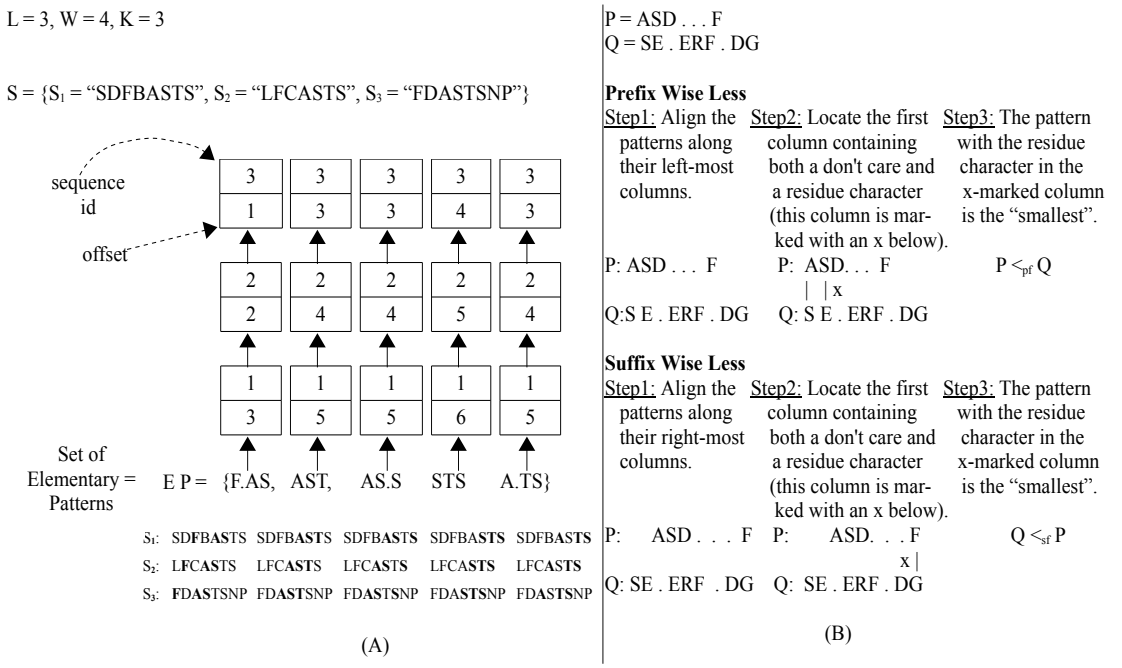

Figure B.3: (A) Example of Scanning Phase Results. (B) Definitions of Prefix Wise Less and Suffix Wise Less Orderings used in Pre-Convolution Phase (adopted from [RF98])

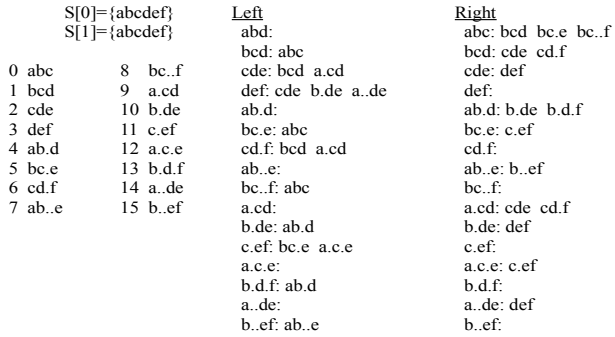

Figure B.4: (A) EP List sorted in ($<_{pf}$) ordering, S[0], S[1] are two sequences (B) $Left$ convolution list for EP List sorted in ($<_{sf}$) for each EP (C) $Right$ convolution list for EP List sorted in ($<_{pf}$) for each EP (adopted from [Wol99])

