Run-time parallelization of irregular DOACROSS loops

by

THULASIRAMAN JEYARAMAN

A thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

ISBN 0-612-13214-5

Canada

Name _____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

COMPOTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

RUN-TIME PARALLELIZATION OF IRREGULAR DOACROSS LOOPS

BY

THULASIRAMAN JEYARAMAN

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Run-time parallelization of irregular DOACROSS loops

Thulasiraman Jeyaraman

Master of Science, 1996

Department of Computer Science, University of Manitoba

Abstract

Dependencies between iterations of loop structures cannot always be determined at compile-time because they may depend on input data which is known only at run-time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, it is necessary to perform run-time analysis. We present a new algorithm to parallelize these loops at run-time. The proposed algorithm handles all types of data dependencies without requiring any special architectural support in the multiprocessor. Our scheme has an inspector which builds the iteration schedule and an executor which uses the schedule to execute the various iterations. Our approach does not require any special synchronization operations during the inspector stage and the executor can be implemented with or without synchronization support. It allows overlap among dependent iterations and requires very little inter-processor communication. Further, the schedule formed by the inspector can be reused across loop invocations.

We evaluate our algorithm with parameterized loops running on an 8-processor UMA (Uniform Memory Access) shared memory multiprocessor. The results show significant speedups over the serial code with the full overhead of run-time analysis. Furthermore, compared to an older scheme with the same generality, our scheme has consistent performance (i.e., performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and ensures good speedup during the executor stage. We have also evaluated the performance of the algorithm on a collection of networked heterogeneous workstations (distributed memory

model) using Parallel Virtual Machine (PVM). PVM is a widely-used software system that allows a heterogeneous set of parallel and serial UNIX-based computers to be programmed as a single distributed-memory parallel machine. It is portable and runs on a wide variety of modern platforms. We obtained improved performance on distributed nodes using our algorithm and the performance is scalable to a larger number of nodes given adequate network support.

Acknowledgments

First, I would like to thank my supervisor, Mark Giesbrecht for showing me the right directions in research; needless to mention his pleasing smile and motivating ideas. I would also like to thank Prasad Krothapalli, Informix Software Inc., CA for his ideas which were valuable contributions to the thesis work. I would also like to thank my thesis committee members Peter C. J. Graham (for his useful comments) and Qiang Ye, for their time and remarks in preparing me for the thesis defense. As well, I would like to thank Professor Derek L. Eager, Department of Computer Science, University of Saskatchewan, Saskatoon; and Professor P. Sadayappan, Department of Computer and Information Science, Ohio State University, Columbus, Ohio for their help in the experiments. I wish to thank Gilbert E. Detillieux and Tom Dubinski, for their support in the experiments. Finally, my thanks goes to my parents and family for their emotional support and encouragement all through.

Contents

List of Figures

Chapter 1

Introduction

To exploit parallelism hidden in loop structures, the various dependencies between loop iterations have to be analyzed, and the iterations have to be effectively scheduled without violating the inter-iteration dependencies, in order to exploit maximum parallelism. The dependencies possible between any two statements S1 and S2 in sequence can be classified as:

1. A *flow* dependency or true dependency, if the statement S2 uses a variable assigned by the statement S1.

2. An *anti-dependency*, when S2 assigns a variable subsequently used by the statement S1.

3. An *output-dependency*, when statements S1 and S2 assign the same variable.

Loop-level parallelism is an important source of speedup in multiprocessors that exploit medium-grain parallelism. To parallelize a loop, it is necessary to identify the data dependence relations between loop iterations via dependence analysis [2]. If the loop does not have any dependencies across iterations, then it can run in parallel. Otherwise, it has to be run serially or, if some parallelism is to be exploited, proper synchronization features have to be inserted in the code to ensure that the memory

```
/* Index arrays I1 and I2 contain values in the range 1 to N. There are M iterations
in the DOACROSS loop. Array A contains N elements. */
    do i = 1 to M
        ...
    S_p: A[I1[i]] = ...
        ...
    S_q: ... = A[I2[i]] + ...
        ...
    enddo
```

Figure 1.1: A generic case of an irregular DOACROSS loop.

accesses are performed in the correct sequence. The latter type of loop is called a DOACROSS loop.

Some DOACROSS loops can be transformed into fully parallel loops. For example, if the difference in iteration numbers between dependent iterations (called the dependence distance) is known at compile-time, techniques like loop-alignment, loop skewing, or dependence uniformization [19, 28, 4, 25] can be used to run the loop in parallel. When array elements are accessed via subscripted subscripts (as in Figure 1.1), however, it is not possible to resolve the dependencies between the iterations of the loop structure at compile-time, since the values of the subscripted subscripts are known only at run-time. In these cases, the compiler cannot prove the independence of iterations. Therefore, even if the loop turns out to be parallel in nature, compile-time schemes fail to parallelize the loop.

In the above loop, the entries in the index arrays *I1* and *I2* have no restriction and are not available until run-time. Each index array can have duplicated values, which implies that all three kinds of data dependencies, namely flow (Read-after-Write), anti- (Write-after-Read) and output- (Write-after-Write) dependencies, could occur between instances of statements S_p and S_q. For example, if $I1[1] = I2[3] = I1[4] =$

$I1[5] = x$, then we have the following dependencies: $S_p[1] \xrightarrow{flow} S_q[3] \xrightarrow{anti-} S_p[4] \xrightarrow{output-} S_p[5]$. These accesses form a *dependence (or access) chain.* The entries in the arrays *I1* and *I2* are assumed to remain constant during the execution of one invocation of the loop, although they are allowed to change outside the loop.

For loops not amenable to compile-time parallelization, we can still perform the dependence analysis at run-time and thereby execute the loop, or part of it, in parallel. This approach is called *run-time parallelization.* Much previous research has been done to design effective run-time parallelization algorithms [3, 10, 11, 12, 15, 16, 21, 29]. The main differences among the schemes proposed are the types of dependence patterns that are handled and the required system or architecture support (some schemes take advantage of special synchronization primitives available on specific machines). The key to success in these schemes is to minimize the time spent on dependency analysis and on process synchronization. Indeed, if too much time is spent on these overheads, it is better to run the loop serially. In general, run-time parallelization schemes have two stages, namely the *inspector* and the *executor* [29, 12, 21, 3]. The inspector determines the dependence relations among the data accesses. The executor uses this information to execute the iterations in parallel in an order that respects the dependences.

In this thesis, we describe and evaluate a new algorithm for the run-time parallelization of DOACROSS loops. We follow the inspector and executor approach. In our scheme, the *inspector* inspects the various dependencies existing between the iterations and constructs an iteration schedule. This involves reordering the iterations. The entire set of iterations is partitioned into subsets called wavefronts. A wavefront is a grouping of the iterations which can be executed concurrently. A specific wavefront contains a collection of iterations which have no dependencies among them. Each one of the iterations in a loop is assigned to a specific wavefront based on its po-

sition in the dependence hierarchy. The wavefronts are executed sequentially whereas the iterations contained in a wavefront are executed concurrently. Since the wavefront number *WF[i]* for iteration i is set to one plus the maximum of the wavefront numbers of the iterations upon which it depends, iteration i is assigned a wavefront number such that all the iterations which it may depend on are in wavefronts which will have completed execution by the time it is scheduled. All the loop iterations are assigned specific wavefront numbers and wavefronts are executed in sequence by wavefront number.

The *executor* is a transformed version of the source code loop structure, which carries out the actual execution of iterations according to the schedule produced by the inspector. It uses the wavefront information produced by the inspector to schedule the iterations. It starts executing the iterations clustered in the first wavefront concurrently by scheduling them on multiple processors. Once the first wavefront finishes, it does the same with the iterations clustered in the next wavefront and so on until it reaches the last wavefront. Thus, the iterations within a particular wavefront are executed concurrently whereas successive wavefronts are executed in a strictly serial fashion.

Our scheme handles all types of data dependencies without requiring any special architectural support. It does not require any special synchronization operation during the inspector stage and the executor can be implemented with or without synchronization support. It allows overlap among dependent iterations and requires very little inter-processor communication. The effectiveness of our scheme is evaluated via measurements of parameterized loops on an 8-processor UMA (Uniform Memory Access) shared memory multiprocessor. The results show significant speedups over the serial code with the full overhead of run-time analysis. Furthermore, compared to an older scheme with the same generality, our scheme has consistent performance

(ie., the performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and ensures good speedup during the executor stage. Some of this work has appeared in [13, 23].

We have also evaluated the performance of our algorithm on a networked cluster of heterogeneous workstations running PVM [24]. PVM [5] is a software system that allows a heterogeneous network of parallel and serial computers to be programmed as a single computational resource. This resource appears to the application programmer as a potentially large distributed-memory virtual computer. Such a system allows the computing power of widely available, general-purpose computer networks to be harnessed for parallel processing. With the rapid advances in workstation performance, such networks already provide a viable and affordable alternative to expensive special-purpose super-computers. PVM is portable and runs on a wide variety of modern platforms. It is the mainstay of the Heterogeneous Network Computing research project, a collaborative venture between the Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University.

PVM supplies functions to start tasks in parallel across the virtual machine, allows these tasks to send data between themselves, and coordinates their parallel execution. A wide variety of computers (including serial computers, vector computers, and even shared or distributed memory parallel computers) can make up the user's personal virtual machine. The most popular use of PVM is the utilization of a small number of high performance workstations to achieve performance comparable to that of a supercomputer (at significantly less cost). The rapid growth in the use of PVM is due to its portability across many different computers, its robustness and ease of installation, and its usage as a standard parallel programming interface. PVM has already out-lasted several parallel vendors.

In an MPP (Massively Parallel Processor), every processor is exactly like the others in capability, resources, software, and communication speed. This in not so on a network. The computers available on a network may be made by different vendors or have different compilers and may be potentially shared by many users. Indeed, when a programmer wishes to exploit a collection of networked computers, he may have to contend with several different types of heterogeneity: processor architecture, data format, computational speed, network bandwidth/latency, machine load and network load. PVM makes it convenient or, in some cases, transparent for the user to deal with these issues.

The PVM user writes his/her application as a collection of cooperating *tasks*. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

1.1 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present the previous research work done on run-time loop parallelization. Chapter 3 explains our new algorithm for run-time parallelization. Chapter 4 describes the performance of our algorithm on an SGI (UMA) shared memory machine. In Chapter 5, we evaluate the performance of the algorithm on the distributed memory model (using PVM). Chapter 6 concludes our study and discusses future research directions.

Chapter 2

Previous work

In this chapter, we trace some of the significant contributions to run-time parallelization of loop structures. Zhu and Yew [29] have proposed an algorithm which handles all types of inter-iteration dependencies. Their algorithm also has a fully parallel inspector and executor. However, its inspector cannot be reused across multiple invocations of the DOACROSS loop because the inspector and the executor are tightly coupled. As well, it requires many memory accesses, which reduces the overall speedup. Midkiff and Padua [16] improve Zhu and Yew's scheme by allowing concurrent reads to the same array element by several iterations. To do so, each data element requires more than one key field. However, their algorithm, like Zhu and Yew's, requires high levels of communication. Other less general schemes have also been proposed [20].

Krothapalli and Sadayappan [11, 10] suggest a method to remove anti- and output-dependencies by analyzing the reference patterns generated and by using extra copies of the variables. Their algorithm is parallel in nature but requires an atomic *Fetch&Add* operation. The additional storage space required in their algorithm is proportional to the number of references. Krothapalli and Sadayappan [12, 10] describe a technique called pre-synchronized scheduling for dynamically scheduling the iterations of a DOACROSS loop. They further show how to use their scheme on vector

processors. Their inspector requires the run-time generation of the Iteration Space Dependency Graph (ISDG) (which can be done in parallel) which introduces run-time overhead. Saltz *et al.*'s [21] approach does not take care of anti- and output-dependencies at the inspector stage.

Leung and Zoharjan [15] suggest two techniques to parallelize dependence analysis (wavefront computation) during the inspector stage. They are:

Sectioning:

The M iterations in a loop are partitioned into P sections, where P is the number of processors. Each section is allocated to one processor. Thus, the wavefront computation goes on in a parallel fashion. Then, the wavefronts computed in each section are grouped together and numbered sequentially. The shortcoming of sectioning is that it produces more wavefronts when compared to the sequential execution of the inspector. This does not affect the overall efficiency as long as the number of processors available for computation is far less than the number of iterations per wavefront. However, as the number of processors is sufficiently high to handle the iterations in a given wavefront, then sectioning is not preferred, because it produces more wavefronts which leads to reduced parallelism among the iterations.

Bootstrapping:

This aims to improve upon sectioning by reducing the total number of wavefronts generated. Sectioning is applied on the source loop and a valid schedule is obtained. Then the schedule obtained is applied to the inspector loop in a wavefront-by-wavefront fashion. This generates fewer wavefronts due to the optimization involved during the second phase. The second phase basically repositions or pulls the iterations up the dependence hierarchy and places them in appropriate wavefronts. In essence, the first phase using sectioning forms the sub-optimal dependence tree. The second phase optimizes it by relocating the iterations to their correct slots.

2.1 Chen, Torrellas and Yew's scheme

The recent work by Chen *et al.* [3] handles all types of dependencies at the inspector stage and the inspector results can be reused across multiple invocations of the same loop structure if the dependencies do not change from one invocation of the DOACROSS loop to another. During the inspector stage, all the dependence information is gathered and stored in a table called the *Ticket* table. The inspector stage is divided into two parts, namely the local and the global inspector phase.

