Task Level Parallelization of Irregular Computations using OpenMP 3.0

by

Eid Albalawi

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

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
December 2013

Thesis advisor

Dr. Parimala Thulasiraman

Dr. Ruppa Thulasiram

Author

Eid Albalawi

Task Level Parallelization of Irregular Computations using OpenMP 3.0

Abstract

OpenMP is a standard parallel programming language used to develop parallel applications on shared memory machines. OpenMP is very suitable for designing parallel algorithms for regular applications where the amount of work is known apriori and therefore distribution of work among the threads can be done at compile time. In irregular applications, the load changes dynamically at runtime and distribution of work among the threads can only be done at runtime. In the literature, it has been shown that OpenMP produces unsatisfactory performance for irregular applications. In 2008, the OpenMP 3.0 version introduced new directives and features such as the "task" directive to handle irregular computations. Not much work has gone into studying irregular algorithms in OpenMP 3.0. In this thesis, I provide some insight into the usefulness of OpenMP 3.0 for irregular problems.

Contents

Abstract ii
Table of Contents iv
List of Figures v
List of Tables vi
Acknowledgments vii
Dedication viii

1 Introduction and Background **1**
1.1 Thesis Overview 5

2 OpenMP 3.0 **7**
2.0.1 Task directive 10
2.0.2 The OpenMP 3.0 collapse feature 14

3 Breadth-First Search **16**
3.1 Introduction and Background 16
3.2 Related work 20
3.3 My implementation 23
3.4 Results 27

4 All-Pairs Shortest Path **30**
4.1 Introduction 30
4.2 Related work 33
4.3 My implementation 34
4.4 Results 35

5 Minimum Spanning Tree **38**
5.1 Introduction 38
5.2 Related work 42
5.3 My implementation 43
5.4 Results 45

6 Conclusions and Discussion **47**

7 Future work **51**

Bibliography **53**

List of Figures

2.1 Directive and Clauses 8

3.1 Breadth-First Search traversal 17
3.2 Graph example 24
3.3 Asynchronous Breadth-First Search traversal 25
3.4 Remove the inefficient path 25
3.5 BFS execution Time for SSAC#2 29
3.6 BFS execution Time for R-MAT 29

4.1 APSP execution Time for SSAC#2 36
4.2 APSP execution Time for R-MAT 37

5.1 MST execution Time for SSAC#2 46
5.2 MST execution Time for R-MAT 46

List of Tables

3.1 The BFS execution time on SSAC#2 27
3.2 The BFS execution time on R-MAT 27

4.1 The APSP execution time on SSAC#2 36
4.2 The APSP execution time on R-MAT 36

5.1 The MST execution time on SSAC#2 45
5.2 The MST execution time on R-MAT 45

Acknowledgments

Foremost, Thank you Allah first and last. Thank you Allah for everything you gave me and for everything you did not give me, for everything you protected me from that which I know and that which I am not even aware of. Thank you for blessings that I did not even realize were blessings. Without your blessings, I would not completed my thesis. Thank you for your guidance when I felt like I was slipping. Thank you for everything else, because no matter how many things I try to list, at the end of the day, I can not even come close to thank you enough.

I would like to express the deepest appreciation to my advisors, Dr. Parimala Thulasiraman and Dr. Ruppa Thulasiram, for their attitude and able guidance during my study in University of Manitoba. My sincere gratitude for their patience, encouragement, motivation and continues support of my M.Sc. Study.

A special thanks to my mother "Aziza Albalawi"and to my sisters "Dr. Halah, Modiy and Gadah Albalawi" and my brothers "Nader and Abdullah Albalawi" for their emotion support during my journey here. Without their support and help, this thesis would not be possible.

In addition, I want to thank "Omar Alhazmi", "Rashed Alqahtani", "Mohammed Alluhybi" and "Bilal Ragoub" for being my best friends and share some unforgettable memories here in Canada.

Finally, I would like to thank graduate students at the Department of Computer Science for being friendly, helpful and supportive.

This thesis is dedicated to in loving memory of my father, " Mohammed Albalawi". I know you would be proud of me now.

Chapter 1

Introduction and Background

Homogeneous multicore architectures have been used widely in the past decade. This is due to the need of having machines with high performance that are more powerful computationally than uniprocessor machines. In a homogeneous multi-core architecture, many identical processors or cores work together to perform complex tasks. Companies such as Intel, have moved toward increasing the processor's power by adding more cores on a single chip. Most of the commodity homogeneous architectures have many duplicated CPUs on a single chip with a shared memory. The different CPUs interact with each other through shared variables.

Shared memory machines can be categorized as either Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) architectures. In UMA machines, the CPUs have the same access time to a shared primary memory. On the other hand, each CPU in NUMA has its own memory. This memory can be accessed by the CPU that it belongs to or , more slowly, by other CPUs. The memory access time is, therefore, non-uniform. Modern homogeneous multicore architectures with a

shared memory system are also multithreaded. The cores have the capabilities of handling several threads concurrently. For example, Sun Niagara's processor with eight cores can execute 32 threads concurrently and supports fine-grained multithreading. The Intel Nehalem microarchitecture that encompasses the Core i7 class of processors provide hyperthreading support in hardware and also introduced a distributed shared memory architecture using the Intel QuickPath Interconnect (QPI). These architectures exploit both instruction level parallelism and thread level parallelism.

There are many parallel programming languages or APIs that support a shared memory paradigm. One such API is OpenMP (OpenMP [1998]). OpenMP specifies a set of compiler directives and provides associated libraries to execute specific code sequences in parallel and to divide the work among threads. OpenMP employs a *fork-join* paradigm. The program starts with one thread called the master thread. Then, whenever there is a parallel region in the program, the master thread invokes a set of slave threads and distributes the work among them. This operation is called *fork*. After forking, the threads are allocated to the processors by the runtime environment and work concurrently to solve the problem. Once the slave threads have completed their work, they are destroyed and the master thread continues until it encounters another parallel region. This operation is called *join*.

OpenMP has been used extensively for regular applications. The data structures used in these problems are structured (such as an array). The program flow and memory access patterns are also very structured and are generally known apriori. An example of a regular problem is matrix-vector multiplication, where A is a dense matrix, x is a vector and b is the resultant vector. In this example, the computations

or operations required to produce the output and the data access patterns are known beforehand. On a multi-processor system, each processor can be assigned the same vector x with a certain number of data elements (a row or a given number of rows)to compute an element(s) in b. All processors perform the same computations to produce the resultant vector but with different data sets. As a result, these problems can be optimized to run on any type of architecture relatively easily. These problems are also classified as *data parallel* applications.

The same is not true for irregular applications. Irregular applications rely on pointer or graph-based data structures. Algorithms that are used to solve irregular applications are referred to as irregular algorithms. Graph problems, list ranking and unstructured grid problems are common examples of irregular computations. In such computations (Zhang and Torrellas [1995]; Secchi et al. [2012]; Kulkarni et al. [2007]; Nieplocha et al. [2007]), the data size changes dynamically at runtime, leading to non-uniform memory access patterns and higher and unpredictable communication/synchronization latencies. The load or amount of work to be distributed to the threads is not known apriori. We could consider the matrix-vector multiplication as an irregular problem, if A is a sparse matrix. Since A is instance specific, the structure of A is unknown at compile time. A matrix is not necessarily the correct data structure to use since there may be many 0's in the matrix wasting memory resources. Instead we might use in such problems, accesses to data often have poor spatial and temporal locality leading to ineffective use of the memory hierarchy (Zhang and Torrellas [1995]).

It is important to find efficient solutions in solving irregular problems. Irregular

adaptive methods (Biswas and Strawn [1994]; Kulkarni et al. [2007]), for example, have applications in many Science and Engineering problems. With multicores becoming very popular, having a standard programming language that addresses both irregular and regular applications is very desirable. OpenMP is one such language. The literature has been shown eg.(Süßand Leopold [2006]; Dedu et al. [2000]; Hisley et al. [1999]) that OpenMP results in reduced performance when dealing with irregular computations..

Several researchers have tried to overcome this issue (Wicaksono et al. [2011]; Labarta et al. [2001]; Wang et al. [2007]; Kulkarni et al. [2008]) by developing efficient algorithms for irregular application in OpenMP by developing automatic parallelling compilers or by introducing new directives in OpenMP to handle irregular applications.

The earlier versions of OpenMP were not meant to handle irregular computations (Mattson [2003]). In 2008, the OpenMP 3.0 version introduced a directive called "task" to help develop parallel algorithms for irregular applications. The directive "task" creates independent work units to be executed. The task in OpenMP 3.0 is nothing but a thread that can be created and destroyed as needed. It can also spawn other tasks that is not possible in the previous versions of OpenMP. Spawning threads this way allows dynamic creation of threads incorporating fine grained parallelism and exploiting load balancing at runtime which is important for performance improvement in irregular computations.