EPs from 0 to 15 are ordered in *prefix-wise* less ($<_{pf}$) order. The way each EP is ordered is to ensure that the don't care's are to the right as far as possible. Given patterns P and Q, aligned one below the other and starting from the leftmost letter,

when we find a position where one pattern contains a residue and the other contains a "." (don't care) then the one that has the specific residue is *prefix-wise* less ($<_{pf}$) than the other. The Lists *Left* and *Right* as shown in Figure B.4 for each *EP* are in ($<_{sf}$) and ($<_{pf}$) order , respectively, so that if L=3 then exactly $L-1$ residues on the right in the *Left* list should match for convolution with the corresponding *EP*s $L-1$ left residues and exactly $L-1$ residues on the left in the *Right* list should match for convolution with the corresponding *EP*s $L-1$ right residues. In Figure B.4, in the list *Left*, consider the $3rd$ *EP*, *cde* : *bcd a.cd*, for L=3 we have $L-1$=2. Thus, *bcd* is left convolvable with *EP cde* because the rightmost two residues in *bcd* are *cd* and they match with the leftmost 2 residues of *EP cde*. Similarly, in the list *Right* consider the $3rd$ *EP*, *cde* : def, for L=3 we have $L-1$=2. Thus, def is right convolvable with *EP cde* because the left most 2 residues in def are *de* and they match with the right most 2 residues of *EP cde*.

The Convolution Phase is the last phase in TEIRESIAS and it generates the *maximal* patterns. The scanning phase builds the *EP*s (which are small patterns with required support) and the convolution phase glues them together to create maximal patterns. Two patterns P and Q are convolvable with a resulting pattern, R, according to the following definition:

$$R = P \oplus Q = \begin{cases} PQ' & \text{if suffix(P) = prefix(Q)} \\ \emptyset & \text{otherwise} \end{cases}$$

where Q' is a string such that Q =*prefix*$(Q)Q'$. Further, *prefix*(P) is defined as a subpattern of pattern P that has exactly $(L-1)$ residues in it which is also a prefix of P. For example for $L=3$, *prefix*("$F.ASTA$") is "$F.A$" and *prefix*("AST") is "AS".

Convolution Phase

```
Order EP according to prefix-wise less and let
all entries in EP be unmarked.
Build DirS and DirP.
Maximal = empty
Clear stack
while EP not empty
   P = prefix-wise smallest unmarked element in
       EP (ties are resolved arbitrarily)
   mark P
   if IsMaximal(P) then
      push (P)
   else
      continue
   end-if
   while stack not empty
start:
      T = top of stack
      w = prefix (T)
      U = { Q ∈ DirS (w) | Q, T have not
            been convolved yet }
      while U not empty
         Q = minimum element of U
            ( according to suffix-wise less )      (5)
         R = Q ⊕ T                                 (6)
         if |Ls (R)| = |Ls (T)| then
            pop stack
         end-if
         if support (R) ≥ K and IsMaximal (R)
            push (R)
            goto start
         end-if
      end-while
      w = suffix (T)
      U = { Q ∈ DirP (w) | Q, T have not
            been convolved yet }
      while U not empty
         Q = minimum element of U
            ( according to prefix-wise less )      (7)
         R = T ⊕ Q                                 (8)
         if |Ls (R)| = |Ls (T)| then
            pop stack
         end-if
         if support (R) ≥ K and IsMaximal (R)
            push (R)
            goto start
         end-if
      end-while
      T = pop stack
      Add T in Maximal
      Report T
   end-while
end-while
```

Figure B.5: Convolution Phase (adopted from [RF98])

Similarly, *suffix*(P) contains exactly $(L-1)$ residues and is a suffix of pattern p. For example for $L = 3$, *suffix*("$F.A...S$") is "$A...S$" *suffix*("$AST S$") is "TS". After the patterns are convolved using the EP List and the *Left* and *Right* convolution lists for each EPs the resulting pattern, R's, offset list is a subset of the offset list of P and is obtained as follows:

$$L_s(R) = \{(i,j) \in L_s(P) | \exists (i,k) \in L_s(Q)$$
$$\text{such that } k - j = |P| - |suffix(P)|\}$$

where, $L_s(P)$ is the offset list of pattern P, $|P|$ is the cardinality of P (only residues are counted).

The Convolution algorithm is shown in Figure B.5. The ordered prefix-wise less *EP* list is taken from the pre-convolution phase along with the *Left* and *Right* lists shown in Figure B.5 as $DirS$ and $DirP$ respectively. The algorithm uses a stack to record the *maximal* patterns found. The left convolution is shown around lines (5) and (6), and the right convolution is shown around lines (7) and (8).

For a working example of convolution phase please refer to Section B.3.