During the local inspector phase, the entire iteration space is divided into different sections and distributed among the available processors. The number of sections is equal to the number of available processors. Each processor records a sequence number in the *Ticket* table for each one of the shared variable (iteration dependent) references in each iteration in its own section. However, it does not assign any sequence number to the first reference in any dependence chain in its section. The sequence number for the first reference in each dependence chain in each of the sections will be assigned during the global inspector phase. Furthermore, during the local inspector phase, each processor constructs a linked list called the *Unresolved list* with nodes corresponding to the *Ticket* table entries in its section which have not been assigned sequence numbers. Each node contains the index of the shared data structure element, the corresponding entry in the *Ticket table*, and a transition sequence number. The transition sequence number is a possible sequence number for the first reference to the same element of the shared data structure in the other following sections. During the global inspector phase, the unfilled entries in each section of the *Ticket* table are filled by their corresponding processors by communicating with other processors using the *Unresolved list* maintained by each one of the processors.

The executor makes use of the dependence information recorded in the *Ticket* table to execute each of the iterations. It makes use of a shared variable *Ready*, for

each of the iteration dependent accesses. During any stage of execution, if the value of the *Ready* variable corresponding to any of the iteration dependent accesses does not equal its corresponding *Ticket* table entry, then the particular iteration dependent access *busy-waits*. The executor can either be *pre-scheduled* or *self-scheduled*.

In their scheme, the synchronization operations during the inspector are implemented using *Cedar* synchronization primitives (like *Cedar_sync*), which give better performance than simple *lock* and *unlock* synchronization primitives. *Cedar_sync* is a powerful synchronization primitive that operates on variables composed of two fields, namely *key* and *data*. A *Cedar_sync* operation on address *target*, first locks target and then performs a test on *target.key*. If the test fails, then the operation aborts. Otherwise, it performs operations on *target.key* and *target.data* and then unlocks target. The whole operation (key check, data access and key update) is specified in one atomic instruction.

During the inspector stage, building the *Ticket* table is very expensive, because it requires inter-processor communication. Furthermore, during the executor stage accesses to iteration dependent variables *busy-wait*. Since the iteration space to processor mapping is not done with a global picture of the dependence hierarchy, it leads to longer *busy-waits* during the executor stage. As well, their algorithm serializes reads with input dependencies, which introduces pseudo-dependencies among truly concurrent iterations.

Chapter 3

A new algorithm for run-time parallelization

To exploit the latent parallelism in irregular DOACROSS loops (Figure 1.1), we follow the inspector and executor approach. During the inspector stage, shadow memory locations (variables) are created for each of the iteration dependent variables. The shadow variables record the iteration in which a particular iteration dependent variable was last read and/or updated. Using the shadowed information pertaining to each iteration dependent variable, the various dependencies between iterations are analyzed, and an appropriate wavefront number is allocated to each of the iterations. The proposed dependence analysis takes care of all types of data dependencies.

In the inspector algorithm shown in Figure 3.1, the arrays *LastW* and *LastR* serve as shadow memory variables for the shared array *A*. *LastW* records the iteration during which a particular shared array element *A[i]* was last modified. Similarly, *LastR* records the iteration during which a particular shared array element *A[i]* was last read. For each of the shared array references, *A[i]*, in a particular iteration, the algorithm determines the iterations in which it was last modified and last read. This is accomplished using the information (iteration numbers) recorded in the shadow arrays *LastW* and *LastR*. Then the algorithm assigns an appropriate wavefront number to

```
/* There are N elements in the arrays LastW and LastR. Index arrays I1 and I2
contain values in the range 1 to N. There are M iterations in the loop. There are M
elements in the array WF. L is an array (size of M) of pointers to linked lists. */
WF[:] = 0; L[:] = NULL; LastW[:] = 0; LastR[:] = 0;
do i = 1 to M
     mywf=0;
     if (LastW[I1[i]] ≠ 0) /* output- dependency */
          { mywf = max(WF[LastW[I1[i]]],mywf); }
     if (LastR[I1[i]] ≠ 0) /* anti- dependency */
          { mywf = max(WF[LastR[I1[i]]],mywf); }
     if (LastW[I2[i]] ≠ 0) /* flow dependency */
          { mywf = max(WF[LastW[I2[i]]],mywf); }
     WF[i] = mywf + 1; /* wavefront allocation */
     Add_node(L[WF[i]],i); /* place iteration i at appropriate level */
     LastW[I1[i]] = i; /* update the shadow variables */
     if (LastR[I2[i]] ≠ 0)
          if (WF[LastR[I2[i]]] < WF[i]) { LastR[I2[i]] = i; }
     else { LastR[I2[i]] = i; }
enddo
```

Figure 3.1: The Inspector algorithm.

the particular iteration. This way an iteration is allotted a wavefront number which is numerically greater than the wavefront numbers of the iterations upon which it depends. Further, the inspector builds an array L of linked lists, where $L[j]$ contains a list of the iterations in wavefront j. This is used during the executor stage to schedule iterations onto different processors. The linked list holds the iteration numbers based on their position in the dependence hierarchy. For example, an iteration with a wavefront number of *four* will be placed in the fourth level in the array of linked lists. In the worst case (i.e., serial loop) there will be M (the total number of iterations) levels.

The dependency information thus recorded in the form of wavefronts can be reused for several invocations of the DOACROSS loop, provided the dependencies do not change. The primary advantage of wavefront generation is that it leads to better load balance among processors because the wavefronts carry a global picture of the dependence hierarchy which leads to efficient scheduling of iterations onto processors during the executor stage. The inspector in the new scheme does not require any *atomic synchronization operations* or *barriers*.

In the inspector algorithm just presented, the entire wavefront computation is done sequentially, which defeats the very purpose of parallelization. To parallelize the wavefront computation, *sectioning* [15] is performed on the inspector algorithm.

In the resulting parallel inspector (Figure 3.2), the entire range of iterations is partitioned into consecutive sub-ranges, called *sections*. Each section is assigned to a different processor. Each processor computes a valid parallel schedule for the iterations in its section by applying the sequential inspector algorithm and ignoring any dependencies on iterations outside of its section. After all the processors have finished, we have a schedule for each section. Every such schedule, called a *sub-schedule*, is a mapping from the iterations in the corresponding section to the wavefronts of

```
/* Index arrays I1 and I2 contain values in the range 1 to N. There are M iterations
in the loop. There are N elements in the arrays LastW and LastR. Arrays LastR and
LastW are local to the procedure. There are M elements in the array WF. L is an
array (size of M) of pointers to linked lists. Piece is the size of each section. P is
the number of processors. MAXW[:] has P elements. WF[:], MAXW[:] and L[:] are
shared data structures. They are initialized to zero before the light-weight threads (one
thread per processor) are activated. Thread id values range from 0 to P-1. */

piece = M/P; Myid = get_myid(); /* get my thread ID */
start = Myid*piece; /* get my share */
if (Myid == (P-1)) { last = M; } else { last = start+piece; }
Max = 0; LastW[:] = 0; LastR[:] = 0;
do i = (start+1) to last
     mywf=0;
     if (LastW[I1[i]] ≠ 0) /* output- dependency */
          { mywf = max(WF[LastW[I1[i]]],mywf); }
     if (LastR[I1[i]] ≠ 0) /* anti- dependency */
          { mywf = max(WF[LastR[I1[i]]],mywf); }
     if (LastW[I2[i]] ≠ 0) /* flow dependency */
          { mywf = max(WF[LastW[I2[i]]],mywf); }
     WF[i] = mywf + 1; /* wavefront allocation */
     LastW[I1[i]] = i; /* update the shadow variables */
     if (LastR[I2[i]] ≠ 0)
          if (WF[LastR[I2[i]]] < WF[i]) { LastR[I2[i]] = i; }
     else { LastR[I2[i]] = i; }
     Max = max(Max,WF[i]); /* record largest wavefront in this section */
enddo
if (Myid == 0) { MAXW[Myid+1] = Max; }
else
     while(MAXW[Myid] == 0); /* Wait for previous section to complete */
     sum = MAXW[Myid]; /* Get largest wavefront upto previous section */
     do i = (start+1) to last
          { WF[i] = WF[i] + sum; } /* Update wavefront numbers in this section */
     MAXW[Myid+1] = sum + Max;
endif
do i = (start+1) to last /* place the iteration i at the appropriate level */
     { Add_node(L[WF[i]],i); }
```

Figure 3.2: The Parallel Inspector algorithm run on each processor.

that section. The overall schedule is formed by concatenating the sub-schedules in the order of the sections. This is done by making processor i communicate the maximum wavefront up to its section to the next processor $i+1$ in sequence until the final section. In the parallel inspector algorithm (Figure 3.2), the shared array $MAXW$ serves to communicate the largest wavefront up to a particular section to the next section.

Iterations	1	2	3	4	5	6	7	8	9
Access1 (write)	1	2	2	1	5	9	7	8	11
Access2 (read)	2	9	6	8	9	1	12	10	12
Wavefront No.	1	2	3	2	1	3	1	3	1

Table 3.1: Sample run of the inspector on the DOACROSS loop (Figure 1.1) with 9 iterations and 2 hot references per iteration.

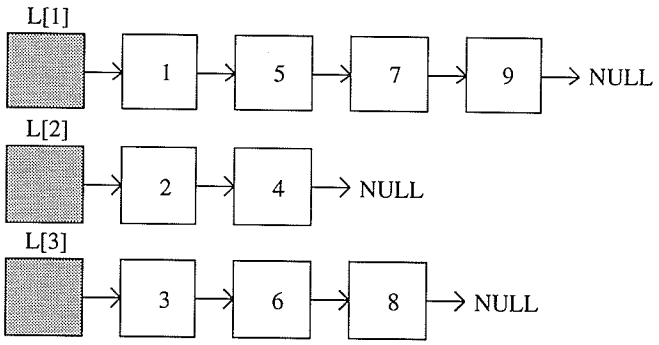

Figure 3.3: The array of linked lists generated during the sample run.

The assignment of iterations onto processors during the executor stage can be either *pre-scheduled* or *self-scheduled.* In *pre-scheduling*, iterations are mapped onto processors statically, usually in a cyclic fashion. In *self-scheduling*, the iteration to processor mapping is done during execution, where the processors dynamically pick the available iterations either one at a time or in chunks (*chunk scheduling*). To achieve better load balance among the processors, *Guided Self-Scheduling (GSS)* is

usually employed, where the free processors pick *(N/P)* of the remaining iterations, where N is the number of remaining iterations to be processed and P is the number of available processors.

Static or *pre-scheduling* of iterations does not lead to efficient iteration to processor mapping. Furthermore, it may lead to poor load balance among the processors. Since the position of iterations in the dependence hierarchy is known only at run-time, *self-scheduling* ensures efficient iteration space to processor mapping and better load balance among the processors.

Avoiding inter-wavefront barriers

In an executor implementation using *barrier sychronization* (Figure 3.4), a *barrier* condition has to be met whenever all the iterations in a particular wavefront have been scheduled. In other words, the processing of iterations corresponding to wavefront *i+1* should not begin before all the iterations corresponding to wavefront i have been completed, because an iteration in wavefront *i+1* may depend on some of the iterations in wavefront i. *The number of barriers required is proportional to the total number of wavefronts.* This obviously reduces the processor utilization as processors tend to idle at the barrier, waiting for other processors to complete. One way to get over the problem of under-utilization of processors is to avoid barriers between wavefronts and to schedule iterations onto free processors irrespective of the wavefront number. But this may conflict with the dependence hierarchy. To preserve the dependence structure, a *busy-wait* mechanism may be employed, where the free processors are allowed to process iterations corresponding to different wavefronts, but with the constraint that all accesses to iteration dependent variables *busy-wait.*

In the *busy-wait* executor implementation shown in Figure 3.6, iterations corre-

```
/* T is a shared pointer to a node in the array of linked lists L. Level is a shared
variable holding the current wavefront level. T = NULL and Level = 0 initially. LP
is a local pointer to a node in the array of linked lists L. Thread id values range from
0 to P-1. */

Myid = get_myid(); /* get my thread id */
while(1)
    lock(); /* prevents all other accesses to shared memory */
    if (T) T = T→next; /* gets next iteration in current level */
    else /* go to the next level */
        if ((Level+1) == (M+1))
            { unlock(); return; } /* all levels completed */
        if (L[Level+1] == NULL)
            { unlock(); return; } /* all iterations completed */
        unlock(); barrier(); /* synchronization point */
        if(Myid == 0)
            { lock(); Level = Level+1; T = L[Level]; } /* go to next level */
        barrier(); /* to prevent race condition */
        if (Myid ≠ 0) continue;
    endif
    if (T == NULL) { unlock(); continue; }
    LP = T; /* pick up the iteration to be processed */
    unlock();
    ...
    A[I1[LP→iteration]] = ... /* process the iteration */
    ...
    ... = A[I2[LP→iteration]] + ...
    ...
endwhile
```

Figure 3.4: *Self-scheduled* executor with *barrier synchronization.*

sponding to different wavefronts can be scheduled onto processors simultaneously, but accesses to iteration dependent variables *busy-wait* until all previous accesses to those variables have been completed. This involves maintaining a shared variable *Ready*, for each of the iteration dependent variables to enforce busy-waiting. The *busy-wait* mechanism improves the load-balance among the processors and optimizes processor utilization. So, whenever a processor becomes free, instead of waiting for other processors to finish their work corresponding to a specific wavefront, it proceeds with the iterations in the next wavefront. Whenever an iteration dependent variable is accessed, it *busy-waits* until all accesses to the specific variable in preceding wavefronts are over, and then it proceeds. The *busy-wait* gives better performance because of the fact that a particular iteration in a given wavefront may not depend on all the iterations in the previous wavefront. Thus, *busy-wait*ing allows overlap of dependent iterations by shifting the dependence granularity from the iteration level to the access level. It gives better performance over *barrier synchronization* particularly when the dependence chains are very long; otherwise *barrier synchronization* performs better due to the run-time overheads involved in the *busy-wait* mechanism.