1.1 Thesis Overview

Irregular problems are very challenging to solve on conventional parallel computers. On shared memory machines, OpenMP has been regarded as the standard API to develop parallel algorithms. Earlier versions of OpenMP are not meant to handle irregular computations. Version OpenMP 3.0 contains new features to tackle irregular applications.

In this thesis, I consider three classic graph algorithms: All-Pairs Shortest-Path (APSP), Breadth First Search (BFS) and Minimum Spanning Tree (MST). These algorithms are described in Chapters 3, 4 and 5, respectively. An overview of OpenMP 3.0 is provided in Chapter 2.

The algorithms were implemented on the AMD Accelerated Processing Unit (APU) 8 quad-core machine. The clock speed of each core is 3.0MHz. There is 4GB of RAM memory. I used the GCC4.4 compiler to compile and run the algorithm. I implemented the algorithms on two types of graphs:

- R-MAT graphs: These are random graphs (Chakrabarti et al. [2004]) allowing high and low degree vertices.

- SSCA#2 graphs: Graphs in this category (Bader and Madduri [2005])have high connected cliques. The size of the cliques is distributed uniformly. Then, they generate inter-clique edges with a chosen probability.

In the literature, most of the work using the OpenMP 3.0 task directive has focused on implementing recursive algorithms such as Fibonacci sequences or merge-

sort. These problems create an easy parent-child relationship and can be easily implemented in using the task directive.

There were many challenges in using task level parallelism for graph algorithms. Graph algorithms are pointer-based data structures and dynamically change at runtime. Tasks (or threads) are created at runtime leading to load balancing issue. This is one challenge. Second is that algorithms are not necessarily recursive and may not reveal a parent-child relationship adaptable to task parallelization. In OpenMP we start with the sequential algorithm, and modify the algorithm to work in parallel by adding parallel directives, that is, it allows incremental parallelization. This is not the case with the task directive. The input to the problem considered in this thesis is a random graph. Therefore, task level parallelization is not straightforward and algorithms have to be modified to take advantage of OpenMP task directives. In breadth-first search algorithm, I developed an asynchronous task level parallelism algorithm using the ideas from (Chandy and Misra [1982]). The MST was more challenging. There was a sequential part and parallel part to Borůvka's MST algorithm Borůvka [1926]. I was able to apply task level parallelism to the parallel part but due to the sequential nature of the other half of the algorithm, I implemented the merge phase of Borůvka's algorithm sequentially. This idea is similar to those of others (Chung and Condon [1996]) who have studied parallel Borůvka's algorithm.

Finally, Chapter 5 summarizes and discusses the results of the three algorithms and provides my experiences with OpenMP 3.0 for irregular problems. The thesis concludes with some future work in Chapter 6.

Chapter 2

OpenMP 3.0

OpenMP directives start with *#pragma omp* followed by the directive name. The syntax of OpenMP directives is as follows:

```
#pragma omp directive name [clause[[,]clause]...]
```

There are many directives in OpenMP. For example, the "parallel" directive supports Single Program Multiple Data (SPMD) model, a more coarse-grained approach, while "parallel for" supports fine-grained parallelism. The "section" directive provides support for functional level parallelism. The "single" directive could be used to declare instructions to be executed by only one thread.

A variable in OpenMP maybe be characterized as either "private" or "shared". A shared variable can be accessed by all threads while a private variable is accessible only to the thread that declared it. By default all variables are shared unless declared otherwise. Therefore, the scope of the variable is important in OpenMP.

Figure 2.1 (Dagum and Menon [1998]) shows which clauses are allowed with which directives. OpenMP language constructs can be categorized as follows: control flow

Directive	Clauses
For	firstprivate, lastprivate, nowait, private, reduction, schedule
Parallel	firstprivate, lastprivate, if, private, reduction
Parallel for	firstprivate, lastprivate, if, private, reduction, schedule
Section	firstprivate, lastprivate, nowait, private, reduction
Single	firstprivate, nowait, private

Figure 2.1: Directive and Clauses

directives, sharing variables, synchronization and runtime functions.

Control flow: Directives in this category control the flow of the program. *parallel* and *single* are the most common directives that follow this category.

Sharing variables: Clauses such as $firstprivate$, *lastprivate* and $default$ are in this group. They specify the visibility of the variables to threads. As mentioned, all variable are shared by default. Therefore, synchronizing variables appropriately is important.

Synchronization: Major directives in this category are *atomic*, *critical*, *barrier* and *taskwait*.

Runtime functions: There are also some predefined functions in the OpenMP library such as *omp_set_num_thread* and *omp_get_num_thread*.

OpenMP normally, provides *incremental parallelization*. That is, it allows a programmer to work with sequential code and incrementally change the blocks of code

one at time to parallelize the code by adding parallel directives. We can easily remove the parallel directives to get a sequential code or compile with a non-OpenMP aware compiler. The earlier versions of OpenMP were very restrictive and the system was optimized to support regular applications.

In 2008, the OpenMP version 3.0 introduced a directive called "task" to help with developing parallel algorithms for irregular applications. The directive "task" creates independent work units to be executed by threads. The task in OpenMP 3.0 combines code and an associated data environment (Ayguadé et al. [2009]). All tasks are either executed immediately if they are ready or postponed to a later time. Tasks that are deferred are placed in a task pool. Threads pick up each task to execute from this task pool until there are no more threads in the task pool. Tasks are either *tied* or *untied*. Tied implies the task will be processed by one thread. If untied, more than one thread maybe be allowed to process different parts of the code. Initially, all tasks are tied.

In OpenMP 3.0, tasks can spawn other threads which was not possible in the previous versions of OpenMP. Spawning threads allows dynamic creation of threads incorporating fine grained parallelism and exploiting load balancing at runtime which is important for performance improvement in irregular computations.

Another new feature is "collapse". By adding "collapse" to a nested loop, it combines into a single loop with fewer fork and join.

2.0.1 Task directive

The tasking model was proposed in (Ayguadé et al. [2007]). As mentioned earlier, a task has its own data environment besides its code. A task can be characterized as wither explicit or implicit. When threads encounter a *parallel* region, a set of implicit tasks will be generated and one task will be assigned to each thread. These implicit tasks are therefore *tied* to each thread. Explicit tasks may be generated by other tasks. Tasks execution can be synchronized using the *taskwait* directive. The syntax for the task directive is as follows :

```
#pragma omp task [clause[[,]clause]...]
```

where the clause could be *shared, private, firstprivate or lastprivate.*

In previous versions of OpenMP, although task parallelism was implicitly provided through the *section* directive, explicit programmer-specified parallelism did not exist. Also, incorporating task parallelism in previous versions of OpenMP, might not actually provide any performance improvement due to synchronization barriers.

Consider the code example in Listing 2.1:

Listing 2.1: Sequential code for tree traversal

```
void traverse (Tree *tree)
{
if (tree->left)
traverse(tree->left);
if (tree->right)
traverse(tree->right);
process(tree);
```

```
}
```

One way to parallelize this code is to use parallel regions as shown in Listing 2.2:

Listing 2.2: Parallel tree traversal with OpenMP 2.5

```
void traverse (Tree *tree)
{
#pragma omp parallel sections num_threads(2)
{
#pragma omp section
if (tree->left)
traverse(tree->left);
#pragma omp section
if (tree->right)
traverse(tree->right);
}
process(tree);
}
```

In OpenMP, *#pragma omp parallel section* is called a work-sharing construct. It precedes a sequence of k blocks of code that may be executed concurrently by k threads. Different blocks will be executed by different threads. There is an implied barrier for synchronization at the end of parallel sections. It is possible for a thread to execute more than one section if it is quick enough and the OpenMP implementation permits it.

In Listing 2.2, there are two independent "section" directives that are nested within a "sections" directive. The enclosed section(s) of code are to be divided among the threads in the team. Each section in Listing 2.2 is executed once by a thread in the team. Since there are two sections, one thread is assigned to each section. In this program due to the use of recursion, there will be too many parallel regions which adds extra overhead and extra synchronization (Ayguadé et al. [2009]). Also, sections cannot be nested and therefore not manual.

The same program can be done more efficiently in OpenMP 3.0 as shown in the code fragment in Listing 2.3

Listing 2.3: More efficient parallel tree traversal with OpenMP 3.0

```
void traverse (Tree *tree)
{
if (tree->left)
#pragma omp task
traverse(tree->left);
if (tree->right)
#pragma omp task
traverse(tree->right);
process(tree);
}
```

The program in Listing 2.3 works much better than the code in OpenMP with many parallel regions. Here, we are not restricted to only two threads as with the code in

Listing 2.2. A thread is assigned to a task (with code and data). Each task produces more tasks dynamically at runtime. Therefore, a thread that executes a task package is a new instance of a task. The new tasks can be executed immediately by the same thread or be executed at a later time by some other thread in the team. Threads can also suspend the execution of a task and resume execution at a later time.