B.2 Scanning Phase Example

Consider a very simple example of 3 sequences STS, STS and STS, with $L = 2$, $W = 3$ and $K = 2$. The scanning phase would be as follows:

σ=S
$sup(S) \geq 2 \rightarrow$ **yes**
$Extend(S)$
$A == L \rightarrow 1 == 2 \rightarrow$ **no**
for $i = 0$ to 1
$P' = SS$ and ST
for $(x, y) \in Ls(S) \rightarrow$ **(1,1)** $\in Ls(S)$
if $2 < 3 \rightarrow$ **yes**
$\sigma = S_1[2] = T$
Add $(1, 1)$ to offset list of ST
...
...
...
For ST two more positions are added to offset list $(2, 1)$ and $(3, 1)$
for all σ
if $sup(P') \geq K \rightarrow sup(ST) \geq K \rightarrow 3 \geq 2 \rightarrow$ **yes**
$Extend(ST)$
$A == L \rightarrow 2 == 2 \rightarrow$ **yes**
Add to EP so the Elementary Pattern is added to EP
Next σ=T is extended and ...

B.3 Convolution Phase Example

Now, consider two sequences $\{abcdef\}$ and $\{abcdef\}$, L=3, W=5 and K=2. (This example is adopted from [Wol99].) The scanning phase gives the following 16 EPs with their offset list L:

$abc \rightarrow$ offset: $\{(1,1),(2,1)\}$
$ab.d \rightarrow$ offset: $\{(1,1),(2,1)\}$
$ab..e \rightarrow$ offset: $\{(1,1),(2,1)\}$
$a.cd \rightarrow$ offset: $\{(1,1),(2,1)\}$
$a.c.e \rightarrow$ offset: $\{(1,1),(2,1)\}$
$a..de \rightarrow$ offset: $\{(1,1),(2,1)\}$
$bcd \rightarrow$ offset: $\{(1,2),(2,2)\}$
$bc.e \rightarrow$ offset: $\{(1,2),(2,2)\}$
$bc..f \rightarrow$ offset: $\{(1,2),(2,2)\}$
$b.de \rightarrow$ offset: $\{(1,2),(2,2)\}$
$b.d.f \rightarrow$ offset: $\{(1,2),(2,2)\}$
$b..ef \rightarrow$ offset: $\{(1,2),(2,2)\}$
$cde \rightarrow$ offset: $\{(1,3),(2,3)\}$
$cd.f \rightarrow$ offset: $\{(1,3),(2,3)\}$
$c.ef \rightarrow$ offset: $\{(1,3),(2,3)\}$
$def \rightarrow$ offset: $\{(1,4),(2,4)\}$

The EP (sorted in prefix-wise less order) and the $DirS$ (sorted in suffix-wise less) and $DirP$ (sorted in prefix-wise less) are shown in Figure B.6 as lists from 0 to 15, Left and Right, respectively.

$T = abc$
Left convolution:
0 left convolutions for abc
Right convolution:
3 right convolutions
w=bc
U=$\{bcd, bc.e, bc..f\}$
while U not empty
Q=bcd
R=$abc \oplus bcd =$ **abcd**

	S[0]={abcdef}	Left	Right
	S[1]={abcdef}	abd:	abc: bcd bc.e bc..f
		bcd: abc	bcd: cde cd.f
0 abc	8 bc..f	cde: bcd a.cd	cde: def
1 bcd	9 a.cd	def: cde b.de a..de	def:
2 cde	10 b.de	ab.d:	ab.d: b.de b.d.f
3 def	11 c.ef	bc.e: abc	bc.e: c.ef
4 ab.d	12 a.c.e	cd.f: bcd a.cd	cd.f:
5 bc.e	13 b.d.f	ab..e:	ab..e: b..ef
6 cd.f	14 a..de	bc..f: abc	bc..f:
7 ab..e	15 b..ef	a.cd:	a.cd: cde cd.f
		b.de: ab.d	b.de: def
		c.ef: bc.e a.c.e	c.ef:
		a.c.e:	a.c.e: c.ef
		b.d.f: ab.d	b.d.f:
		a..de:	a..de: def
		b..ef: ab..e	b..ef:

Figure B.6: (A) EP List sorted in ($<_{pf}$) ordering, S[0], S[1] are two sequences (B) $Left$ convolution list for EP List sorted in ($<_{sf}$) for each EP (C) $Right$ convolution list for EP List sorted in ($<_{pf}$) for each EP (adopted from [Wol99])