Iterations	1	2	3	4	5	6	7	8	9
Sequence No. (Access1)	0	2	3	1	0	2	0	1	0
Sequence No. (Access2)	0	1	0	0	0	2	1	0	0

Table 3.2: *Sequence table* with sequence numbers for the different accesses in each iteration in the DOACROSS loop (Figure 1.1).

The *busy-wait* mechanism needs some data structures to be set up in order to sequence the different accesses. It uses a two-dimensional shared array *sequence table* (Table 3.2) and another shared array *Ready* to sequence the different accesses in each iteration. Setting up the *sequence table* (Figure 3.5) has to be done sequentially.

```
/* S_table is a two-dimensional shared array (sequence table). It contains entries
relating to iteration dependent accesses in all the iterations. Ready is a shared array
with N elements. LP is a local pointer to a node in the array of linked lists L. */

Ready[:] = 0;
do j = 1 to Maximum_wavefront_number
     LP = L[j]; /* get the first iteration in level i */
     while LP ≠ NULL /* for each one of the iterations in level i */
          i = LP→iteration; /* get the iteration number */
          /* fill S-table entry with appropriate sequence number */
          S_table[i][1] = Ready[I1[i]];
          Ready[I1[i]] = Ready[I1[i]] + 1; /* increment the sequence number */
          S_table[i][2] = Ready[I2[i]];
          Ready[I2[i]] = Ready[I2[i]] + 1;
          LP = LP→next; /* get the next iteration */
     endwhile
enddo
```

Figure 3.5: Setting up the *sequence table*.

```
/* T is a shared pointer to a node in the array of linked lists L. Level is a shared
variable holding the current wavefront level.  T = NULL and Level = 0 initially.
S_table is a two-dimensional shared array. Ready is a shared array with N elements.
LP is a local pointer to a node in the array of linked lists L. Thread id values range
from 0 to P-1. */

Ready[:] = 0;
Myid = get_myid(); /* get my thread id */
while(1)
    lock(); /* prevents all other accesses to the shared memory */
    if (T) T = T→next; /* gets next iteration in current level */
    else /* go to the next level */
        if ((Level+1) == (M+1))
            { unlock(); return; } /* all levels completed */
        if (L[Level+1] == NULL)
            { unlock(); return; } /* all iterations completed */
        Level = Level + 1; /* go to next level */
        T = L[Level];
    endif
    if (T == NULL) { unlock(); continue; }
    LP = T; /* pick up the iteration to be processed */
    unlock();
    i = LP→iteration; /* get the iteration number */
    ...
    while( Ready[I1[i]] ≠ S_table[i][1] ); /* busy_wait */
    A[I1[i]] = ···
    Ready[I1[i]] = Ready[I1[i]] + 1;
    ...
    while( Ready[I2[i]] ≠ S_table[i][2] ); /* busy_wait */
    ··· = A[I2[i]] + ···
    Ready[I2[i]] = Ready[I2[i]] + 1;
    ...
endwhile
```

Figure 3.6: *Self-scheduled* executor with *busy-wait.*

Chapter 4

Performance on a shared memory machine

In this chapter, we present the performance of our algorithm on a shared memory machine. Our experiments were performed on a 33 MHz, 8-processor SGI shared memory multiprocessor. The machine has 4 CPU boards with 2 CPU's on each board. First level cache hits cost 1 processor cycle; first level misses (i.e., second level hits) cost 3 processor cycles; second level misses costs 57 cycles to bring in 64 bytes. Illinois protocol [1] is the cache coherence scheme used. The machine was run in single-user mode for all experiments.

4.1 Workloads

The performance of our algorithm depends on certain parameters of the loop that we try to parallelize. Table 4.1 shows what these parameters are (Column 2) and what they measure (Column 3). In our analysis we use a loop like that in Figure 4.1 as the workload and try to cover as wide a range of values as possible for these parameters. Column 4 shows the range of values chosen for testing purposes. The array A (Figure 4.1) has $N*r$ elements. The *Hfrac* and *Hsize* parameters are used to simulate different dependence patterns. Each access has a probability *Hfrac* of being

```
/* There are N iterations and r accesses per iteration. INDEX contains the actual
array references. tmp1 and tmp2 are dummy variables. */
    do i = 1,N,1
        do j = 1,r,1
            if(odd(j)) tmp1 = A[INDEX[i*r+j]];
            else A[INDEX[i*r+j]] = tmp2;
            do j = 1,W,1
                dummy loop simulating useful work
            enddo
        enddo
    enddo
```

Figure 4.1: Loops used as experimental workload.

a *hot* reference (reference to a shared variable). The hot section is a portion *Hsize* of array *A*. Loops with long dependence chains and therefore low parallelism have high *Hfrac* and low *Hsize*. In the following sections, a *Mostly-Serial Loop* is one with (*Hsize, Hfrac*) = (10%, 90%), a *Mixed Loop* is one with (*Hsize, Hfrac*) = (50%, 50%) and a *Mostly-Parallel Loop* is one with (*Hsize, Hfrac*) = (90%, 10%).

Param.	Explanation	What We Want to Measure	Values
N	Iteration Count	Problem size scalability	1600, 3200, 6400, 12800,25600
Hfrac	Hot spot fraction	Consistency over dependence patterns	10%, 50%, 90%
Hsize	Hot spot size		0.1%, 10%, 50%, 90%
W	Iteration grain size	Consistency over grain size	40μs (1320 cycles), 160μs (5281 cycles), 640μs (21122 cycles)
r	# references per iteration	Consistency over number of accesses	1, 2, 4, 8

Table 4.1: Loop parameters used in the experiments. The cycles in column 4 are 30 *ns* processor cycles.

4.2 Evaluation

In this section, we analyze our experimental results. First, we show the speedup of our scheme over serial execution. Then we compare our scheme with Chen *et al.*'s scheme. A *pre-scheduled* form of Chen *et al.*'s executor and a *self-scheduled* form of our executor with *barrier* synchronization are used for the comparisons. In Chen *et al.*'s scheme, the *pre-scheduled* executor and *self-scheduled* executor both give very similar performance, since the iteration space is not reordered and the real overhead is due to busy-waiting.

4.3 Performance comparison with serial execution

To determine the speedup of our algorithm, we consider two situations: one in which it performs both inspection and execution (Table 4.2) and another in which it only performs the execution (Table 4.3). The latter corresponds to the case where we reuse the wavefront information gathered during previous inspection of the loop. We use 8 processors in both experiments, and vary the number of iterations, the iteration grain size and the number of references per iteration. Overall, we obtain good speedups: up to 7 when inspector and executor are considered, and up to 8 when only the executor is considered. Obviously, the best results occur when the size of the loop body (W) is large and the number of accesses (r) is low. Table 4.2 and Table 4.3 also show that, as expected, the performance is better when the dependence chains are short; that is why the performance of the *Mostly-Parallel Loop* is better than the other two loops.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.20	1.33	1.30	1.44	1.29
	4	1.45	1.78	1.67	2.06	1.86
	2	1.33	1.94	2.52	2.40	2.63
	1	2.14	2.31	2.77	2.67	2.88
160 μs	8	3.00	3.18	3.13	3.49	3.47
	4	3.60	3.81	4.14	4.41	4.36
	2	3.81	4.44	4.75	4.75	4.98
	1	4.29	4.65	5.24	5.22	5.44
640 μs	8	5.02	5.43	5.75	5.79	5.85
	4	5.74	6.01	5.34	6.50	6.46
	2	5.83	6.30	6.53	6.81	6.91
	1	6.13	6.39	6.75	6.96	7.09

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.29	1.38	1.48	1.25	1.21
	4	1.78	1.68	1.76	1.81	1.91
	2	2.14	2.07	2.33	2.31	2.48
	1	2.14	2.50	2.65	2.86	2.82
160 μs	8	3.15	3.02	3.38	3.42	3.15
	4	3.81	4.21	4.28	4.18	4.16
	2	4.36	4.62	4.84	4.94	5.01
	1	5.00	5.50	5.24	5.45	5.50
640 μs	8	5.74	5.70	5.78	5.79	5.67
	4	6.18	6.17	6.45	6.48	6.56
	2	6.64	6.75	6.81	6.89	7.00
	1	6.46	6.94	7.10	7.03	7.19

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.21	1.25	1.40	1.44	1.43
	4	1.89	2.00	1.91	2.02	2.05
	2	2.30	2.13	2.52	2.60	2.54
	1	3.00	3.10	2.65	2.71	2.72
160 μs	8	3.26	3.05	3.25	3.16	3.02
	4	4.36	4.36	4.44	4.33	4.22
	2	5.00	4.62	5.04	5.03	5.11
	1	5.00	5.22	5.58	5.52	5.60
640 μs	8	5.61	5.95	5.85	5.89	5.96
	4	6.49	6.58	6.54	6.51	6.53
	2	6.64	6.73	6.89	6.89	6.95
	1	7.03	6.94	7.10	7.12	7.17

(c)

Table 4.2: Speedup of our algorithm over serial execution including *inspector* and *executor* using 8 processors.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
40 μs	8	3.60	5.14	5.46	5.96	6.50
	4	4.00	4.00	5.42	6.19	6.44
	2	3.75	4.43	5.73	5.95	6.41
	1	3.75	4.30	5.55	6.15	6.28
160 μs	8	5.73	6.53	6.94	7.30	7.43
	4	5.55	6.42	6.97	7.30	7.50
	2	6.10	6.32	6.91	7.22	7.49
	1	6.00	6.40	7.09	7.27	7.52
640 μs	8	6.70	7.00	7.37	7.62	7.78
	4	6.89	7.07	6.24	7.63	7.76
	2	6.64	7.04	7.38	7.60	7.79
	1	6.83	7.04	7.43	7.62	7.80

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	4.50	6.00	6.36	6.22	6.36
	4	5.33	6.40	6.50	6.50	6.60
	2	7.50	6.20	6.30	5.68	6.58
	1	5.00	6.00	6.78	6.83	6.81
160 μs	8	7.00	6.89	7.58	7.51	7.43
	4	6.78	7.63	7.18	7.52	7.61
	2	6.78	7.06	7.33	7.56	7.61
	1	7.50	7.56	7.53	7.62	7.70
640 μs	8	7.53	7.67	7.73	7.80	7.81
	4	7.53	7.63	7.75	7.82	7.84
	2	7.47	7.73	7.80	7.82	7.86
	1	7.47	7.60	7.85	7.84	7.87

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	5.67	5.83	6.36	6.81	6.50
	4	5.67	6.40	6.50	6.55	6.72
	2	5.33	6.40	6.30	6.58	6.78
	1	7.50	7.75	6.78	6.42	7.17
160 μs	8	7.75	7.35	7.35	7.58	7.60
	4	7.63	7.63	7.40	7.64	7.66
	2	7.50	7.06	7.56	7.55	7.64
	1	6.67	7.50	7.74	7.62	7.70
640 μs	8	7.53	7.65	7.78	7.88	7.88
	4	7.74	7.74	7.81	7.87	7.87
	2	7.71	7.71	7.84	7.85	7.89
	1	7.71	7.73	7.80	7.88	7.90

(c)

Table 4.3: Speedup of our algorithm over serial execution including *executor only* using 8 processors.

The results show that run-time parallelization is a viable approach to speed up loops that cannot be handled by compile-time parallelization. This approach gives

good results when (a) both the inspector and executor are considered and (b) the executor only is considered. There is no big leap in performance when the inspector results are reused because the inspector stage is very efficient and fast. To see how the speedup varies with the number of processors, we also ran several experiments with the three different types of loops using various numbers of processors. With up to 8 processors, we obtain almost linear speedups using (1) both inspector and executor and (2) executor only. We expect the speedup to be scalable to a larger numbers of processors.

4.4 Performance comparison with Chen *et al.*'s scheme

Table 4.4 gives the speedup of our scheme over Chen *et al.*'s when both inspector and executor are considered. It indicates the superior performance of our algorithm in this case. It is also clear that the performance of our approach does not degrade rapidly as the DOACROSS loop structure becomes more and more parallel in nature. This is largely because our inspector is much faster and more efficient. Moreover, it does not require special synchronization operations and uses very little inter-processor communication.