Despite the parent-child relationship between tasks, a parent task can finish execution even before the child task has finished. This can be prevented by adding a synchronization primitive. The code in Listing 2.4 depicts how to use the *taskwait* directive to achieve this effect.

Listing 2.4: Using synchronization directive

```
void traverse (Tree *tree)
{
if (tree->left)
#pragma omp task
traverse(tree->left);
if (tree->right)
#pragma omp task
traverse(tree->right);
#pragma omp taskwait
process(tree);
}
```

Please note that tasks can be nested inside parallel regions, inside other tasks and inside worksharings.[1]. Therefore, tasks are composable. In the above example, tasks are "tied" by default. Each task is always executed by the same thread. This restriction can be lifted by declaring tasks as "untied". This allows the tasks to migrate between threads at any point. Although, this may increase performance, programming is more complicated as the data scope must be carefully handled by the programmer to avoid unsafe sharing of data.

2.0.2 The OpenMP 3.0 collapse feature

"Collapse" is a new feature in OpenMP 3.0. This feature gives programmers more control to parallelize nested loops. Programmers just need to add "collapse" after *#pragma omp for*. The syntax is as follows:

```
#pragma omp for ... collapse(N) ...
```

where N is the number of loop iterations that we want to collapse. Consider the code in Listing 2.5:

Listing 2.5: Nested loops

```
for (i=0; i<N; i++)
 for(j=0; j<k; j++)
  work(i,j);
```

There are many ways to parallelize this nested loop with previous OpenMP versions. One way is to parallelize the second loop. The code in Listing 2.6 shows one way to

[1] worksharings constructs, such as # omp for, allows the compiler to subdivide works among the threads in an SPMD model

parallelize this nested loop in previous versions of OpenMP:

Listing 2.6: Parallelizing nested loop with OpenMP 2.5

```
for (i=0; i<N; i++)
#pragma parallel for
 for(j=0; j<k; j++)
  work(i,j);
```

The program starts with $i = 0$. Forks k iterations and each iteration j is executed by a thread for work(i,j). At the end, there is a join and the for is executed again. There are $N-1$ forks and joins in total. OpenMP 3.0 overcomes this issue by adding the collapse feature. The code Listing 2.7 illustrates this:

Listing 2.7: Parallelizing nested loop with OpenMP 3.0

```
#pragma parallel for collapse(2)
for (i=0; i<N; i++)
 for(j=0; j<k; j++)
  work(i,j);
```

Now, these two loops in Listing 2.7 are collapsed into one single loop and the iterations are divided efficiently among the threads.

Chapter 3

Breadth-First Search

Breadth-first search is one of the well-known graph traversal techniques in graph theory. Moore (Moore [1959]) developed the BFS algorithm while trying to find the shortest path inside mazes. BFS works by dividing the vertices of the graph into multiple levels. In this chapter, a brief introduction to the BFS algorithm is given, followed by my parallel solution using task level parallelism. The implementation details and results are discussed towards the end of the chapter.

3.1 Introduction and Background

For a given connected graph $G = (V, E)$, V denotes the set of vertices in a graph G, and E denotes the set of edges in G. BFS starts at the root vertex (level 0) and explores all the neighbouring nodes connected to the root. These neighbouring nodes are placed in level 1 (Figure 3.1). The algorithm then moves to these neighbouring nodes in level 1 and explores all their unexplored neighbouring nodes, and places

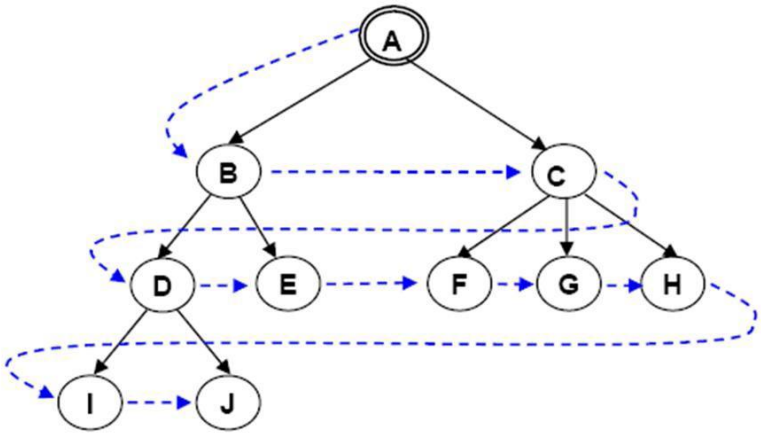

Figure 3.1: Breadth-First Search traversal

these nodes in level 2. The algorithm continues until it finds what it is looking for.

The sequential BFS algorithm is shown in Algorithm 1.

Algorithm 1: BFS Algorithm

Input: $G = (V, E)$

begin

    enqueue(Q, u) `/* Insert the root */`

    while *! empty(Q)* **do**

        dequeue(Q, v)

        for *each vertex u connected to v* **do**

            visit u

            enqueue(Q, u)

The BFS algorithm can also be used to find the shortest path from the root (source) to all the vertices in the graph. That is, it finds a path from the source vertex to all the vertices in the graph.

Here, besides having a boolean variable that indicates whether the vertex has already been visited, another variable, called *pred* is needed to store or to record the predecessor vertex (parent vertex) of the currently visited vertex. This predecessor information can be used to backtrack from a vertex to the source to determine the path and length of the shortest path.

The sequential algorithm for finding the shortest path using BFS is given in Algorithm 2.

Algorithm 2: BFS Algorithm finding shortest path

Input: $G = (V, E)$

begin

$visited(s) \longleftarrow True$

for *each vertex* $v \in V$ **do**

$visited(v) \longleftarrow false$

$pred(v) \longleftarrow -1$

Enter s into empty queue Q

while *! empty(Q)* **do**

$dequeue(Q, v)$

for *each vertex u of v* **do**

$visited(u) = true$

$pred(u) = v$

From Figure 3.1 and Algorithm 2, it can be seen that the BFS algorithm is synchronous. That is, each level needs to be explored first before processing the next level. Although, the nodes within a level can be explored in parallel, there is syn-

chronization between levels.

One way of implementing the parallel BFS in OpenMP 2.5 is as follows. One thread examines the source vertex. The source vertex's neighbouring vertices are placed in a queue data structure. These are the vertices at level 1. The vertices may be assigned to each thread or the work or computations of the vertices may be shared among the threads depending on the number of nodes in the queue, which changes dynamically at runtime. I call the vertices in the queue "active vertices". The threads working on the active vertices may in turn place the unexplored neighbouring vertices in the queue (level 2). After the threads have finished executing the active vertices, they synchronize before considering the next set of vertices in the queue. This process continues until there are no more vertices in the queue.

The Algorithm 3 shows the high level OpenMP 2.5 implementation of BFS.

Algorithm 3: Parallel BFS Algorithm in OpenMP 2.5

Input: $G = (V, E)$

begin

    enqueue(Q, u)

    while *! empty(Q)* **do**

        dequeue(Q, v)

        for *each vertex u connected to v* **do in parallel**

            visit u

            Synchronization barrier

            enqueue(Q, u)

First, as can be seen there is synchronization overhead at each level. This leads to many forks and joins. Also, there may be a data race problem if two threads

executing on two active vertices encounter the same unexplored neighbour. Here, again, synchronization is required to control access to the shared data.

To avoid unnecessary synchronization, in this thesis, I develop an asynchronous version of BFS using the ideas from Chandy and Misra (Chandy and Misra [1982]). Chandy and Misra designed a theoretical distributed algorithm for finding single source shortest paths. Their algorithm is asynchronous and iterative. In an asynchronous algorithm, we can avoid level synchronization by allowing tasks to be created and destroyed at runtime. However, termination detection in an asynchronous algorithm is difficult. In Chandy and Misra [1982], the authors propose an interesting idea for detecting termination in a distributed environment for the single source shortest path problem.

The asynchronous BFS algorithm constructs a tree where each vertex in the tree can be regarded as a task. I implement this algorithm in OpenMP 3.0 using the task directive.

3.2 Related work

Bader and Madduri (Bader and Madduri [2006]) implemented BFS on a multithreaded architecture. Unlike previous implementations that consider load balancing or distributed queue, they take advantage of the BFS characteristics. They used a concept called level synchronization. Level synchronization takes vertices that belong to a specific level to be processed in parallel. They keep track of all active vertices in the graph at a given level. They used their algorithm on a scale free graph which is representative of most real-world graphs. In this graph, the majority of the vertices

have small degree where the rest have large degree. Results show that their implementation takes less than 5 seconds with 400 million vertices and 2 billion edges of scale free graph and achieved speedup of 30 times on 40 processors.