$Ls(R) \rightarrow \{(1,1),(1,2)\}$ and $Ls(T) \rightarrow \{(1,1),(2,1)\}$
if $(|Ls(R)|) = (|Ls(T)|) \rightarrow 2 = 2$
POP *abc*
if $support(R) \geq K$ and $isMaximal(R) \rightarrow 2 \geq 2$ and TRUE
$PUSH$ *abcd*
$T = abcd$
Left convolution:
0 left convolutions for *abcd*
Right convolution:
2 right convolutions
w=*cd*
U={*cde*, *cd.f*}
Q=*cde*
R=*abcd* $\bigoplus$ *cde* = **abcde**
$Ls(R) \rightarrow \{(1,1),(1,2)\}$ and $Ls(T) \rightarrow \{(1,1),(2,1)\}$
if $(Ls(R)) = (Ls(T)) \rightarrow 2 = 2$
POP *abcd*
if $support(R) \geq K$ and $isMaximal(R) \rightarrow 2 \geq 2$ and TRUE
$PUSH$ *abcde*
$T = abcde$
Left convolution:
0 left convolutions for *abcde*

Right convolution:
1 right convolution
w=de
U=$\{def\}$
Q=def
R=$abcde \bigoplus def$ = **abcdef**
$Ls(R) \rightarrow \{(1,1),(1,2)\}$ and $Ls(T) \rightarrow \{(1,1),(2,1)\}$
if $(Ls(R)) = (Ls(T)) \rightarrow 2 = 2$
POP $abcde$
if $support(R) \geq K$ and $isMaximal(R) \rightarrow 2 \geq 2$ and TRUE
$PUSH$ $abcdef$
    T=$abcdef$
all the rest are Subsumed.

Bibliography

[AAB+02] D. Angulo, R. Aydt, F. Berman, A. Chien, K. Cooper, H. Dail, J. Dongarra, I. Foster, D. gannon, L. Johnsson, K. Kennedy, C. Kesselman, M. Mazina, J. Mellor-Crummey, D. Reed, O. Sievert, L. Torczon, S. Vadhiyar, and R. Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[AAF+01] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications 15, 4 (2001), 345–358*, 2001.

[AF99] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamis MPI Programs on Clusters of Workstations. In *Proceedings of 8^{th} IEEE Int. Symp. on High Perf. Dist. Computing*, 1999.

[Aho01] J. Ahola. Mining Sequential Patterns. Research Report TTE1-2001-10, VTT Information Technology, Finland, May 21 2001.

[AO04] C. Antunes and A. L. Oliveria. Sequential Pattern Mining Algorithms: Trade-offs between Speed and Memory. In *Proc. of 2nd Intl. Workshop on Mining Graphs, Trees and Sequences (MGTS 2004),Pisa, Italy*, 2004.

[AP00] A. Agbaria and J. S. Plank. Design, Implementation, and Performance of Checkpointing in NetSolve. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN2000*, 2000.

[Bar10] Blaise Barney. Introduction to Parallel Computing. Available at: https://computing.llnl.gov/tutorials/parallel_comp/, March 2010.

[BCC+02] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High-Performance Computing Applications, 15(4)*, 2002.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen king Su. Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15:29–36, 1995.

[BHP03] S. Bouchenak, D. Hagimont, and N. D. Palma. Efficient Java Thread Serialization. In *2nd ACM International Conference on Principles and Practice of Programming on Java (ACM PPPJ'03), Kilkenny, Ireland*, June 2003.

[BLL92] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary, Version 4.1b. Technical report, Computer Science Department, University of Wisconsin-Madison, Wisconsin, USA, 1992.

[Boy10] R. D. Boyle. Tutorial on hidden markov models, May 2010.

[But97] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[BVL03] Brona Brejova, Tomas Vinar, and Ming Li. Pattern Discovery: Methods and Software. In *Stephen A. Krawetz, David D. Womble, ed., Introduction to Bioinformatics, chapter 29, Humana Press*, pages 491 – 522, 2003.

[CCG+95] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing. Technical report, 1995.

[CCK+95] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. Technical report, Technical Report CSE-95-002, 1, 1995.

[CD96] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science problems. In *Proceedings of Supercomputing'96, Pittsburgh*, 1996.