W	r	**Mostly-Serial Loop** ($Hsize = 10\%$, $Hfrac = 90\%$)				
		N=1600	3200	6400	12800	25600
40 μs	8	4.60	13.85	37.40	110.56	251.99
	4	1.82	3.83	10.08	32.67	76.97
	2	1.11	1.50	3.28	7.75	21.80
	1	1.00	1.00	1.45	2.07	5.04
160 μs	8	3.71	10.21	26.11	76.44	194.45
	4	1.82	2.88	7.17	19.00	49.40
	2	1.25	1.67	2.41	4.70	11.44
	1	1.14	1.27	1.52	1.87	3.27
640 μs	8	2.46	5.27	13.18	33.94	86.60
	4	1.62	2.14	3.71	8.17	19.83
	2	1.44	1.62	1.90	2.84	5.12
	1	1.41	1.47	1.58	1.77	2.25

(a)

W	r	**Mixed Loop** ($Hsize = 50\%$, $Hfrac = 50\%$)				
		N=1600	3200	6400	12800	25600
40 μs	8	23.71	98.31	348.4	744.25	1554.58
	4	6.78	17.58	69.22	232.65	614.60
	2	2.86	4.20	12.67	48.04	165.73
	1	1.43	1.75	2.96	8.14	29.93
160 μs	8	16.55	62.61	226.22	576.05	1153.98
	4	4.13	11.86	44.82	142.49	359.50
	2	1.86	2.85	7.22	26.74	86.73
	1	1.33	1.55	1.98	4.38	15.29
640 μs	8	8.62	30.39	100.75	255.22	538.18
	4	2.31	5.09	17.85	57.16	145.60
	2	1.33	1.68	3.23	9.93	31.27
	1	1.03	1.14	1.35	2.05	5.68

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	24.50	95.36	348.70	1984.33	1898.82
	4	6.78	21.50	80.24	349.65	690.24
	2	2.86	4.13	14.00	56.69	177.16
	1	2.20	2.30	2.91	7.98	30.82
160 μs	8	18.47	66.63	229.27	560.09	1142.25
	4	4.79	12.93	49.29	156.82	379.82
	2	2.17	2.88	7.88	29.42	92.64
	1	1.25	1.43	2.07	4.74	16.39
640 μs	8	8.77	33.67	107.56	268.71	584.97
	4	2.41	5.53	19.20	60.29	151.33
	2	1.36	1.69	3.35	10.86	32.70
	1	1.15	1.14	1.33	2.19	5.90

(c)

Table 4.4: Speedup of our algorithm over Chen et al.'s scheme including *inspector* and *executor* using 8 processors.

Table 4.5 considers the executor only, which is important if the inspector results are reused across loop invocations. It shows that our executor gives better performance for the *Mostly-Serial* and *Mixed* Loops. This is because our scheme has better iteration space to processor mapping during the executor stage whereas their executor spends processor cycles *busy-waiting*. In the case of *Mostly-Parallel* loops, both the schemes show nearly identical results because there is not much *busy-waiting* overhead involved.

W	r	**Mostly-Serial Loop** ($Hsize = 10\%$, $Hfrac = 90\%$)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.43	1.54	1.63	1.75
	4	1.25	1.00	1.42	1.62	1.66
	2	1.25	1.14	1.45	1.57	1.64
	1	1.00	1.14	1.36	1.50	1.54
160 μs	8	1.36	1.58	1.61	1.68	1.73
	4	1.45	1.47	1.60	1.67	1.71
	2	1.40	1.47	1.57	1.64	1.70
	1	1.40	1.47	1.59	1.65	1.70
640 μs	8	1.50	1.55	1.63	1.68	1.71
	4	1.51	1.56	1.62	1.69	1.71
	2	1.47	1.56	1.62	1.72	1.71
	1	1.51	1.54	1.62	1.67	1.72

(a)

W	r	**Mixed Loop** ($Hsize = 50\%$, $Hfrac = 50\%$)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.67	1.82	1.78	1.84
	4	1.00	1.20	1.30	1.40	1.43
	2	1.00	1.00	1.10	1.05	1.21
	1	0.67	0.80	1.11	1.06	1.08
160 μs	8	1.00	1.06	1.18	1.15	1.16
	4	0.89	1.06	1.03	1.06	1.09
	2	0.89	0.94	0.97	1.03	1.04
	1	1.00	1.00	1.00	1.00	1.02
640 μs	8	1.00	1.02	1.02	1.03	1.03
	4	1.00	0.98	1.00	1.01	1.02
	2	0.94	0.98	0.99	1.00	1.01
	1	0.94	0.97	0.99	1.00	1.00

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.00	1.09	1.33	1.11
	4	1.00	1.00	1.00	1.15	1.08
	2	0.67	0.80	0.90	1.00	1.03
	1	1.50	1.25	1.00	0.95	1.03
160 μs	8	1.13	1.06	1.03	1.03	1.03
	4	1.00	1.00	1.00	1.02	1.02
	2	1.00	0.94	1.00	1.00	1.00
	1	0.89	0.94	1.00	1.00	1.01
640 μs	8	1.00	1.00	1.01	1.01	1.01
	4	1.00	0.98	1.01	1.01	1.00
	2	1.00	0.98	1.00	1.00	1.01
	1	1.00	0.98	0.98	1.00	1.01

(c)

Table 4.5: Speedup of our algorithm over Chen et al.'s scheme including *executor only* using 8 processors.

In Figures 4.2 and 4.3, the sensitivity of the inspector (both schemes) to the different types of loops is shown. The columns are normalized to the *Mostly-Serial* case. Results show that our inspector is less sensitive to differences in loop type and gives consistent performance compared to Chen *et al.*'s scheme.

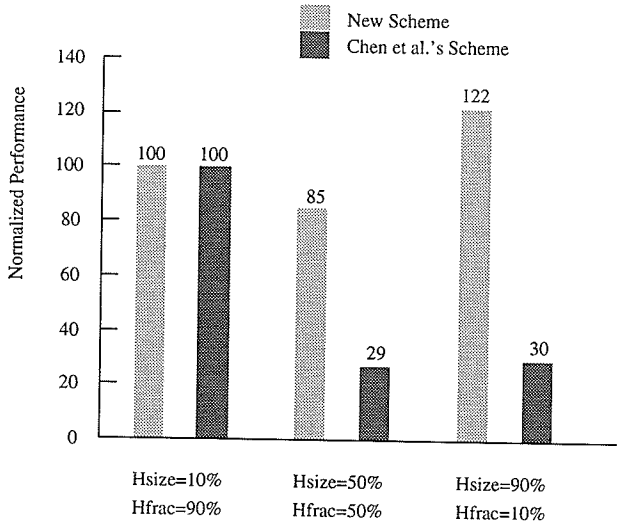

Figure 4.2: The sensitivity of the *inspector* to different types of loops for iteration count $(N) = 1600$ and number of accesses per iteration $(r) = 1$.

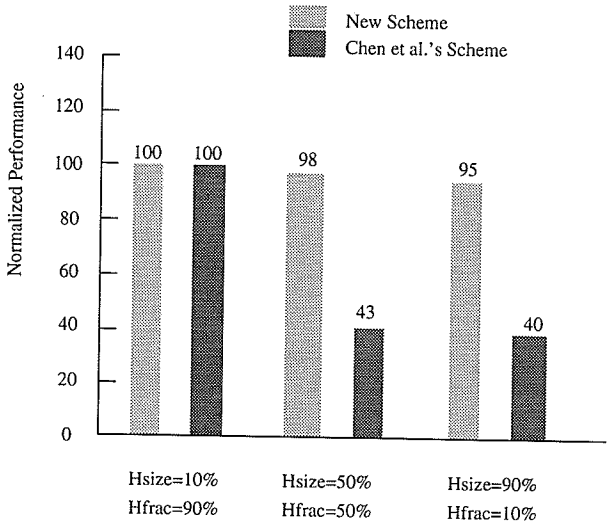

Figure 4.3: The sensitivity of the *inspector* to different types of loops for iteration count $(N) = 25600$ and number of accesses per iteration $(r) = 1$.

In Figures 4.4 and 4.5, the sensitivity of the inspector (both schemes) to the number of references per iteration is shown. The columns are normalized to the $r = 1$ case. Results show that our inspector gives much better performance compared to Chen *et al.*'s scheme even though it, too, is sensitive to the number of references per iteration.

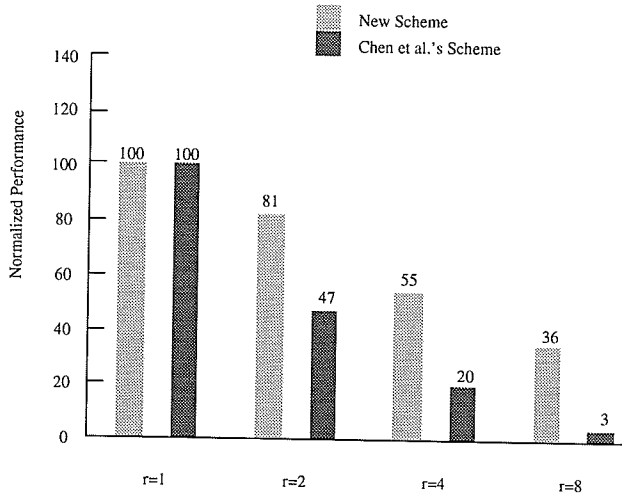

Figure 4.4: The sensitivity of the *inspector* to the number of references per iteration for iteration count $(N) = 1600$.

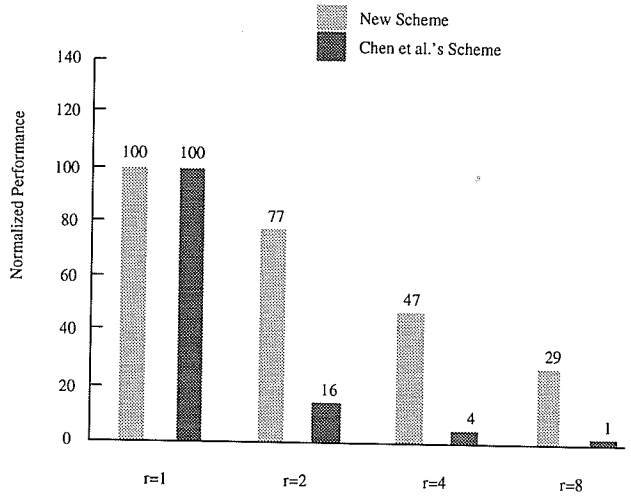

Figure 4.5: The sensitivity of the *inspector* to the number of references per interation for iteration count $(N) = 25600$.

In Figures 4.6 and 4.7, the sensitivity of the executor (both schemes) to the different types of loops is shown. The columns are normalized to the *Mostly-Parallel* case. Results show that our executor is less sensitive to differences in loop types compared to Chen *et al.*'s scheme.

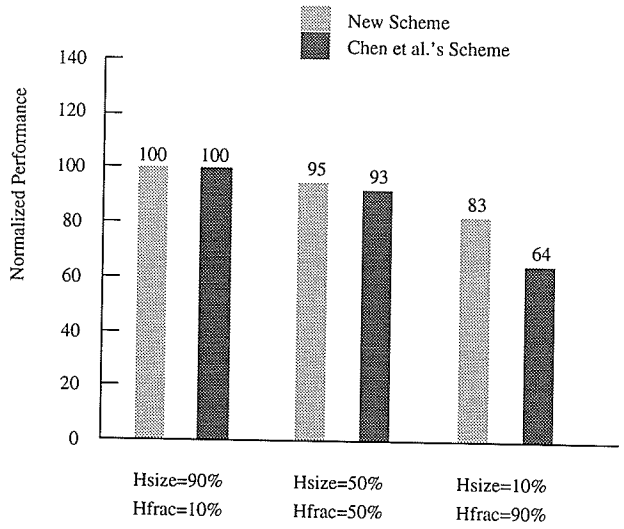

Figure 4.6: The sensitivity of the *executor* to different types of loops for iteration count $(N) = 1600$ and number of accesses per iteration $(r) = 1$.

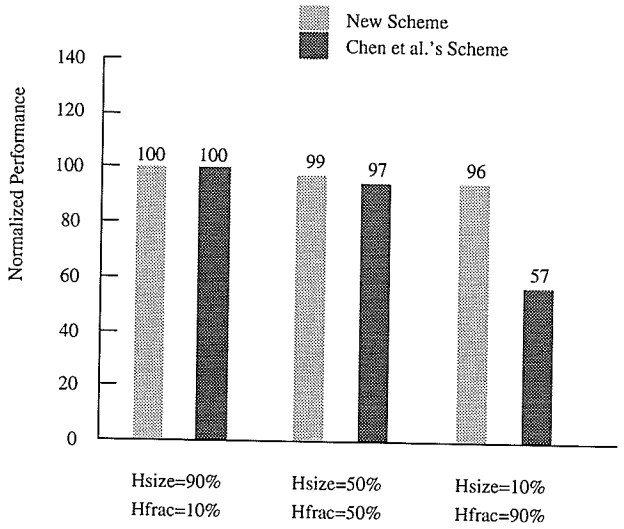

Figure 4.7: The sensitivity of the *executor* to different types of loops for iteration count $(N) = 25600$ and number of accesses per iteration $(r) = 1$.

In Figures 4.8 and 4.9, the sensitivity of the executor (both schemes) to the number of references per iteration is shown. The columns are normalized to the $r = 1$ case. Results show that both the schemes are relatively insensitive to the number of references per iteration. Still our scheme has slightly better performance compared to Chen *et al.*'s scheme.