Later, Leiserson and Schardl (Leiserson and Schardl [2010]) used the level synchronization concept also for the BFS problem using a data structure called bag data structure. The idea of a bag is to store an unordered set of nodes that belong to a level L_k. Therefore, a bag data structure can contain many such trees, each tree of size $2k$ where k is the index in the bag. To find nodes that belong to the next level L_{k+1}, the BFS algorithm performs a union-disjoint operation. The nodes in the bag (L_k) are first split into two disjoint bags with some fraction of the nodes in one bag and the rest in the current bag. The current bag is split again. This is done recursively creating several smaller bags. Each of the smaller bags are then traversed (visited) in parallel creating an output bag with nodes that belong to the next level L_{k+1}. Several such output bags are created. The union operation is performed on the resulting output bags to create single output bag with all nodes that belong to level L_{k+1}. Leiserson and Schardl proposed a linked list to implement the bag and implemented their code using Cilk++ (Blumofe et al. [1995]). In Cilk++, the threads manage two worksets, the first is the current workset and the other is the next workset. In Leiserson and Schardl's algorithm, the current workset has all the nodes of the current level where the next workset has all the nodes of the next level. Each thread takes one node from the current workset, explores its neighbours and places the unvisited neighbours into the next workset. Leiserson and Schardl study the complexity of their proposed data structure for the BFS algorithm. Their algorithm runs in $O((V+E)/P + D\lg^3(V/D))$

where P is number of processors and D diameter of the graph $G = (V, E)$.

Recently, John et al. (John et al. [2012]) implemented the BFS algorithm on the Freeze Breeze Program eXecution Model (PXM) (John et al. [2012]). The main contribution in the paper was to show the effectiveness of the model to exploit fine-grained parallelism in irregular applications. The Freeze Breeze PXM follows a dataflow model proposed by Dennis (Dennis [1975]). The model exploits fine-grained parallelism. That is, the basic unit of parallelism in the PXM model is a task. The tasks follow a spawn/join model. The master worker spawns many child workers which may in turn spawn more child workers. In this model, the master has no interaction with child workers and the child workers have no interaction with other workers in the system until the join phase. This allows threads to be created independently of one another so as many concurrent tasks as possible to exploit the parallelism available in the algorithm. The Freeze Breeze PXM also proposes a tree-structured global virtual memory model. The model is tested on some irregular computations such as BFS. John et al. used standard BFS algorithm that uses queues to store the visited nodes. They Implemented BFS on the Freeze Breeze PXM system without locking or atomic operations for synchronization. They used Graph 500 benchmark specifications (gra) with different graph sizes starting from 16 till 1024. The system was loaded with 40 processors. The system achieved a high performance peak that each processor able to handle on average 242,847 edges per second.

The BFS algorithm has been regarded as one of the simplest yet most interesting graph traversal algorithms that represents irregular computations. It has been used as a benchmark to examine new architectures or runtime systems. For example, an

article by Tumeo et al. (Secchi et al. [2012]) like (John et al. [2012]) shows the power of the Cray XMT multithreaded architecture for irregular applications through the BFS algorithm.

3.3 My implementation

BFS can be done in parallel by processing all the vertices that belong to the same level concurrently. However, for my solution, I developed an asynchronous parallel method to make effective use of the "task" directive. There is a one-one mapping between a vertex and a task. I designate v_i as vertex i and its corresponding task as T_i. I store the *current* distance and predecessor information of each of the vertices in global shared memory as a vector $(distance(v_1), distance(v_2), ..., distance(v_n))$. Note that for my BFS algorithm, each edge has unit distance. Similarly, for predecessor $(pred(v_1), pred(v_2), ..., pred(v_n))$.

The algorithm works as follows. Assume the algorithm starts at the source vertex v_0 with distance = 0. This vertex will be represented as a task that is, T_0. Note that a task can be encapsulated as an individual entity with its own data set (that is, a task T_i carries private data such as the distance $(distance_i)$and predecessor $(pred_i)$ information for the individual vertex). Since this is an asynchronous algorithm, a vertex may receive a shorter distance value from its neighbour at any point in time during the execution of the algorithm. If the vertex's distance information is updated with a shorter distance then the task will send the updated information out to its neighbouring nodes by spawning its neighbouring vertices as tasks. For example, consider the graph in figure 3.2.

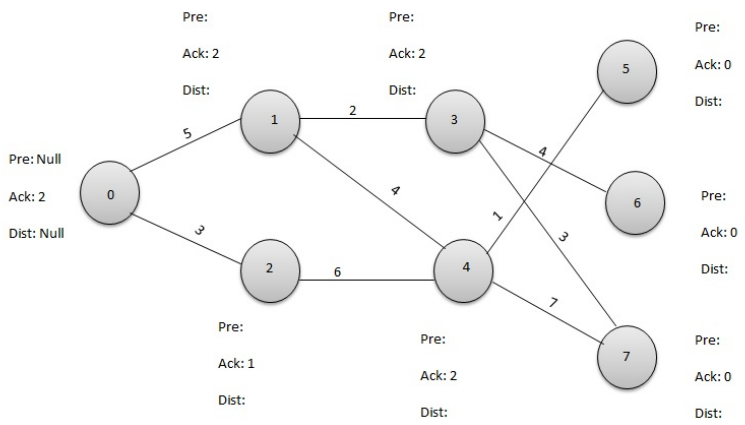

Figure 3.2: Graph example

Every vertex has three private variables which are *Pred*, *dist* and *Ack* which indicates number of the acknowledgement from each neighbour. The source vertex will explore its nighbour, then creates tasks for each nighbour and it updates its private information (*pred*, *dist* and *Ack*). It happens that the current distance is not the shortest distance. So, the vertex will check the shortest distance and when it finds the shortest distance, it will eliminate all other distances. Figure 3.3 shows that vertex 7 received 16 as a short distance from vertex number 4. However, this is not the shortest path. The shortest path is through vertex number 3 with 10 as a short distance. So, vertex 7 will update its *dist* to 10, changes *pred* to vertex 3 and eliminates the path to vertex 4. Figure 3.4 depicts this scenario.

The process will be the same for all vertices till we reach to the end of the graph. vertex will send back acknowledgment when *Ack* is 0 to its *pred*. The algorithm finishes when *Ack* of the source vertex is 0. The asynchronous parallel BFS algorithm

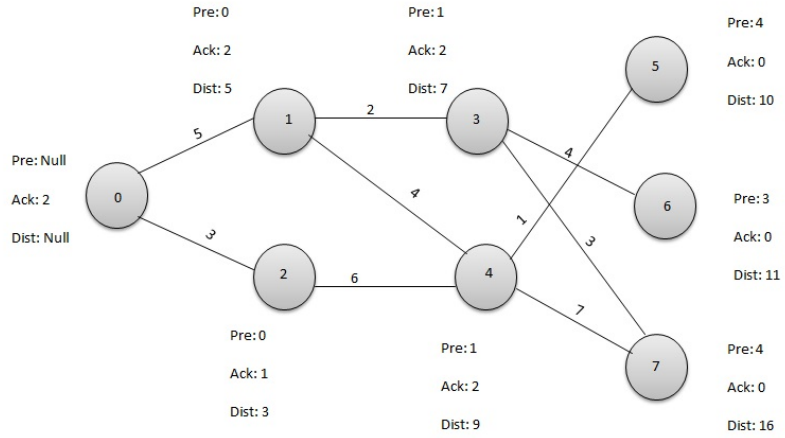

Figure 3.3: Asynchronous Breadth-First Search traversal

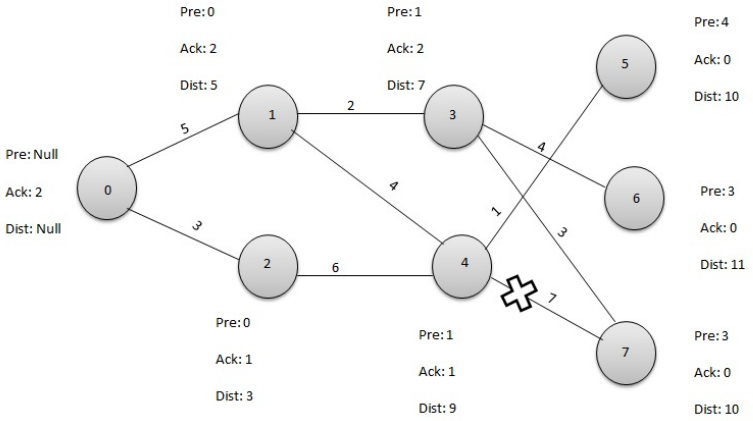

Figure 3.4: Remove the inefficient path

is shown in Algorithm 4.