[CFK+02] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: A Protocol for Negotiating Service Level Agreements and Coor-

dinating Resource Management in Distributed Systems. In *8th Workshop on Job Scheduling Strategies for Parallel Processing, Edinburgh, Scotland*, July 2002.

[CJP07] Barbara Chapman, Gabriele Jost, and Rudd van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CN93] P. Coad and J. Nicola. *Object-Oriented Programming*. P T R Prentice Hall, 1993.

[DHR02] J. Duall, P. Hargrove, and E. Roman. The Design and Implementation of Berkley Lab's Linux Checkpoint/Restart. Technical report, Berkley Lab Technical Report LBNL-54941, 2002.

[DLVS94] L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot. Dynamic PVM – Dynamic Load Balancing on Parallel Systems. In *Proceedings Volume II: Networking and Tools, Springer-Verlag, Munich, Germany*, pages 273–277, 1994.

[EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Proformance of Consistent Checkpointing. In *Proceedings of 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[FB89] S. I. Feldman and C. B. Brown. Igor: a System for Program Debugging via Reversible Execution. In *Proceedings of ACM SIGPLAN Notices,*

Workshop on Parallel and Distributed Debugging, volume 24(1), pages 112–123, January 1989.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications, 11(2):115- 128*, 1997.

[FK99] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Fransisco, CA, 1999.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22 2002.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. In *International J. Supercomputer Applications, 15(3)*, 2001.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A Portable Instrumented Communication Library, C Reference Manual. Technical report, ORNL Technical Report ORNL/TM-11130, July 1990.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[Gol08] Martin Gollery. *Handbook of Hidden Markov Models in Bioinformatics*. CRC Press, Tahoe Informatics, Incline Village, Nevada, USA, June 12 2008.

[Hen02] Erik Hendriks. Bproc: The Beowulf Distributed Process Space. In *16th Annual ACM International Conference on Supercomputing*, 2002.

[Hol96] Jeffrey K. Hollingsworth. An Online Computation of Critical Path Profiling. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Philadelphia, Pennsylvania, United States*, pages 11 – 20, 1996.

[Inc03] Mellanox Technologies Inc. Introduction to InfiniBand. Technical Report 2003WP, Mellanox Technologies Inc., 2900 Stender Way, Santa Clara, CA, 2003.

[Jr.73] G. D. Forney Jr. The Viterbi Algorithm. In *Proceedings of the IEEE*, volume 61, March 1973.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KR00] S. Kalaiselvi and V. Rajaraman. A Survey of Checkpointing Algorithms for Parallel and Distributed Computers. *Sādhanā*, Vol. 25, Part 5, pp.:489–510, October 2000.

[Las02] A. Lastovetsky. Adaptive Parallel Computing on Heterogeneous Networks with mpC. *Parallel Computing*, 28:1369–1407, June 2002.

[LB05] P. L'Ecuyer and E. Buist. Simulation in Java with SSJ. In *WSC'05: Proceedings of the 37th Conference on Winter Simulations*, pages 611–620, Orlando, Florida, 2005. Winter Simulation Conference.

[Lev66] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics*, 10:707-710, 1966.

[Lit87] Michael J. Litzkow. Remote Unix: Turning Idle Workstations into Cycle Servers. In *Proceedings of the USENIX Summer Conference. USENIX*, June 1987.

[LLM88] M. Litzkow, M. Livny, and M.W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of 8^{th} International Conference on Distributed Computing Systems*, pages 104–111, june 1988.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, University of Wisconsin-Madison Computer Sciences Technical Report No. 1346, April 1997.

[ML87] M. W. Mutka and M. Livny. Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network. In *Proceedings of 7^{th} International Conference Distributed Computing Systems, Berlin, West Germany*, pages 2–9, 1987.

[MS99] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. In *In IEEE Transactions on Parallel and Distributed Systems, 10(7):703–713*, July 1999.

[MSRSM09] J. Mehnert-Spahn, T. Ropars, M. Schoettner, and Christine Morin. The Architecture of the XtreemOS Grid Checkpointing Service. In *Euro-Par*

2009, 15th International Euro-Par Conference, Proceedings, Delft, The Netherlands, pages 429–441, August 25–28 2009.