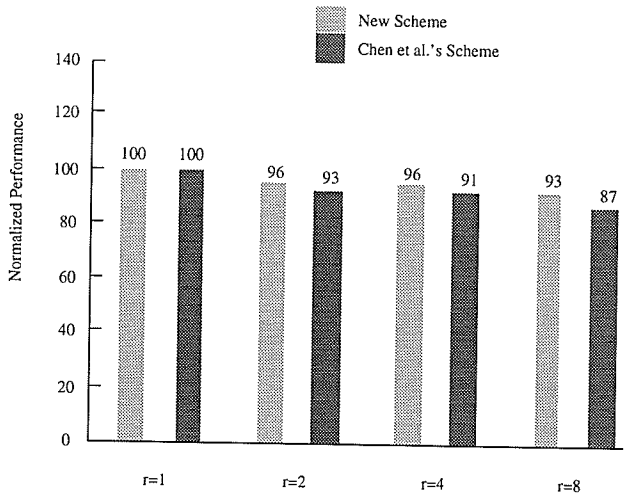

Figure 4.8: The sensitivity of the *executor* to the number of references per iteration for iteration count $(N) = 1600$.

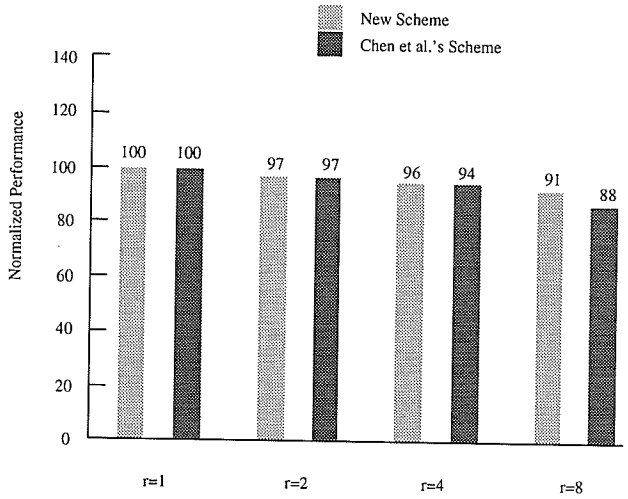

Figure 4.9: The sensitivity of the *executor* to the number of references per iteration for iteration count $(N) = 25600$.

Summarizing these results we have the following:

- Our inspector gives consistent performance for all the different types of loops whereas Chen *et al.*'s inspector lacks consistency across the various loop types (i.e, performance degrades as the DOACROSS loop structure becomes more and more parallel in nature).
- During the executor stage, the new scheme ensures better iteration space to processor mapping thereby achieving better load balance and overall speedup.

4.5 Performance of the executor: *Barrier synchronization vs. Busy-wait*

Table 4.6 compares the speedups of the two different types of executor in the new scheme with Chen *et al.*'s executor. The results show that the executor (*busy-wait*) gives better performance than the executor (*barrier synchronization*) for *Mostly-Serial* loops. From Table 4.7 it is evident that for *Mostly-Parallel* loops the performance of the executor (*busy-wait*) is poor compared to the executor (*barrier synchronization*). The results include the run-time overhead of setting up the *sequence table* in the case of executor (*busy-wait*). Also note that *busy-waiting* does not induce any significant increase in bus traffic.

W	r	**Executor (New scheme)** *(Barrier synchronization)*				
		N=1600	3200	6400	12800	25600
40 μs	8	0.52	0.39	0.47	0.60	1.08
	4	0.37	0.49	0.42	0.55	0.98
	2	0.39	0.35	0.47	0.51	0.94
	1	0.28	0.32	0.35	0.42	0.88
160 μs	8	0.95	0.71	0.85	1.09	1.99
	4	0.72	0.97	0.78	1.07	1.93
	2	0.66	0.73	0.98	1.03	1.92
	1	0.61	0.66	0.72	0.97	1.85
640 μs	8	1.32	0.96	1.14	1.47	2.74
	4	1.00	1.35	1.09	1.47	2.74
	2	0.92	1.00	1.36	1.44	2.72
	1	0.85	0.93	1.01	1.36	2.61

(a)

W	r	**Executor (New scheme)** *(Busy-wait)*				
		N=1600	3200	6400	12800	25600
40 μs	8	0.89	0.79	0.88	1.03	1.42
	4	0.94	1.10	0.99	1.25	1.80
	2	1.13	1.00	1.30	1.37	2.19
	1	0.94	1.03	1.07	1.39	2.37
160 μs	8	1.30	0.98	1.15	1.45	2.36
	4	1.12	1.44	1.18	1.55	2.68
	2	1.06	1.15	1.53	1.59	2.89
	1	0.98	1.07	1.16	1.55	2.91
640 μs	8	1.52	1.08	1.28	1.63	2.96
	4	1.17	1.57	1.25	1.68	3.10
	2	1.08	1.18	1.60	1.67	3.17
	1	0.99	1.08	1.19	1.60	3.07

(b)

Table 4.6: Comparison of the speedup of the two different types of executors in the new scheme relative to Chen et al.'s executor. The results correspond to a Mostly-Serial Loop ($Hsize = 0.1\%$, $Hfrac = 90\%$).

W	r	**Executor (New scheme)** (*Barrier synchronization*)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.00	1.09	1.33	1.11
	4	1.00	1.00	1.00	1.15	1.08
	2	0.67	0.80	0.90	1.00	1.03
	1	1.50	1.25	1.00	0.95	1.03
160 μs	8	1.13	1.06	1.03	1.03	1.03
	4	1.00	1.00	1.00	1.02	1.02
	2	1.00	0.94	1.00	1.00	1.00
	1	0.89	0.94	1.00	1.00	1.01
640 μs	8	1.00	1.00	1.01	1.01	1.01
	4	1.00	0.98	1.01	1.01	1.00
	2	1.00	0.98	1.00	1.00	1.01
	1	1.00	0.98	0.98	1.00	1.01

(a)

W	r	**Executor (New scheme)** (*Busy-wait*)				
		N=1600	3200	6400	12800	25600
40 μs	8	0.25	0.26	0.25	0.29	0.26
	4	0.43	0.36	0.34	0.48	0.36
	2	0.40	0.40	0.45	0.48	0.48
	1	1.00	0.71	0.60	0.60	0.59
160 μs	8	0.53	0.53	0.52	0.50	0.50
	4	0.67	0.64	0.66	0.64	0.65
	2	0.80	0.80	0.78	0.77	0.76
	1	1.00	0.83	0.84	0.85	0.84
640 μs	8	0.84	0.83	0.82	0.82	0.81
	4	0.91	0.90	0.90	0.89	0.89
	2	0.97	0.94	0.95	0.94	0.94
	1	0.97	0.97	0.96	0.96	0.98

(b)

Table 4.7: Comparison of the speedup of the two different types of executors in the new scheme relative to Chen et al.'s executor. The results correspond to a Mostly-Parallel Loop($Hsize = 90\%$, $Hfrac = 10\%$).

In the *Mostly-Serial* loops, there are many more opportunities for overlap among dependent iterations (since there will be number of dependent iterations), whereas in

the case of *Mostly-Parallel* loops there are fewer dependent iterations, so the chances for overlapped execution of dependent iterations are far fewer. Thus, it is quite obvious that the executor (*busy-wait*) does not give better performance when compared to the executor (*barrier synchronization*) in the *Mostly-Parallel* case.

Chapter 5

Performance on a distributed memory machine

In this chapter, we evaluate the performance of our algorithm on a distributed memory machine. We use PVM (Parallel Virtual Machine) [5] to utilize a collection of networked heterogeneous workstations as a single distributed memory machine. Parallel Virtual Machine (PVM) is a widely-used software system that allows a heterogeneous set of parallel and serial UNIX-based computers to be programmed as a single distributed-memory parallel machine. It is portable and runs on a wide variety of modern platforms. It is the mainstay of the Heterogeneous Network Computing research project, a collaborative venture between the Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University. We present a brief overview of PVM, its architecture, its computing model, the programming interface it supports, auxiliary facilities for process groups and its use on highly parallel systems like the IBM SP-2, Cray T3D, etc. Further, issues and possibilities for faster message passing and active messaging are discussed. We then touch upon MPVM (a Migration transparent version of PVM) which supports transparent process migration among the multiple hosts that constitute the virtual machine. We then present the speedups of the distributed version of our algorithm for loop paral-

lelization over serial execution. The performance of our algorithm on the distributed memory model is encouraging particularly when the granularity of the computation is large enough to justify distribution.

5.1 PVM: Architecture and programming

PVM architecture

The PVM system consists of a daemon process called the *pvmd* running on each host on a network of workstations (NOW) and a run-time library called the *pvmlib* linked into each application process (Figure 5.1). Each *pvmd* is assigned a unique host ID or hid. It is designed so that any user with a valid login can install this daemon on a machine. A user wishing to run a PVM application creates a virtual machine by starting up PVM. This involves starting *pvmd* on each of the nodes in the virtual machine (which is done automatically). The *pvmd* running on the node which started up PVM becomes the master *pvmd*. When a master *pvmd* dies, all other *pvmd*s (slave *pvmd*s) also die. But if any one of the slave *pvmd*s die due to node failure or loss of network accessibility, the node is removed from the virtual machine configuration automatically (by a time-out mechanism). The pvmlib defines a suite of PVM primitives that presents a "message-passing parallel machine" API (Application Programming Interface) to the application.

A PVM application is composed of Unix processes linked with the pvmlib. These processes, called tasks in PVM, communicate with each other via message-passing primitives found in the pvmlib. Just like the *pvmd*s, each task is assigned a task ID or tid which uniquely identifies each task in the virtual machine. These tids are used to designate the source and destination tasks for messages (i.e., messages are addressed to tasks, not to ports or mailboxes). Although PVM encodes information in each tid,

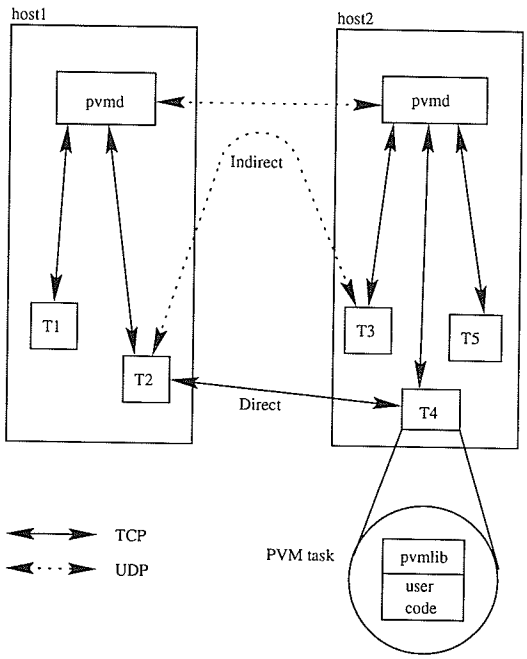

Figure 5.1: PVM system.

the user is expected to treat the tids as opaque integer identifiers. PVM contains several routines that return tid values so that the user application can identify other tasks in the system.

PVM includes the concept of user named groups. When a task joins a group, it is assigned a unique "instance" number in that group. Instance numbers start at 0 (the group owner) and count up. In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user. For example, any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Also, groups can overlap, and tasks can broadcast messages to groups of which they are not a member. There are specific group functions available for group operations like broadcast, barrier synchronization, global sum etc. Applications have to be linked with a special library *libgpvm* for group operations. Normally, the various tasks in a group are identified by the numbers *0, ..., (p - 1)*,

where p is the number of tasks in the group.

Messages passed within the PVM system can be categorized into system messages and application messages. System messages are used exclusively by PVM to manage the virtual machine and perform application code requests (e.g, spawn a new task, get information about the virtual machine, etc.). The application code is not aware of these messages. Application messages on the other hand are generated and used exclusively by the PVM applications.

PVM provides two routing mechanisms for application messages; indirect and direct routing. The choice of routing mechanism to use is controlled by the application code. By default, messages are routed indirectly. Using indirect routing, as illustrated in Figure 5.1, a message from task T2 to T3 passes through T2's local *pvmd* (*pvmd* on host1), through T3's local *pvmd* (*pvmd* on host2), and finally to T3. *Pvmd*-to-*Pvmd* communication uses UDP (User Datagram Protocol) socket connections while task-to-task communications use a TCP (Transmission Control Protocol) socket connection which is established during task start-up. In direct routing (PvmRouteDirect), a message from task T2 to T4, also illustrated in Figure 5.1, uses a TCP socket connection between T2 and T4, by-passing the *pvmd*s altogether. TCP connections between tasks are created "on-demand". Only when tasks that have set their routing option to use direct routing start communicating with each other are TCP connections established.

In indirect routing (the default routing mechanism), connectionless UDP sockets guarantee scalability, since a single UDP socket can communicate with any number of tasks (local or remote). Because the communication between tasks is routed through the *pvmd*s, however, messages need three hops to reach their destination. This is not very efficient. In the case of direct routing (task-to-task), connection-oriented

TCP sockets are used for direct communication between the tasks. The use of TCP sockets tends to exhaust the limited number of file descriptors in a system. Since TCP connections establish a direct communication link between the tasks, however, messages reach their destination in a single hop.

An important aspect to remember when using PVM is the message ordering semantics it provides. PVM guarantees that messages sent from one task to another are received in the same order they were sent. The importance of recognizing this "guarantee" is that there are PVM applications that will take advantage of this message-ordering semantics. Hence, new versions of PVM such as MPVM should maintain the same semantics.