Algorithm 4: Parallel BFS Algorithm

Input: $G = (V, E)$

begin

$distance(v_i) \longleftarrow \emptyset$

$Pred(v_i) \longleftarrow \infty$

$numack_i \longleftarrow 0$

for $v_i \in V$ **do in parallel**

Create task ()

if $Currentdistance < distance_i$ **then**

if $Numack_i \neq 0$ **then**

Send acknowledgment to $Pred_i$

$Numack = Numack - 1$

Synchronization barrier

$Pred(v_i = Parent_{v_i})$ `/* Update the predecessor */`

$distance_i = Currentdistance$ `/* Update the the distance */`

$Numack = Numack + numberofnieghborsofv_i$

if $Numack = 0$ **then**

Send acknowledgment to $Pred_i$

else

Send acknowledgment to $Pred_i$

3.4 Results

I used undirected graphs for our experiments. I start from 16 vertices and increase the number of vertices to 4096. I compare with OpenMP 2.5 and newer OpenMP 3.0 versions for both types of graphs. The implementation is in seconds.

Number of vertices	**OpenMP 3.0**	**OpenMP 2.5**
16	0.05	0.04
32	1.05	1.04
64	2.07	2.08
128	3.09	3.09
256	4.11	4.16
512	4.17	5.41
1024	4.38	5.28
2048	4.60	5.12
4096	5.43	7.80

Table 3.1: The BFS execution time on SSAC#2

Number of vertices	**OpenMP 3.0**	**OpenMP 2.5**
16	0.06	0.12
32	1.85	1.13
64	1.76	1.05
128	1.75	1.11
256	1.81	1.17
512	2.35	3.29
1024	3.62	5.49
2048	5.14	7.49
4096	6.15	10.31

Table 3.2: The BFS execution time on R-MAT

Table 3.1 (Figure 3.5) and Table 3.2 (Figure 3.6) represent the execution times for both OpenMP 3.0 and OpenMP 2.5 for the SSCA#2 and R-MAT graphs. In both versions, the execution time of both OpenMP 3.0 and OpenMP 2.5 increase with

respect to problem size. This is understandable due to increase in overheads such as communication or synchronization as problem size increases.

In OpenMP 3.0, for the SSCA#2 graph, the execution time for small problem size is relatively same. There is some difference in the execution times when we reach 2048 vertices. In the OpenMP 2.5 version, there is synchronization at each level and the active vertices in the queue are executed by a team of threads. For larger number of vertices, the vertices would be partitioned among the threads and thereby producing a coarse-grained implementation. However, due to synchronization overhead, there is an increase in execution time. On the other hand, the proposed asynchronous BFS algorithm exploits fine-grained parallelism. Each task is executed by one thread in the team as the data becomes available for the thread. The number of tasks generated increases or decreases relative to the graph size. However, fine-grained parallelism has not affected significantly the execution time for large graphs.

In the R-MAT graph, the execution times in OpenMP 3.0 increases compared to the execution times in OpenMP 2.5 until we reach 1024 vertices. This, I believe is due to the randomness of the graphs generated. Synchronization does not affect the algorithm running on OpenMP 2.5 for smaller graph sizes as we saw from the SSCA#2 graphs. Coarse-grained parallelism actually aids the performance of the algorithm in OpenMP 2.5. In OpenMP 3.0, fine-grained parallelism deters performance for small number of vertices, and in R-MAT this is exacerbated due to the randomness of the graph (low and high vertices are mixed). As the number of vertices is increased in R-MAT, the task-parallel algorithm performs better than the coarse-grained algorithm executed in OpenMP 2.5. This is due to the synchronization overhead associated

with the parallel BFS algorithm running on OpenMP 2.5. Fine-grained or task-level parallelism does not decrease performance of the algorithm. In the asynchronous algorithm, tasks overlap computation with communication and the algorithm reduces synchronization to a large extent.

For larger graphs, the asynchronous algorithm with task parallelism runs 1.6 times faster than the algorithm running on OpenMP 2.5 with an efficiency of 40%.

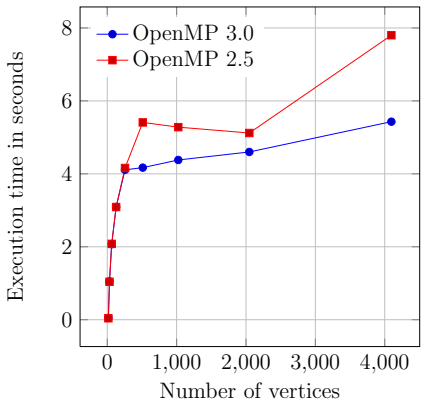

Figure 3.5: BFS execution Time for SSAC#2

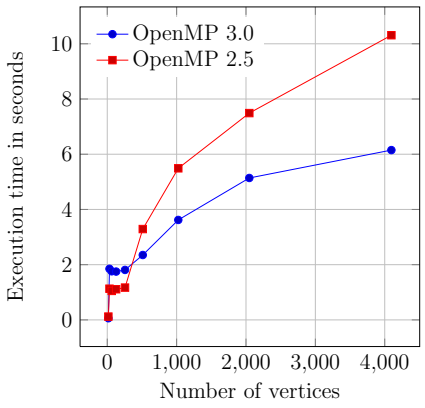

Figure 3.6: BFS execution Time for R-MAT

Chapter 4

All-Pairs Shortest Path

The All-Pairs Shortest Path (APSP) problem finds the shortest paths between every pair of nodes in a graph. Unlike the BFS algorithm, there is some cost or weight associated with each edge. The data structure is usually represented as a matrix and may be sparse depending on the structure of the graph. The all-pairs shortest path algorithm is chosen to illustrate or to explore the use of the "collapse" directive.

4.1 Introduction

The all-pairs shortest path algorithm finds the shortest paths from every vertex to all other vertices in a graph. There are two common sequential algorithms used to solve all-pairs shortest path problem: the Floyd-Warshall and Dijkstra algorithms.

Dijkstra's algorithm is used to find the shortest path for a single vertex to other vertices by checking the cost or weight of each path individually. So, it can be used to solve APSP problem by applying Dijkstra's algorithm on every vertex.

Dijkstra's algorithm is inherently sequential. Therefore, it does not lend itself easily to parallelization. Also, it does not exhibit loop level parallelism required to study the collapse directive. Therefore, I considered the Floyd-Warshall algorithm.

Floyd-Warshall's algorithm uses a 2D adjacency matrix, an associated weight or cost to store the total distance cost from every vertex to every other vertex in the graph. It basically consists of three loops. The inner loop computes the cost. The pseudocode for sequential Floyd-Warshall's algorithm is given in Algorithm 5:

Algorithm 5: Sequential Floyd-Warshall's Algorithm

Input: $G = (V, E)$

begin

$Cost_(i, j) \longleftarrow \infty$

$Cost_(i, j) \longleftarrow Weight_(i, j)$

for $i \leftarrow 0$ **to** n **do**

for $j \leftarrow 0$ **to** n **do**

for $k \leftarrow 0$ **to** n **do**

$Cost_(j, k) = \min(Cost_(j, k), Cost_(j, i) + Cost_(i, k))$

Floyd-Warshall is a cubic-time algorithm. So, the processing time is high for large graphs. One way of implementing the algorithm in parallel is by using dynamic programming. The algorithm suggests a recursive rule for computing the shortest paths. At each iteration, a partial solution which is optimal is found and stored in a table. This partial solution is modified at each step until an optimal shortest path solution is found.

In dynamic programming, there are two concerns: the memory requirement

of the table and the cost of synchronization at each iteration. To implement the algorithm, we could use the dynamic programming technique or in OpenMP, we can take advantage of loop level parallelism. One way is to allow the second inner loop to compute in parallel.

We can parallelize Floyd-Warshall's algorithm in OpenMP 2.5 as shown in algorithm 6.

Algorithm 6: Parallel Floyd's Algorithm using OpenMP 2.5

Input: $G = (V, E)$

begin

$Cost_(i, j) \longleftarrow \infty$

$Cost_(i, j) \longleftarrow Weight_(i, j)$

for $i \leftarrow 0$ **to** n **do**

for $j \leftarrow 0$ **to** n **do in parallel**

for $k \leftarrow 0$ **to** n **do**

$Cost_(j, k) = \min(Cost_(j, k), Cost_(j, i) + Cost_(i, k))$

In the above OpenMP example, the master thread executes the two outer loops sequentially. For every i and j, n ($k = 0 \quad to \quad n$) iterations are forked which are executed by the threads. If there are n threads, each iteration k will be executed by n threads. At the end of the execution, the threads join or synchronize and then the master thread resumes. If n is large, there are many forks and joins degrading performance.

In section 4.3 I will discuss how to implement Floyd's algorithm more efficiently using OpenMP 3.0.

4.2 Related work

Venkataraman et al. (Venkataraman et al. [2003]) proposed a blocked version of Floyd -Warshall's algorithm to solve the all-pairs shortest path problem. Their algorithm exploits cache locality to optimize performance. The algorithm divides the initial adjacency matrix into blocks of B $\times$ B and each block processes individually in B iterations. They tested their blocked algorithm on two different machines, a Sun Ultra Enterprise 4000/5000 and a SGI O2. Their blocked algorithm delivered a speedup between 1.6 to 1.9 for graphs between 480 to 3200 vertices on Sun Ultra Enterprise 4000/5000 and from 1.6 to 2 on the SGI O2 for graphs between 240 to 1200 vertices.