[MSSMM07] John Mehnert-Spahn, Michael Schoettner, David Margery, and Christine Morin. XtreemOS Grid Checkpointing Architecture. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2008)*, Lyon, France, May 2007.

[PBB+01] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing Data Storage in the Network. *IEEE Internet Computing, pp. 50-58*, 5(5), September/October 2001.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *In Proc. of Usenix Technical Conference*, 1995.

[PH01] J. Pei and J. et al Han. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Int'l Conf Data Engineering*, pages 215 – 226, 2001.

[PLP98] J. S. Plank, K. Li, and M. A. Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[Pos81] J. Postel. Internet Protocol. *RFC 792*, September 1981.

[RF98] I. Rigoutsos and A. Floratos. Combinatorial Pattern Discovery in Bi-

ological Sequences: The TEIRESIAS Algorithm. In *Bioinformatics, 14(1)*, pages 55 – 67, 1998.

[SA96] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Int'l Conf Extending Database Technology. Springer*, pages 3 – 17, 1996.

[SBL03a] J. M. Squyres, B. Barrett, and A. Lumsdaine. Technical Report TR576: Boot System Services Interface (SSI) Modules for LAM/MPI [API Version 1.0.0/SSI Version 1.0.0], Indiana University Department of Computer Science. July 2003.

[SBL03b] J. M. Squyres, B. Barrett, and A. Lumsdaine. Technical Report TR577: MPI Collective Operations System Services Interface (SSI) Modules for LAM/MPI [API Version 1.0.0/SSI Version 1.0.0], Indiana University Department of Computer Science. July 2003.

[SBL03c] J. M. Squyres, B. Barrett, and A. Lumsdaine. Technical Report TR578: Checkpoint/Restart System Services Interface (SSI) Modules for LAM/MPI [API Version 1.0.0/SSI Version 1.0.0], Indiana University Department of Computer Science. July 2003.

[SBL03d] J. M. Squyres, B. Barrett, and A. Lumsdaine. Technical Report TR579: Request Progression Interface (RPI) System Services Interface (SSI) Modules for LAM/MPI [API Version 1.0.0/SSI Version 1.0.0], Indiana University Department of Computer Science. July 2003.

[SG08] Rajendra Singh and Peter Graham. Performance Driven Partial Checkpoint/Migrate for LAM-MPI. In *22nd International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 110 – 116, June 2008.

[SH02] Frank Schmuck and Roger Haskin. "gpfs: A shared-disk file system for large computing clusters". In *Proceedings of the USENIX FAST'02 Conference on File and Storage Technologies*, pages 231–244, Monterey, California, USA, January 2002.

[Sin03] Rajendra Singh. The Dynamic Construction of Clusters from Idle Workstations Using Active Networks. Master's thesis, The Department of Computer Science, University of Manitoba, 2003.

[SL03] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings of the ACM SIGPLAN Symposium on Practice of Parallel Programming (PPoPP 2003), Vol. 38, Issue 10*, October 2003.

[SSB+03] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *Proceedings of LACSI Symposium*, October 2003.

[SSK05] N. Stone, D. Simmel, and T. Kielmann. An Architecture for Grid Checkpoint Recovery Services and a GridCPR API,. Technical report, September 2005.

[Ste96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience, vol. 2(4), pp. 315–339*, December 1990.

[Sys10] Veridian Systems. Portable Batch System, June 2010.

[VD03] S. S. Vadhiyar and J. J. Dongarra. A Performance Oriented Migration Framework for the Grid. In *3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan*, May 12 - 15 2003.

[Vit67] A. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. In *IEEE Transactions on Information Theory*, volume IT-13, pages 260–269, April 1967.

[WM89] P. R. Wilson and T. G. Moher. Demonic Memory for Process Histories. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989.

[WMES07] C. Wang, F. Mueller, C. Engelmann, and S.L. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Proceedings of the Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[Wol99] Murray Wolinsky. An Experimental Implementation of the TEIRESIAS

Algorithm. *Final Project for Computational Molecular Biology, Stanford University*, 1999.

[ZB03] Q. Zhao and S. S. Bhowmik. Sequential Pattern Mining: A Survey. Technical Report 2003118, Nanyang Technological University, 2003.

[ZN02] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical report, Technical Report CUCS-014-01. Department of Computer Science. Columbia University, November 2002.