In PVM version 3.3.4 and above, it is possible to designate a special task as the resource manager. The resource manager, also called the global scheduler (GS), is responsible for decision making policies such as task-to-processor allocation for sensibly scheduling multiple parallel applications. Using a global scheduler makes it convenient to experiment with different scheduling policies. In MPVM, the interface between the *pvmd*s and the GS has been extended to accommodate task migration, allowing the GS to use dynamic scheduling policies.

PVM programming

The PVM computing model assumes that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism. In this method all the tasks are

the same, but each one solves only a small part of the problem using a subset of the data. This is also referred to as the SPMD (single-program multiple-data) model of parallel computing. PVM supports either or a mixture of these methods. The PVM system currently supports the C, C++, and Fortran languages.

The general paradigm for application programming with PVM is as follows. A user writes one or more sequential programs in C, C++, or Fortran 77 that contain embedded calls to the PVM library. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from the relevant machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem. Note that while the above is a typical scenario, as many tasks as appropriate may be started manually. As mentioned earlier, tasks interact through explicit message passing, identifying each other with a system-assigned, opaque tid.

Shown in Figure 5.2 is the body of the PVM program *hello*, a simple example that illustrates the basic concepts of PVM programming. This program is intended to be invoked manually. After printing its task id (obtained with `pvm_mytid()`), it initiates a copy of another program called *hello_other* using the `pvm_spawn()` function. A successful spawn results in code being executed which causes the spawning program to execute a blocking receive using `pvm_recv`. After receiving the desired message from the spawned process, the program prints the message sent by its counterpart, as well as the counterpart's task id. The message buffer is extracted from the message

```
#include "pvm.h"
main()
{
int cc, tid, msgtag;
char buf[100];

printf("i'm t%x\n", pvm_mytid());

cc = pvm_spawn("hello_other", (char **) 0, 0, "", 1, &tid);

if (cc == 1) {
msgtag = 1;
pvm_recv(tid, msgtag);
pvm_upkstr(buf);
printf("from t%x: %s\n", tid, buf);
}
else
printf("can't start hello_other\n");

pvm_exit();
}
```

Figure 5.2: PVM program *hello.c*

```
#include "pvm.h"

main()
{
int ptid, msgtag;
char buf[100];

ptid = pvm_parent();

strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);
msgtag = 1;
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, msgtag);

pvm_exit();
}
```

Figure 5.3: PVM program *hello_other.c*

using `pvm_upstr`. The final `pvm_exit` call dissociates the program from the PVM system.

Figure 5.3 is a listing of the "slave" or spawned program; its first PVM action is to obtain the task id of the "master" using the `pvm_parent` call. This program then obtains its hostname and transmits it as the message string to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place the string, in a strongly typed and architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, "tagging" the message with the number 1.

XPVM

XPVM [9] is a graphical user interface for PVM. Even though the primary function of XPVM is to visualize what is going on inside a PVM application, the interface actually performs four separate functions. First, XPVM is a graphical console allowing a user to start PVM, add hosts, spawn tasks, reset the virtual machine, and stop PVM all by clicking buttons. Second, XPVM is a real-time performance monitor. While an application is running, XPVM displays a space-time diagram of the parallel tasks showing when they are computing, communicating, or idle. The display also animates the message traffic between tasks. Third, XPVM is a "call level" debugger. The user can see the last PVM call made by each task as the application is running. Fourth, XPVM is a post-mortem analysis tool. XPVM writes out a trace file that can be replayed, stopped, stepped, and rewound in order to study the behavior of a completed parallel run. XPVM is built on TCL/TK [17], which is a library for developing X-Windows applications. TCL/TK makes it easy to change, customize, and extend the XPVM interface.

5.2 PVM extensions

Faster message passing

The communication routines *pvm_psend* and *pvm_precv* combine the initialize, pack, and send operations into a single call reducing the call overhead. The biggest performance gain is seen when using these routines on an MPP (Massively Parallel Processor). Psend and precv can be as fast as native communication routines on an MPP. The vendors of MPPs like the IBM SP-2, the Intel Paragon, the CM-5, etc., provide optimized implementations of *pvm_psend* and *pvm_precv* for their own architectures [6].

PVM uses the XDR (eXternal Data Representation) data encoding format by default (*PvmDataDefault*) for communication between different nodes in the virtual machine. If the communication is between nodes using the same data format, then the data encoding option (*PvmDataRaw*) can be specified instead, while initializing the send buffer. Psend and precv use the *PvmDataRaw* data encoding format, which means that it can only be used for communication between nodes which have the same data format.

To reduce the number of data copies, an *InPlace* packing option was added. With this option the user's data is left in-place in the user's memory and not copied into a send buffer. The user has to make sure that the data is not overwritten before the actual send takes place. PVM keeps track of what data the user has packed and copies the different pieces directly from the user's memory to the network. This option not only improves the communication speed, but also reduces the buffer space required inside PVM.

For XPVM to work, PVM must send out trace events of what is happening in the virtual machine. These trace messages can be quite intrusive and cause a degradation in performance (tracing is turned off by default). To address this problem traces may be buffered. Experiments have shown that local buffering of even a modest number of trace events (20 events) can give a significant performance increase over no buffering for many (but not all) applications. Because buffered tracing is less intrusive, it also results in more accurate monitoring.

To implement faster collective communication operations and more efficient group operations on multiprocessors, investigations are underway to add the concept of static groups to PVM. Currently, all groups are dynamic in PVM. Group members can join and leave asynchronously. What may be added is a way for the user to

specify that a group is now static, i.e., its membership will no longer change. Once this occurs, PVM can cache the group information about the tasks.

There are also plans to incorporate the concept of *context* into PVM. Context as defined in the MPI (Message Passing Interface) [26] documentation isolates a communication space and allows the creation of "safe" parallel libraries. Such libraries simplify application development while also supplying fast, robust, parallel implementations. Thus, the addition of context into PVM will improve application performance. Unlike MPI, PVM will incorporate context as an option so that all existing code continues to run unchanged and users need not understand or use context calls unless they require them.

Network multicast is a feature being incorporated into most new operating systems. Multicast sends one message onto a network that is received by a predetermined set of hosts. This is potentially much faster than the present method in PVM, which sends separate copies of a message to each host.

Different approaches have been taken to build faster message passing mechanisms into PVM [7]. One of the recent efforts is to use shared memory based channels to replace the socket-based mechanism between the PVM daemon and its local tasks. Another effort is the construction of a new mechanism named PvmRouteAtm to exploit the bandwidth of an ATM network based on the socket-like application programming interface from FORE Systems (FORE Systems supplies this interface with their ATM cards). Test results show that better performance has been achieved with PvmRouteAtm than with PvmRouteDirect. The shared memory effort failed to gain performance due to reasons such as context switching overhead and slow semaphore operations. Standard PVM works with ATM networks by using TCP/IP. Experiments have shown that communication between two SPARC2 computers is up to five

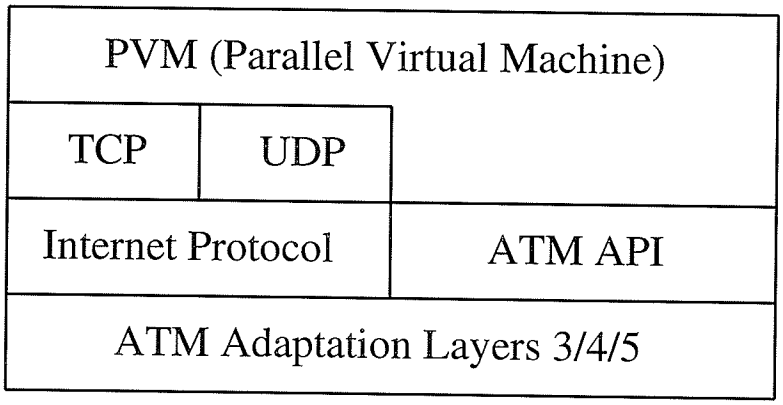

Figure 5.4: Protocol Hierarchy.

times faster on an ATM network than on an ethernet. Other experiments have shown that an additional 30% improvement is possible by using an API below TCP.

The FORE systems ATM API (Figure 5.4) library routines provide a portable, socket-like interface to the ATM data link layer. This interface is at about the same level as IP. It provides support for a connection-oriented client-server model. After a connection is set up, the network makes a 'best' effort to deliver ATM cells to the destination. But cells may be dropped during the transmission depending on the availability of resources. So, unlike the connection-oriented TCP socket, flow control and retransmission facilities have to be provided by the applications. Like BSD socket programming's *socket()* call, *atm_open()* is used to open a file descriptor in the ATM API and then bind a local Application Service Access Point to the file descriptor with *atm_bind()*.

There are 5 AALs (ATM Adaptation Layers) in ATM (AAL1-5) with different features. FORE systems API is based on AAL3/4 (AAL3 and AAL4 are treated identically) and AAL5. AAL5 is provided to support communication in computer networks with performance optimization. It was proposed mainly by the computing industry as a method of reducing data transport overhead within the telecommunication-oriented

ATM infrastructure. It has a Simple and Efficient ATM Adaptation Layer (also known as SEAL) that can be used for bridged and routed Protocol Data Units (PDUs). *Atm_send()* and *atm_recv()* are used to transfer messages. One PDU is transferred on each call. The maximum size of a PDU depends on the AAL selected and the constraints imposed by the underlying device driver implementation.

ATM also has lower level support for operations such as *multicast*. This is another topic of interest to PVM. Further, a single host may be multi-homed, as a result it may have different names in the Ethernet and ATM environments. Something has to be done to avoid naming conflict in a PVM/ATM implementation.

Possibilities for Active Messaging

Active messaging [18] is a communications model designed around the interaction of a network interface and its driving software in an operating system. By utilizing this model, the user can design applications that make better use of the available computing and communication resources. Currently, successful implementations exist only for a certain subset of workstations and network adapters.

Active messaging (AM) was first introduced at the University of California at Berkeley [22] as a means to reduce the communication latency inherent in distributed computing. This idea was driven by the discrepancy in performance between traditional multi-phase communication protocols and the actual access time of the network device. The concept of an AM is as elegant as it is simple. When an AM arrives, the address of a user-defined handler is read out of the message's header. This handler is then invoked with a pointer to the message as its argument. This handler is small and compact. It performs some finite amount of work, which may include sending a reply AM, and then returns. The user's program can then continue. The role of the

user-defined handler is merely to fill or empty the data from the network's buffers. Usually, this consists of a simple *read()* or *memcpy()* and the setting of a few flags that are polled during computation.

In active messaging, the key to performance lies in the close interaction with the network hardware, to avoid protocol specific delays. This may very well limit the portability of applications. Another relevant issue is the delivery of active messages while the user's code is executing in a critical section. This of course, can be avoided by polled I/O. The question of whether AMs fully eliminate the OS message buffering problem is still debatable. Furthermore, the cost of context switching, particularly when the user's code is compute-intensive is to be noted.

Overall, active messaging could prove to be a worthwhile addition to the PVM suite if done carefully. However, an AM implementation will not provide an overwhelming "out-of-the-box" improvement over PVM's direct routed *psend* and *precv* APIs. AMs will provide users the tools to redesign their applications to communicate more efficiently. The asynchronous nature of AMs combined with overlapped communication and computation can lead to orders of magnitude speedup, especially on algorithms that interleave massive data movement and computation. If it can be shown that the AM model can support PVM's level of functionality and that active messages routinely generate more robust and efficient distributed applications, then the issue of the conversion of the PVM suite to use this paradigm should be seriously considered.

MPVM: A Migration transparent version of PVM

MPVM [8] is an extension of PVM which allows parts of the parallel computation to be suspended and subsequently resumed on other workstations. There are three

key goals to consider in the design of MPVM. First, migration has to be transparent to both application programmer and user. Neither the programmer nor the user needs to know that portions of the application are migrating. Second, source-code compatibility with PVM has to be maintained. Source-code compatibility will allow existing PVM applications to run under MPVM without, or at least with minimal, modification. Lastly, MPVM has to be as portable as possible.

Task migration is the ability to suspend the execution of a task on one machine and subsequently resume its execution on another. A major requirement for task migration is that the migration should not affect the correctness of the task. Execution of the task should proceed as if the migration never took place. To ensure the "transparency" of the migration, it is necessary to capture the state of the task on the source machine and reconstruct it on the target machine. The problem addressed by MPVM is how to capture and reconstruct the state information so that tasks can be migrated from one machine to another without affecting the correctness of the entire application.

Once the application is up and running, it executes just like a traditional PVM application would, until a task has to migrate. There are a number of reasons for a task to migrate: excessively high machine load, machine reclamation by its owner, a more suitable machine becomes available, etc. Regardless of the rationale for migration, the same migration mechanism can be used.

A migration protocol is used to facilitate the migration. The migration protocol is divided into four stages as shown in Figure 5.5. While the first stage addresses "when" migration occurs, the last three stages correspond exactly to the main steps in migration: state capture, transfer, and state re-construction. An important component of the migration protocol is what are collectively called Control Messages.