Likewise, Ma et al. (Ma et al. [2010]) developed a parallel version of Floyd-Warshall's algorithm for multi-core architectures using Threading Building Blocks (TBB). A TBB is a parallel programming model for C++ code. It is a runtime based programming model that specifies tasks. The tasks are mapped to threads. However, unlike Venkatraman et al., Ma et al. use task and data level parallelism available in the algorithm to find all-pairs shortest-paths. The results reveal that the parallel algorithm (uses up to 128 threads) surpasses both serial and single threaded algorithms by 57.26% and 50.06% respectively.

Recently, Jasika et al. (Jasika et al. [2012]) used Dijkstra's algorithm for APSP. They used OpenMP to parallelize Dijkstra algorithm. They use the algorithm to find the single source shortest path for every vertex. They compared the OpenMP implementation to an OpenCL implementation by Munshi [2009] and showed that there was no gain in performance in the two implementations. This is due to the inherent

sequential nature of Dijkstra's algorithm problems which makes this algorithm very difficult to be efficiently parallelized.

4.3 My implementation

I used the new directive called "collapse" available in OpenMP 3.0 to handle the nested loops. This directive deals efficiently with multi-dimensional loops. In other words, it combines multiple loops into a single loop. Thus, by using the "collapse" directive, I avoided the overhead of spawning threads within the nested loop in the Floy-Warshall algorithm. Also, I created a task for each vertex and process them in parallel since each vertex is independent of each other. If I collapsed the three nested loops, number of chunks to be handled by thread will be high making it coarse-grained algorithm. Instead, I collapsed the first two nested loop so the algorithm will be find-grained algorithm. Algorithm 7 shows my parallel APSP algorithm using OpenMP 3.0.

Algorithm 7: Parallel APSP Algorithm

Input: $G = (V, E)$

begin

$Cost_(i, j) \longleftarrow \infty$

$Cost_(i, j) \longleftarrow Weight_(i, j)$

for $i \leftarrow 0$ **to** n **do in parallel collapse (2)**

for $j \leftarrow 0$ **to** n **do**

Create task ()

for $k \leftarrow 0$ **to** n **do**

$Cost_(j, k) = \min(Cost_(j, k), Cost_(j, i) + Cost_(i, k))$

4.4 Results

I used undirected graphs for our experiments. I started from 16 vertices and increase the number of vertices to 4096. I compared OpenMP 2.5 to newer OpenMP 3.0 versions for both R-MAT and SSCA#2 graphs. The execution time is in seconds.

As shown in Table 4.1 and Table 4.2 and the subsequent Figures 4.1 and 4.2 respectively, the algorithm runs a bit slower on OpenMP 3.0 for small numbers of vertices. However, for large numbers of vertices, the algorithm on OpenMP 3.0 surpasses the one on OpenMP 2.5 by 1.6 times and also achieved 40% efficiency. The new collapse feature allows effective use of the OpenMP 3.0 threads. By collapsing the loops we make more efficient use of the resources and also eliminate unnecessary synchronization between the first two inner *for* loops that are collapsed.

Number of vertices	OpenMP 3.0	OpenMP 2.5
16	0.002	0.001
32	0.003	0.001
64	0.01	0.004
128	0.03	0.01
256	0.11	0.07
512	0.53	0.50
1024	3.06	4.06
2048	19.59	31.81
4096	158.85	257.47

Table 4.1: The APSP execution time on SSAC#2

Number of vertices	OpenMP 3.0	OpenMP 2.5
16	0.002	0.001
32	0.003	0.001
64	0.01	0.004
128	0.03	0.01
256	0.11	0.07
512	0.73	0.52
1024	4.08	3.91
2048	21.56	31.02
4096	154.21	251.12

Table 4.2: The APSP execution time on R-MAT

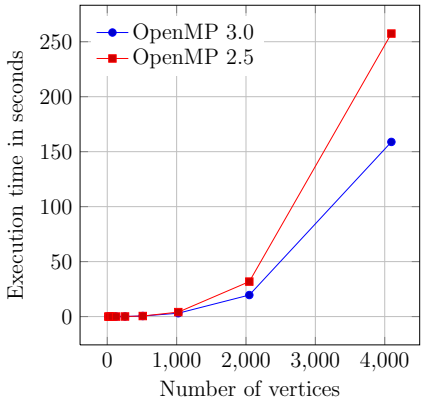

Figure 4.1: APSP execution Time for SSAC#2

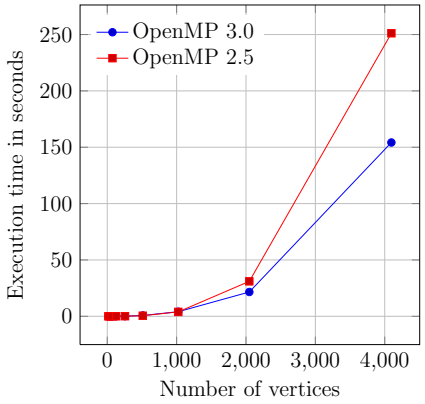

Figure 4.2: APSP execution Time for R-MAT

Chapter 5

Minimum Spanning Tree

The Minimum Spanning Tree (MST) algorithm has applications in network optimization problems (Bertsekas [1992]; Ahuja et al. [1993]). MST constructs a tree from a given graph. In particular, it constructs a spanning tree, which includes all the vertices in the graph and total weight of the tree is minimized. A brute force method of finding an MST in a graph requires exponential time . The most classical approach is using a greedy technique. Greedy algorithms build the tree incrementally.

5.1 Introduction

Given a connected, weighted undirected graph G, MST finds a tree, T, such that the tree contains all vertices of the graph and the total weight of the tree is minimized. There are many algorithms to find MST.

Prim's algorithm and Kruskal's algorithm (Leiserson et al. [2001]) are well-known algorithms to find a MST. Prim's algorithm initially starts with a single vertex. It

constructs a partial solution at each iteration by adding an edge (in turn a vertex) to the existing partial solution as long as it does not form a cycle. The chosen edge is of minimum weight connecting to the vertices in the existing partial solution. The algorithm terminates when all the vertices have been added to the tree.

Algorithm 8 shows a sequential version of Prim's algorithm (Neapolitan and Naimipour [2004]):

Algorithm 8: Prim's Algorithm

Input: $G = (V, E)$

begin

$SetEdges \leftarrow \emptyset$

$SetVertex \leftarrow v_1$

$min \leftarrow \infty$

for $EveryVertex$ **do**

$dist[i] \leftarrow weight[1][i]$

for $EveryVertex$ **do**

if $dist[i] < min$ **then**

$min \leftarrow dist[i]$

$VertexNear \leftarrow i$

Add $CurruntEdge$ to $SetEdges$

for $VertexNear$ *to* $CurrentVertex$ **do**

if $weight[i][VertexNear] < dist[i]$ **then**

$dist[i] \leftarrow weight[i][VertexNear]$

$SetVertex \leftarrow VertexNear$

Kruskal's algorithm on the other hand, builds a forest of subtrees. The algorithm creates a set of disjoint subtrees of vertices. The trees are then merged if there is a connecting edge of smallest weight between two subsets. The algorithm terminates when there is one tree contains all the vertices.

Algorithm 9 shows a sequential implementation of Kruskal's algorithm.

Algorithm 9: Kruskal's Algorithm

```
Input: G = (V, E)
begin
    Sort all edges by weight in increasing order
    SetEdges ← ∅
    while Number of edges in SetEdges < disjoint subsets do
        e← least weighted edge not yet in SetEdges
        u← the first connected by vertex e
        v← the second connected by vertex e
        if u and v not in the same set then
            merge u and v
            Add e to SetEdges
    return SetEdges
```

Prim's algorithm is hard to parallelize. It is more sequential in nature. In Kruskal's algorithm the first part of the algorithm, the sorting part, can be parallelized. However, it requires a bit more effort to parallelize the disjoint-union part of the algorithm. A lot of synchronization is needed here since many threads want to edges to the final set of edges. Also, in each step of Kruskal's algorithm, only two partial trees are merged. The level of concurrency in this algorithm, is therefore,

limited.

There is another algorithm called, Borůvka's MST algorithm (Bader and Cong [2006]). This algorithm is iterative and suited for parallelization.

Like Kruskal's algorithm, it partitions the vertices into subtrees, creating a forest of trees. By merging two or more vertices, the algorithm creates what is called a supervertex. The vertices and all edges incident on the vertices are contracted into one supervertex. Like Kruskal's algorithm, Borůvka's algorithm selects one edge of smallest weight coming out of a component for the merging process. The algorithm merges components until there is one supervertex in the end.