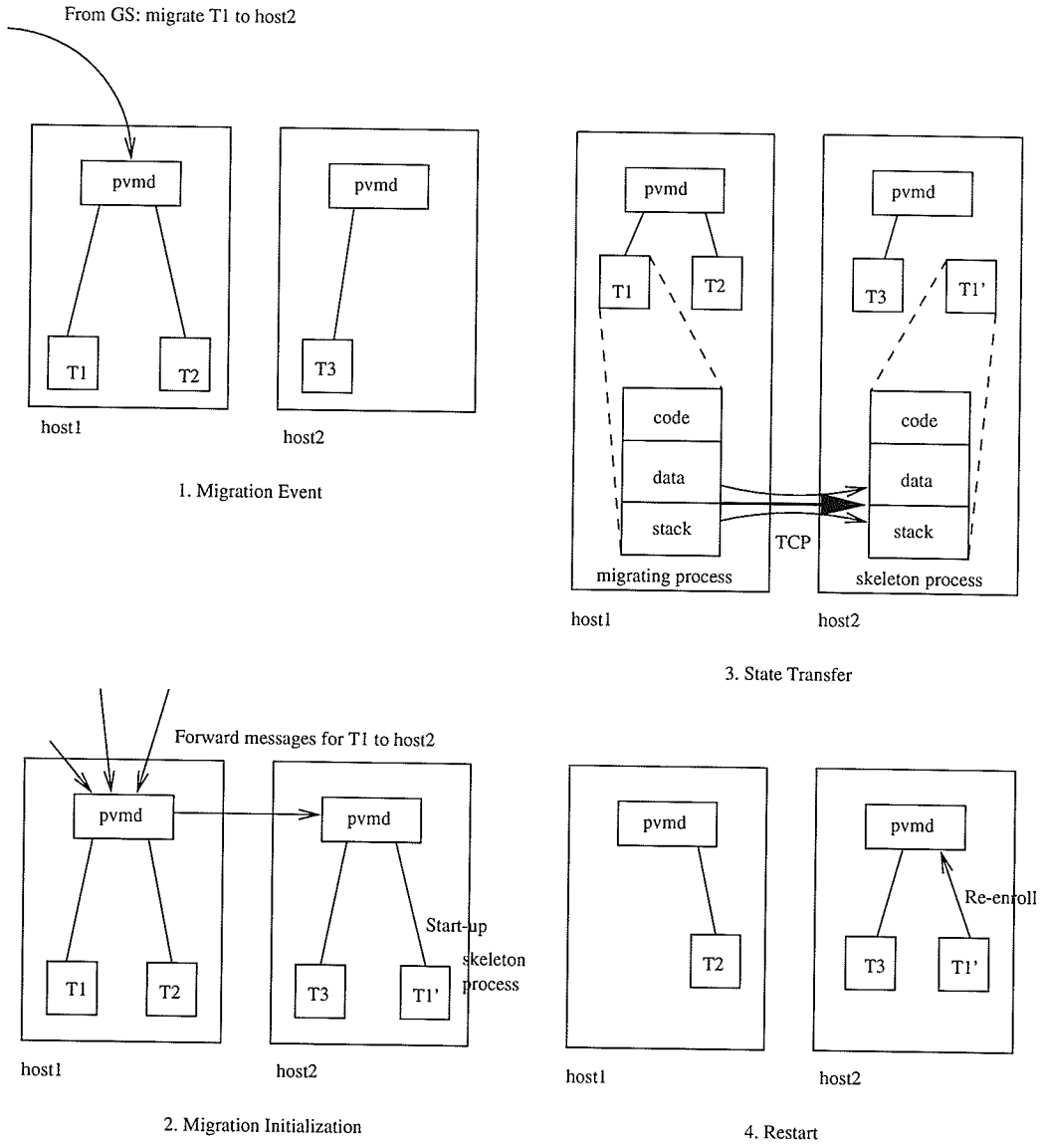

Figure 5.5: The migration protocol, illustrating the stages involved in migrating task T1 from host 1 to host 2.

These control messages, or CMs, are special system messages added to the *pvmds* and the pvmlib code for the purpose of managing task migration. Just like other system messages, these control messages are invisible to the application code. Examples of CMs is *SM_MIG* (SM stands for Scheduler Message and MIG stands for Migrate) sent by the GS to the *pvmd* on the host where the task to be migrated is currently executing. The migration of a task is triggered by a migration event. This event triggers the GS which determines whether or not tasks have to be migrated. If so, it also decides which tasks to migrate, to where and subsequently initiates the appropriate actions through CMs.

Upon receipt of an *SM_MIG* CM, the pvmd (on the host which holds the task to be migrated) verifies that the tid belongs to a locally executing task. Migration initialization is divided into two components which occur in parallel. The first component, local initialization, involves "priming-up" the task to be migrated for the state transfer. This involves flushing all messages in the TCP socket connections and closing the connections. It is necessary to flush the TCP socket connections to avoid losing any message that may be buffered. The second component, remote initialization, involves the creation of a "skeleton process" that will be the recipient of the state information to be transfered. The skeleton process provides the infrastructure to which process state can be transferred and which will eventually be executing in the context of the original task.

State transfer has three requirements: the source of the state, the recipient of the state, and the medium through which the state will be transferred. The first two components are satisfied by the task to be migrated and the skeleton process respectively. For the transfer medium, a TCP connection, to be established at process state transfer time, is used. For the TCP connection to be established, it is necessary

that the skeleton process have a TCP socket to which the original task can connect. State transfer involves capturing the process' state (text, data, stack, and processor context), transferring it to another host and reconstructing it.

After sending all the necessary state information to the skeleton process, the original process terminates. It is at this point where the original task is officially removed from the source host. The skeleton process after receiving the original task's state, assimilates it as its own. This assimilation of state is done by placing the received data and stack state in their appropriate place in the skeleton process' virtual address space. Before the skeleton process can re-participate as part of the application, it first has to re-enroll itself with its local pvmd using the *pvm_mytid()* routine. By re-enrolling with the PVM system, the skeleton process officially becomes an MPVM task, at the same time re-establishing its indirect communications route with the other tasks. As for the TCP connections that were closed prior to the state transfer, note that direct connections are established "on-demand" in PVM. That is, only when a message is first sent between two tasks (which have set their routing mode to use direct routing) is the TCP connection established. By closing down the TCP connections in such a way that the tasks involved "think" that there was never a connection, direct connections with the just-migrated task will automatically be re-established, using the protocol provided by PVM, once messages start flowing between them again.

Note however, that task migrations can happen only between homogeneous machine pools. By implementing the migration at user-level, state information managed by the OS kernel such as process IDs and pending signals cannot be automatically preserved on migration. Additional transparency problems appear if the task directly uses Unix facilities that depend on the location of the task. Examples of such facilities are shared memory, pipes, semaphores, sockets, and shared libraries.

5.3 Performance evaluation of loop parallelization on PVM

In this section, we present the performance results of our algorithm for loop parallelization (described in chapter 3) in a distributed memory environment using PVM. We show the speedup of our scheme over serial execution. A *self-scheduled* form of our executor with *barrier synchronization* is used for the comparison. A test loop structure as shown below (this is same as the loop structure in Figure 4.1, re-stated for the reader's convenience) is used for the experiments.

```
/* There are N iterations and r accesses per iteration. INDEX contains the actual
array references. tmp1 and tmp2 are dummy variables. */
    do i = 1,N,1
        do j = 1,r,1
            if(odd(j)) tmp1 = A[INDEX[i*r+j]];
            else A[INDEX[i*r+j]] = tmp2;
            do j = 1,W,1
                dummy loop simulating useful work
            enddo
        enddo
    enddo
```

Table 5.1 shows some of the parameters of the loop structure (Column 2) and what they measure (Column 3). Column 4 shows the range of values chosen. Array A has N∗r elements. The *Hfrac* and *Hsize* parameters are used to simulate different dependence patterns. Each access has a probability *Hfrac* of being a *hot* reference. The hot section is a portion *Hsize* of array *A*. Loops with long dependence chains, and therefore low parallelism, have high *Hfrac* and low *Hsize*. A *Mostly-Serial* loop is one with (*Hsize, Hfrac*) = (0.1%, 90%), a *Mixed* loop is one with (*Hsize, Hfrac*) = (10%, 90%) and a *Mostly-Parallel* loop is one with (*Hsize, Hfrac*) = (90%, 10%). Note that

the *Hfrac* and *Hsize* values for *Mostly-Serial* and *Mixed* loops are different compared to those used in the experiments on the shared memory machine. Also note the iteration grain size (W) is expressed as the number of workload loop iterations instead of the number of processor cycles. This is because the various nodes constituting the parallel virtual machine are heterogeneous, and it is difficult to compute a common workload (in terms of processor cycles).

Param.	Explanation	What We Want to Measure	Values
N	Iteration Count	Problem size scalability	1600, 3200, 6400, 12800,25600
Hfrac	Hot spot fraction	Consistency over dependence patterns	10%, 90%
Hsize	Hot spot size		0.1%, 10%, 90%
W	Iteration grain size	Consistency over grain size	10,000 iterations, 20,000 iterations, 40,000 iterations
r	# references per iteration	Consistency over number of accesses	1, 2, 4, 8

Table 5.1: Loop parameters used in the experiments.

The experiments are timing runs on a virtual machine consisting of 10 nodes. The physical distribution of the nodes is illustrated in Figure 5.6. There are two subnets, with all nodes in each subnet linked via 10 Mbit/sec Ethernet, most of which is thin-wire (or 10Base2). One of the nodes (labeled # 1 in Figure 5.6) acts as the router between the 2 subnets. The application was implemented using the master/slave computation model. The master task spawns and synchronizes all the slave tasks, distributes the workload among the slave tasks and does load balancing. The master task normally executes on the router node, to save on communication costs. The

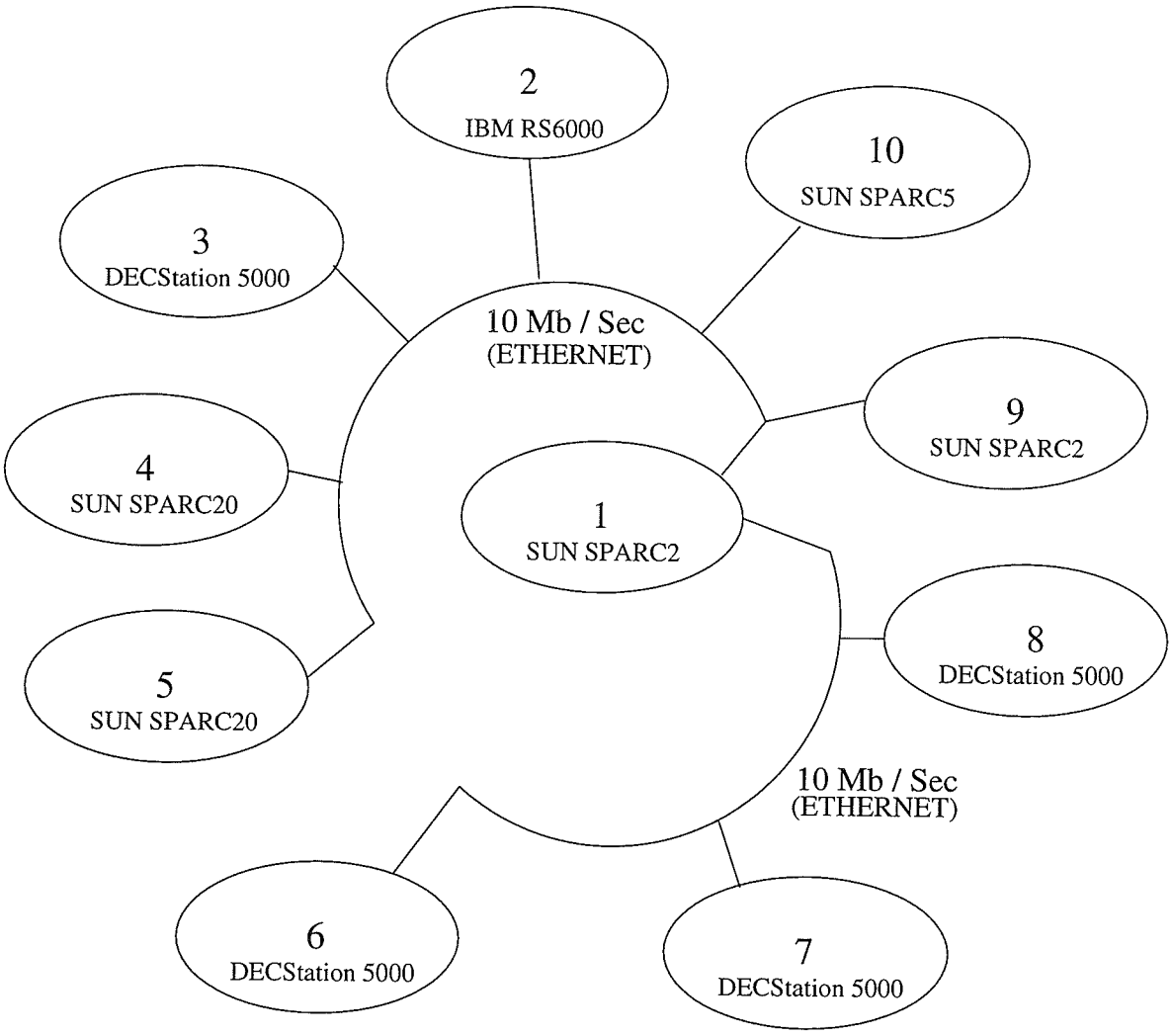

Figure 5.6: The physical distribution of the various nodes in the virtual machine.

slave tasks do the actual inspection and execution. The slave tasks are potentially distributed on all the available nodes/hosts. The master task intially spawns the slave tasks, sends initialization information and then distributes the inspection among the multiple slaves. After receiving the wavefront schedule from the slaves, the master again distributes the execution among the slave tasks using the wavefront schedule. This involves the master task sending the slaves synchronization messages as well as distributing the workload periodically.

W	r	**Mostly-Serial Loop** ($Hsize = 0.1\%$, $Hfrac = 90\%$)				
		N=1600	3200	6400	12800	25600
10 K	8	0.23	0.30	0.33	0.63	0.74
	4	0.21	0.31	0.34	0.68	0.80
	2	0.27	0.31	0.35	0.74	0.87
	1	0.26	0.33	0.35	0.68	0.88
20 K	8	0.25	0.32	0.37	0.82	0.89
	4	0.28	0.34	0.38	0.76	0.94
	2	0.29	0.33	0.38	0.90	0.98
	1	0.36	0.37	0.38	0.92	1.01
40 K	8	0.26	0.36	0.40	0.95	1.02
	4	0.32	0.36	0.40	1.02	1.04
	2	0.33	0.39	0.40	0.91	1.07
	1	0.36	0.37	0.39	1.06	1.07

(a)

W	r	**Mixed Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
10 K	8	0.89	1.01	1.35	1.38	1.42
	4	1.19	1.33	1.62	1.81	0.79
	2	1.31	1.70	1.04	1.98	2.39
	1	1.44	1.82	2.03	2.47	2.49
20 K	8	1.20	1.54	1.77	1.91	1.96
	4	0.77	1.87	2.19	2.34	2.38
	2	1.62	0.82	2.52	2.70	0.83
	1	1.75	2.27	2.63	2.78	2.89
40 K	8	1.58	0.81	2.20	2.36	2.51
	4	0.80	2.29	2.64	2.71	2.85
	2	1.85	2.37	2.73	2.99	3.08
	1	1.92	2.49	2.77	0.83	3.18

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
10 K	8	0.98	0.66	1.15	1.35	1.41
	4	1.33	1.29	1.73	1.87	1.87
	2	1.59	2.04	1.04	2.22	2.42
	1	1.69	0.78	1.01	2.29	2.57
20 K	8	1.43	1.59	1.81	1.91	1.90
	4	1.76	2.02	2.22	2.35	2.48
	2	2.03	1.76	2.57	2.73	2.68
	1	2.20	2.61	1.42	2.88	2.92
40 K	8	0.79	2.10	2.36	2.46	2.55
	4	1.99	2.53	2.30	2.85	2.79
	2	2.30	2.75	2.87	3.07	3.15
	1	2.14	2.74	2.71	3.12	3.23

(c)

Table 5.2: Speedup of our algorithm over serial execution including *inspector* and *executor* using 9 slave tasks.

Performance results pertaining to the speedup of the application on the virtual machine over serial execution are shown in Table 5.2. The computational system consists of a master process and 9 slave processes. The timings include every single piece

of essential computation. We use the same parallel inspector and executor approach. The executor is implemented using the *barrier synchronization* technique described in chapter 3. Iteration to processor scheduling is done at run-time (obviously), and *self-scheduling* is used to dynamically balance the load among the multiple slave tasks or nodes.

As expected, the best performance is achieved in the case of mostly-parallel loop structures. In the case of mostly-serial loops, the amount of inherent parallelism is much less, and as a result we do not see any speedup over serial execution. As the loop becomes more and more parallel in nature, the speedup increases and it is worth parallelizing the computation. The performance doubles when the number of slave tasks increases from one to two, but from then on, the performance increases only by a logarithmic factor of the number of slave tasks. During automatic parallelization of loop structures, an initial heuristic analysis of the available parallelism maybe useful in deciding whether to parallelize the computation or not.

The performance results obtained on the distributed memory machine are not very impressive compared to those on the shared memory machine. The reason is, in part, that in the case of the distributed memory implementation the serial runs were performed on a fast IBM RS/6000 RISC machine with relatively low workload. Furthermore, the speedups are not a fair relative performance measure between the shared memory and distributed memory models. Each one individually shows the speedup of the parallelized computation over serial execution on their own architecture.

In a heterogeneous environment, many other issues come into play, including node diversity, network bandwidth/latency, and dynamic network/machine load. Dynamic load balancing (process migration) maybe a useful technique to achieve improved

performance. Nothing of that nature was tried, and as a result quite a few processor cycles on faster nodes were wasted trying to synchronize with the computation on the slower nodes.

Due to the inherent bottlenecks of ethernet-based communication, there are some virtual limits on performance and scalability (even though the application maybe perfectly scalable, the existing network may result in reduced performance because of issues like bus contention, etc). Still, by intelligently placing the master/slave tasks on appropriate nodes (like placing the master task on the router node), better performance can be achieved for a given configuration. With better network technology (like ATM - Asynchronous Transfer Mode), we can expect to see improved performance and scalability.

Chapter 6

Conclusions and future directions

The automatic parallelization of loops is usually based on compile-time analysis of data dependencies. In some loops, however, the data dependencies cannot be determined at compile time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, run-time analysis is needed. In this thesis, we presented and evaluated a new algorithm to parallelize these loops at run-time. Our scheme handles any type of data dependence pattern without requiring any special architectural support. It minimizes inter-processor communication, leads to better processor utilization and allows reuse of the inspector results across several loop invocations. Comparison of the new algorithm with an older scheme with the same generality shows that the new scheme offers consistent performance (i.e., performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and leads to higher speedups during the executor stage.

We have evaluated our algorithm with a set of parameterized loops running on an 8-processor SGI shared memory multiprocessor. We used loops with varying parameters, such as the number of iterations and references. The results show better speedups and they are scalable to higher number of processors. We also presented the performance of the algorithm on a collection of distributed networked heterogeneous

workstations using PVM. The algorithm seems to be scalable to a larger number of nodes with adequate network support (to overcome network latency). There are some issues involved in making a decision on whether to parallelize the computation of an application or not. In particular, one has to make a trade-off between the computational workload and the cost of distribution/parallelization.

A parallel inspector may result in a non-optimal iteration schedule compared to its serial counterpart, since the number of wavefronts produced in the case of a parallel inspector is more by a factor of the number of concurrent tasks involved in the inspection. This does not affect the performance of the executor significantly in the case of Mostly-Parallel loops (only very slightly due to the additional communication required for the extra wavefronts), since there is enough work to do per task in any given wavefront. Further, the performance can be improved by overlapping the execution (*busy-wait* mechanism) of multiple wavefronts. This does not affect the performance of the executor in the case of Mostly-Serial loops either, since the loop has lots of serial points (which would have resulted in a new wavefront anyway), and by appropriately choosing a serial point for partitioning during *Sectioning*, generation of additional pseudo-wavefronts can be avoided.

So, the real decisions to be made while parallelizing the computation are:

- how many slave tasks to spawn (in the case of mostly-serial loops) depending on the number of iterations per wavefront, since too many tasks with too few iterations per wavefront will result in poor performance.

- In the case of Mostly-Parallel loops, whether to perform serial inspection or parallel inspection followed by *bootstrapping*, to obtain optimal parallel schedule for execution. It may also be true, that the sub-optimal iteration schedule formed by the parallel inspector may not be really hindering the performance

in a significant way.

- choosing appropriate serial points for wavefront formation (in the case of mostly-serial loops) during parallel inspection, to avoid adding additional pseudo-wavefronts.

Future work

Ways to improve the sub-optimal iteration schedule produced by the parallel inspector and making use of static (compile-time) dependence information during run-time dependence analysis are excellent areas for future research. There are also many optimizations that can be attempted in the case of the distributed memory implementation:

- Attempting dynamic process migration is a good strategy to obtain increased performance in a heterogeneous distributed environment, when there is a mixture of fast and slow nodes.

- Partition the workload to be distributed unevenly, based on the computational power of the individual nodes.

- An *a priori* knowledge of the existing network connectivity and communication support may be very useful in gaining performance. For example, in the case of ATM networks, implementing the communications between the nodes using a specialized API (Application Programming Interface) may be helpful in realizing better performance.

Bibliography

[1] J. Archibald and J. Baer, *Cache coherence protocols: evaluation using multiprocessor simulation model*, ACM Transactions on Computer Systems, 4(4):273-298, November 1986.

[2] U. Banerjee, *Dependence analysis for supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[3] D.-K. Chen, J. Torrellas and P.-C. Yew, *An efficient algorithm for the run-time parallelization of DOACROSS Loops*, Proceedings of Supercomputing 1994.

[4] D.-K. Chen and P.-C. Yew, *A scheme for effective execution of irregular DOACROSS loops*, in Proceedings in Int'l Conf. on Parallel Processing, pages 285-292, August 1992.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[6] Henri Casanova, Jack Dongarra, and Weicheng Jiang, *The Performance of PVM on MPP Systems*, Univ. of Tennessee Technical Report CS-95-301, August 1995.

[7] Honbo Zhou and Al Geist, *Faster Message Passing in PVM*, Proceedings in 9th Intel Parallel Processing Symposium: Workshop on High-Speed Network Computing, Santa Barbara, April 1995.

[8] Jeremy Casas, Dan Clark, Ravi konuru, Steve Otto, Robert Prouty, Jonathan Walpole, *MPVM: A Migration Transparent Version of PVM*, Tech Report No. CSE-95-002, Dept. of Computer Science and Engineering, Oregon Graduate Institue of Science & Technology, Portland, Oregon, USA.

[9] J. A. Kohl, G. A. Geist, *XPVM 1.0 User's Guide*, Computer Science and Mathematics Division, ORNL/TM-12981, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.

[10] V. P. Krothapalli, *On synchronization and scheduling for multiprocessors*, Ph.D thesis, The Ohio State University 1990.

[11] V. P. Krothapalli and P. Sadayappan, *An approach to synchronization for parallel computing*, in Proc. 1988 Conf. on Supercomputing, St. Malo, 1988.

[12] V. P. Krothapalli and P. Sadayappan, *Dynamic scheduling of DOACROSS loops for multiprocessors*, IEEE Parallel Architectures 1991.

[13] V. P. Krothapalli, J. Thulasiraman and M. Giesbrecht, *Run-time Parallelization of irregular DOACROSS loops*, in in Parallel algorithms for irregularly structured problems, September 4-6, 1995, Lyon, France : Proceedings of Irregular '95, LNCS 980, pp.75-80, Springler-Verlag publishers, 1995.

[14] D. Kuck et al, *The Cedar system and an initial performance study*, in 20th Int'l Symp. on Computer Architecture, May 1993.

[15] S.-T. Leung and J. Zahorjan, *Improving the performance of run-time parallelization*, in 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pages 83-91, May 1993.

[16] S. Midkiff and D. Padua, *Compiler algorithms for synchronization*, IEEE Trans. on Computers, C-36(12), December 1987.

[17] J. K. Ousterhout, *An X11 Toolkit based on the Tcl language*, 1991 winter USENIX conference.

[18] Philip J. Mucci and Jack Dongarra, *Possibilities for Active Messaging in PVM*, University of Tennessee Technical Report CS-95-277, February 1995.

[19] D. A. Padua, *Multiprocessors: Discussion of some theoretical and practical problems*, Ph.D thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1979.

[20] C. Polychronopoulos, *Advanced loop optimizations for parallel computers*, In E. Houstis, T. Papatheodorou, and C. Polychronopoulos, editors, *Lecture Notes in Computer Science No. 297: Proc. First Int'l Conf. on Supercomputing, Athens, Greece*, pages 255-277, New York, June 1987, Springler-Verlag.

[21] J. Saltz, R. Mirchandaney, and K. Crowley, *Run-time parallelization and scheduling of loops*, IEEE Trans. Computers, 40(5):603-612, May 1991.

[22] Thorsten von Eicken, David E. Cullerand Seth Copen Goldstein, and Klaus Erik Schauser, *Active Messages: a mechanism for integrated communication and computation*, Proceedings of the 19th International Symposium of Computer Architecture, May 1992.

[23] J. Thulasiraman, V. P. Krothapalli and M. Giesbrecht, *Run-time parallelization of irregular DOACROSS loops*, June 5, 1995, Technical Report No. 95/03, Department of Computer Science, University of Manitoba, Winnipeg, Canada.

[24] J. Thulasiraman, M. Giesbrecht and P. Graham, *PVM: an overview and evaluation of loop parallelization on PVM*, December 1995, Technical Report No. 95/03, 28 pp., Department of Computer Science, University of Manitoba, Winnipeg, Canada.

[25] T. H. Tzen and L. Ni, *Dependence uniformization: A loop parallelization technique*, IEEE Trans. on Parallel and Distributed Systems, 4(5), May 1993.

[26] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[27] M. J. Wolfe, *Optimizing compilers for supercomputers*, Ph.D thesis, University of Illinois at Urbana-Champaign, 1982.

[28] M. J. Wolfe, *Loop skewing: The wavefront method revisited*, Int. J. of Parallel Programming, 4(15), 1986.

[29] C.-Q.Zhu and P.-C.Yew, *A scheme to enforce data dependence on large multiprocessor systems*, IEEE Trans. on Software Engineering, June 1987.