In Borůvka's algorithm, in a single step, all the supervertices participate in the merger allowing more concurrency, unlike Kruskal's algorithm. Choosing multiple edges in a single step to merge the components requires longer processing time and is more complex.

Algorithm 10 illustrates a sequential implementation of Borůvka's algorithm.

Algorithm 10: Borůvka's Algorithm

Input: $G = (V, E)$

begin

$Supervertices \leftarrow numberofvertices$

$Supervertices_i \leftarrow v_i$

$T \leftarrow \emptyset$

while $Supervertices \neq 1$ **do**

for $Eachvertex$ **do**

Find the smallest edge between v_i and v_j such that

$Supervertices_{v_i} \neq Supervertices_{v_j}$.

Put the smallest edge in T

Set $Supervertices_{v_j} = Supervertices_{v_i}$

$Supervertices = Supervertices - 1$

5.2 Related work

Bader and Cong. (Bader and Cong [2006]) introduced a non-deterministic shared memory parallel MST algorithm by combining Borůvka's and Prim's algorithms. At the beginning, the algorithm starts with Prim's algorithm from different vertices to construct many sub-trees. Then, the algorithm uses Borůvka's algorithm to merge these sub-trees to form an single spanning tree (also so-called a supervertex).

Setia et al. (Setia et al. [2009]) used the same idea of Bader and Cong to find MST using Prim's algorithm. The algorithm chooses random vertices as roots and then concurrently each thread builds a subtree. Each node has a unique color which indicates its thread id. A vertex is added into a tree by a thread if it does not belong

to any other trees. However, when a thread wants to add a vertex to its tree and the vertex belongs to another tree, both trees are merged into one tree and the root is updated to the smaller id thread between the two trees. At the end, thread 0 will have the MST. This algorithm achieved speedup of 2.64 for 4 threads on dense graph with 5000 vertices.

Chung and Condon (Chung and Condon [1996]) implemented Borůvka's algorithm on Thinking machine's CM-5. Their implementation was straightforward to measure the speedup of the algorithm on a homogeneous architecture. The speedup obtained by their algorithm was about 4 for sparse graph of 64,000 vertices on 16 processors.

5.3 My implementation

I used Borůvka's algorithm for my parallel implementation of MST for a shared memory architecture. This algorithm is a bit more complicated than others. The algorithm partitions vertices into supervertices. At the beginning, each vertex is regarded as a supervertex. Each supervertex is a task. The task contains information about the vertex, its corresponding neighbours, and the code it needs to execute. Each task selects a cheapest edge and determines which supervertex it should merge with. Each task also determines the root of its subtree. Each task is executed by a thread.

In the merging phase, although all the components can participate, creating one supervertex of two components requires a lot of synchronization. This, I felt was not worthwhile doing since it will degrade performance and may be-

have very poorly. In the literature (Chung and Condon [1996]) the merging phase is done sequentially. Therefore, I sequentially implemented the merging step. After the merging step, the supervertices that were merged are now task. Since each task consists of code and data, we can avoid accessing the global memory frequently to store the merged information. The Parallel implementation of Borůvka's MST algorithm is given in Algorithm11.

Algorithm 11: Parallel Borůvka Algorithm

Input: $G = (V, E)$

begin

$Supervertices \leftarrow numberofvertices$

$Supervertices_i = v_i$

$T \leftarrow \emptyset$

while $Supervertices \neq 1$ **do**

for $Eachvertexv$ **do in parallel**

Create task ()

Find the cheapest edge between v_i and v_j such that

$Supervertices_{v_i} \neq Supervertices_{v_j}$.

Put the cheapest edge in T

Synchronization barrier ()

Set $Supervertices_{v_j} = Supervertices_{v_i}$

$Supervertices = Supervertices - 1$

5.4 Results

I used undirected graphs for our experiments. I start from 16 vertices and increase the number of vertices to 4096. I compared OpenMP 2.5 to OpenMP 3.0 versions for both R-MAT and SSCA#2 graphs. The execution time is in seconds.

In OpenMP 2.5, I used a coarse-grained single-program multiple data approach. The vertices are partitioned among the threads and the threads execute the sequential Borůvka algorithm. I merge the supervertices and store the information in shared memory. The algorithm, implemented in OpenMP 2.5, uses the *parallel* directive.

Number of vertices	**OpenMP 3.0**	**OpenMP 2.5**
16	0.06	0.06
32	0.12	0.13
64	0.19	0.17
128	0.28	0.27
256	0.50	0.42
512	1.25	1.21
1024	2.82	2.14
2048	6.40	7.21
4096	17.30	21.66

Table 5.1: The MST execution time on SSAC#2

Number of vertices	**OpenMP 3.0**	**OpenMP 2.5**
16	0.06	0.10
32	0.08	0.10
64	0.27	0.30
128	0.03	0.01
256	0.27	0.20
512	1.22	1.63
1024	3.31	4.20
2048	6.14	7.30
4096	13.90	28.10

Table 5.2: The MST execution time on R-MAT

It can be seen from Table 5.1(Figure 5.1) and Table 5.2 (Figure 5.2) that OpenMP 3.0 implementation outperforms OpenMP 2.5 in general especially, in large number of vertices. Due to synchronization of supervertices in shared memory, the OpenMP 2.5 execution times are higher. Since there were more than one supervertex participating the merge step, synchronization overhead had a significant impact on the performance. The average speedup attained is 1.6 times faster than OpenMP 2.5 with efficiency 40%. One reason for that is using the task directive for choosing the edges.

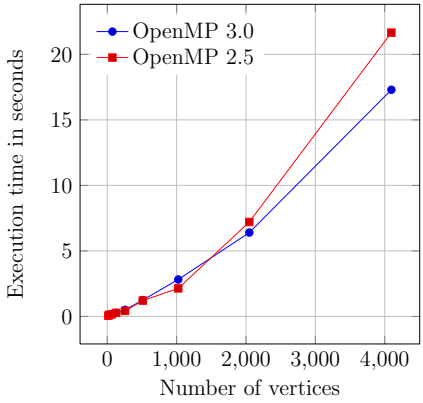

Figure 5.1: MST execution Time for SSAC#2

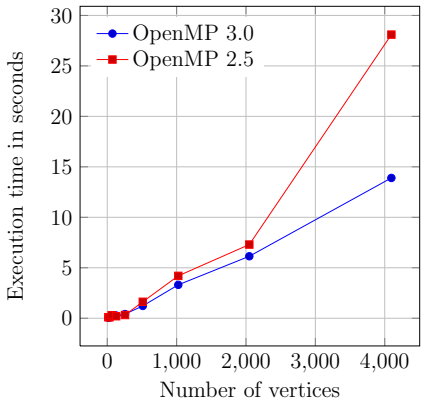

Figure 5.2: MST execution Time for R-MAT

Chapter 6

Conclusions and Discussion

One of the main contributions in my thesis is to provide some insight into the usefulness of OpenMP 3.0 for irregular problems. OpenMP was meant for solving scientific computing problems involving loops written in high level languages such as C or C++. Algorithms that incorporate a regular data structure such as an array, and provide lots of data parallelism can be efficiently implemented in OpenMP. An example of such an algorithm is dense matrix multiplication. Most of the algorithms that possess such regularity manifest themselves into loops. The bounds of the loops are known at compile time and there are no real surprises at runtime in terms of the computations being performed by the algorithms. Although OpenMP provides many other types of compiler directives other than to exploit loop level parallelism, the strength in OpenMP lies in solving problems that are loop based.

Irregular algorithms, on the other hand, commonly use pointer-based data structures which are unstructured, and data size changes at runtime require intermittent load balancing among processors. The amount of computations may not be known

at compile time complicating matters further. On homogeneous multi-core architectures, caches exacerbate the problem since access to memory in irregular algorithms is not necessarily contiguous and frequent transfer of data between caches and memory would degrade performance (Zhang and Torrellas [1995]), a long studied issue. From a software perspective, OpenMP (Mattson [2003]) was not meant for irregular applications. To develop parallel algorithms for irregular computations, any parallel language requires the ability to dynamically create and destroy threads under explicit programmer control. This has now been incorporated in version 3.0 of OpenMP which provides programmers with the flexibility to create and destroy threads in a parallel region. This is done through the "task" directive, a new directive available in OpenMP 3.0. Another feature provided in OpenMP 3.0 is the "collapse" feature. This feature tries to reduce the number of forks and joins in nested loops by allowing user to specify where loop joining is safe to apply. This is useful in some irregular algorithms.

In this thesis, I explored the use of task parallelization on two graph problems and the *collapse* feature on one loop based graph problem. I first considered the Breadth-First Search (BFS) algorithm to find the shortest path from a root vertex to all the other vertices in the corresponding graph. A synchronous BFS algorithm creates too much synchronization overhead at each level of the algorithm. The number of levels is unknown at runtime. I therefore developed an asynchronous version of the algorithm which allows me to create tasks following a parent-child relationship. The idea comes from (Chandy and Misra [1982]). The main issue in the implementation was destroying tasks and their parents appropriately to avoid programs to terminate

abruptly. I have not seen any published work study this BFS implementation using ideas from (Chandy and Misra [1982]). I was able to achieve up to 40% efficiency over OpenMP 2.5.

Next, I tested the collapse feature of OpenMP 3.0. The purpose of the collapse feature is to condense multiple nested loops into a single loop. To accomplish this I used Floyd-Warshall's all-pairs shortest-path algorithm. This algorithm normally works very slowly as the number of vertices increases. I used collapse with this algorithm to combine the nested loops in the algorithm. Consequently, my implementation of Floyd-Warshall's algorithm ran faster on OpenMP 3.0 by factor of 1.6 times. This was due to the reduction of forks and joins.

Finally, I chose the Minimun Spanning Tree (MST) problem to evaluate task parallelism on OpenMP 3.0. Borůvka's algorithm was chosen over Prim and Kruskal's algorithm. Here I used a different approach from BFS. I did not create a parent-child relationship between tasks. Instead, each vertex was an independent task initially. Each task encompasses some data and the code, in this case selecting the cheapest edge. The tasks we combined to form supervertices and these supervertices then in turn became new tasks. In this algorithm, I could not parallelize the whole algorithm since there are some portions of the algorithm that were intrinsically sequential. Although, it performed better than the OpenMP 2.5 version (which uses parallel regions alone) the overall execution time increases for large number of vertices.

It was quite challenging to work with the task directive. Using the task directive for recursive algorithms would be easier to work with. When the algorithms are non-recursive, and are irregular, it would be more efficient to develop an approach that

can efficiently make use of task level parallelism rather than simply massaging the existing algorithm to suit the task level parallelism features. This requires more effort on the programmers part and moves away from adding just directives to sequential code that OpenMP so easily allows one to do. The level of programming now is more complicated, but due to the sharing of resources provided in shared memory machines, the simplicity of understanding OpenMP, it is still worthwhile the effort to consider OpenMP over other APIs for irregular problems.

Chapter 7

Future work

The work described in this thesis aimed to investigate the capability of OpenMP 3.0 to work with irregular problems. Through this thesis, I have shown that OpenMP 3.0 provides features for better handling of task oriented problems needed to handle irregularity compared to the previous versions OpenMP 2.5.

For my immediate future work, I will consider the extension of OpenMP. OpenMP 4.0 was launched in April 2013. OpenMP 4.0 provides support for SIMD constructs for vectorization. This will allow implementing data parallel applications more efficiently. OpenMP 4.0 also enhances task level parallelism. Moreover, OpenMP 4.0 gives more control over threads which leads to better locality for threads and less false sharing between them which I believe may be helpful for irreguler applications which introduced dynamic threads creation.

The second choice is to use OpenACC (ope). OpenACC is, therefore, similar to OpenMP in the sense that it also incorporates SIMD instructions that can be implemented specifically on Nvidia GPUs. OpenACC is still in its infancy and not

much research on its capabilities exists in the literature. One of the issues with OpenACC is that it does not allow programmers to have control of threads, such as destroying threads. As with OpenMP, OpenACC consists of compiler directives that can be added to parallel regions and can be executed on the CPU or GPU cores or both for handling loop level parallelization on vectorization. In the current version, to run programs in OpenACC requires Nvidia CUDA enabled GPUs.

It would also be interesting to see how irregular problems can be implemented in OpenACC. This would allow me to compare the performance of irregular problems in OpenACC to CUDA (cud) implementations. CUDA is a well-known API for accelerators. There is some work in graph problems (Solomon et al. [2010]) on Nvidia GPUs already and the results have been impressive. I plan to implement the same graph algorithms and compare the results to the CUDA performance results in the literature.

Bibliography

Cuda. http://www.nvidia.ca/object/cuda_home_new.html.

The graph500 list. http://www.graph500.org/.

openacc. http://www.openacc-standard.org/.

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, New Jersey, 1993.

E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A proposal for task parallelism in OpenMP. In *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era, Beijing, China*, pages 1–12, 03–07 June 2007.

E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proceedings of International*

Conference on Parallel Processing (ICPP 2006), Columbus, Ohio, USA, pages 523–530, 2006.

D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, November 2006.

D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC 2005), Goa, India*, pages 465–476, 2005.

D. P. Bertsekas. *Linear Network Optimization: Algorithms and Codes*. The MIT Press, 1992.

R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 1995), Santa Barbara, CA, USA*, pages 207–216, 1995.

O. Borůvka. About a certian minimal problem. *Práca Moravské Přirodovědecké Společnosti*, 3:37–58, 1926.

D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph

Mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining (2004), Lake Buean Vista, FL, USA*, 22–24 April 2004.

K. M. Chandy and J. Misra. Distribued computations on graphs: shortest path algorithms. In *Communications of the ACM*, volume 25, pages 833–837, 1982.

S. Chung and A. Condon. Parallel implementation of Broůvka's minimum spanning tree algorithm. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS 1996), Honolulu, HI, USA*, 15–19 April 1996.

L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

E. Dedu, S. Vialle, and C. Timsit. Comparison of OpenMP and classical multithreading parallelization for regular and irregular algorithms. In *Proceedings of Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD 2000), Champagne-Ardenne, France*, pages 53–60, 19–21 May 2000.

J. B. Dennis. Packet communication architecture. In *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, pages 224–229, 1975.

D. Hisley, G. Agrawal, P. Satya-narayana, and L. Pollock. Porting and performance evaluation of irregular codes using OpenMP. In *Proceedings of First European Workshop on OpenMP (EWOMP 1999), Lund, Sweden*, pages 47–59, 1999.

N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *Proceeding*

of the 35th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2012), Opatija, Croatia, pages 1811 –1815, 21–25 May 2012.

T. S. John, J. B. Dennis, and G. R. Gao. Massively parallel breadth first search using a tree-structured memory model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012), New Orleans, LA, USA*, pages 115–123, 2012.

M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2007), San Diego, CA, USA*, pages 211–222, 2007.

M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), Munich, Germany*, pages 217–228, 2008.

J. Labarta, E. Ayguadé, J. Oliver, and D. Henty. New OpenMP directives for irregular data access loops. *Scientific Programming*, 9(2,3):175–183, August 2001.

C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM symposium on Parallelism in Algorithms and Architectures (SPAA 2010), Thira, Santorini, Greece*, pages 303–314, 2010.

C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to Algorithms*. The MIT press, 2001.

J. Ma, K. ping Li, and L. yan Zhang. A parallel floyd-warshall algorithm based on TBB. In *Proceedings of The 2nd IEEE International Conference on Information Management and Engineering (ICIME 2010), Bangkok, Thailand*, pages 429–433, 2010.

T. G. Mattson. How good is openmp. *Scientific Programming*, 11(2):81–93, 2003.

E. F. Moore. *The shortest path through a maze*. Bell Telephone System., 1959.

A. Munshi. The opencl specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

R. E. Neapolitan and K. Naimipour. *Foundations of Algorithms: Using Java Pseudocode*. Jones & Bartlett Learning, 2004.

J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Proceedings of the 4th International Conference on Computing Frontiers (CF 2007), Ischia, Italy*, pages 47–58, 7–9 May 2007.

OpenMP. The OpenMP API specification for parallel programming. `http://openmp.org/wp/`, 1998.

S. Secchi, A. Tumeo, and O. Villa. A bandwidth-optimized multi-core architecture for irregular applications. In *Proceedings of 12th IEEE/ACM International Symposium*

on Cluster, Cloud and Grid Computing, (CCGrid 2012), Ottawa, ON, Canada, pages 580–587, 13–16 May 2012.

R. Setia, A. Nedunchezhian, and S. Balachandran. A new parallel algorithm for minimum spanning tree problem. In *Proceeding of the 16th annual IEEE International Conference on High Performance Computing (HiPC 2009), Kochi, India*, pages 1–5, 16–19 December 2009.

S. Solomon, P. Thulasiraman, and R. K. Thulasiram. Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC 2010), Melbourne, Australia*, pages 297–304, 1–3 September 2010.

M. Süßand C. Leopold. Implementing irregular parallel algorithms with OpenMP. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par 2006), Dresden, Germany*, pages 635–644, 2006.

G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal on Experimental Algorithmics*, 8, December 2003.

J. Wang, C. Hu, J. Zhang, and J. Li. OpenMP extensions for irregular parallel applications on clusters. In *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era (IWOMP 2007), Beijing, China*, pages 101–111, 3–7 June 2007.

B. Wicaksono, R. C. Nanjegowda, and B. Chapman. A dynamic optimization frame-

work for OpenMP. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era (IWOMP 2011), Chicago, IL, USA*, pages 54–68, 2011.

Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA 1995), S. Margherita Ligure, Italy*, pages 188–199, 1995.