在對稱式多處理器系統上運用 OpenMP 以降低負載提昇執行效率於不同型式之資料傳送模式

研究生：陳獻昌　　　　　　　　指導教授: 楊朝棟　博士

私立東海大學理學院資訊工程與科學系

摘要

本篇論文主要是探討在對稱式多處理器系統上於不同型式的資料傳送模式下，如何降低負載來提昇執行效率。運用 OpenMP 的多執行緒功能在 SUN Fire 6800 對稱式多處理器電腦環境下，針對兩種不同型式的資料傳送模式，進行負載與效能提昇的分析。第一種為資料獨立模式(Independent Model)，是資料各自獨立處理相互間不會有交換或更新；例如雙質數(Twin Primes)及矩陣相乘之計算。第二種為相鄰資料交換模式(Nearest Neighbor Model)是相鄰資料間相互會有交換或更新；例如 Laplace 方程式計算。其他重要的分析考量因素尚有：計算式的邏輯模式、記憶體的可用狀態、與資料的切割等，皆為提昇執行效率的重要影響因素。

Minimize Overhead to Improve Performance of Different Data Communication Styles by OpenMP on SMP

Student: Shien-Chang Chen Advisor: Dr. Chao-Tung Yang
Department of Computer Science and Information Engineering
Tunghai University
Taichung, 407, Taiwan, Republic of China

Abstract

In this thesis, we study the execution overhead of loop iterations by using OpenMP with multithreads programming on SMP (Symmetric Multiprocessors) systems. To analyze the speedup and improvement the performance, two data communication styles are used: Independent model and Nearest Neighbor model [1]. Two distinct models based on the span of the data stencil used to update the next point. For Independent model the update algorithm requires the data only from the previous time step or initial conditions and the general algorithm, i.e. Twin Primes and Matrix Multiplication computing. But Nearest Neighbor model the update algorithm is similar to the method of approximating derivatives, which used central differences. First derivatives calculated using central differences only use values from neighboring points. The method can be used to solve several classes of equations, such as Laplace's Equation computing. As we analyze the overhead, some important factors need to be taken into account, which include computation algorithm, available memory of the system and the data decomposition or partition method.

Acknowledgements

I would like to thank all the people who have made writing this thesis a more pleasant task. In particular, I'd like to thank my principal advisor, Dr. Chao-Tung Yang, who introduced me to this topic and gave me broad support and guidance throughout my time as Tunghai. I'd like to thank Professor Wuu Yang, Professor Nai-Wei Lin and Professor Yi-Min Wang for their valuable comments and advice given while serving on my reading committee.

There are many other people whom I would like to thank. Especially to My wife, without her encourages and supports that I can't pursue further education in Tunghai. Many colleagues encouraged and supported me on studying. For them, I can make writing this thesis with no fear of disturbance in the rear.

Last, but certainly not the last, I'd like to make my family and all my friends whose unconditional support made this thesis possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Threading in a Multiprocessor Systems

For multiprocessor system a CPU could not write directly to memory simply by wiggling the voltages on a few wires that connected the CPU chip to the memory chip. All was well with the world. In multithreaded systems, only one path to memory existed reads and writes to memory. It is always occurred whenever the CPU executed the associated machine instruction. The introduction of memory caches didn't fundamentally change that model (once they got the cache-coherency bugs worked out). Indeed, the cache is transparent to the program if it's implemented correctly. That simple memory model -- the CPU issues an instruction that modifies memory with an immediate effect -- remains in most programmers' minds.

Somebody had the bright idea that two or more processors could run in the same box at the same time, sharing a common memory store (Suddenly, the world became much more complicated). In that situation, a given CPU can no longer access memory directly because another CPU might be using the memory at the same time. To solve the problem, along came a traffic-cop chip, called a *memory unit.* Each CPU was paired with its own memory unit, and the various memory units coordinated with each other to safely access the shared memory. Under that model, a CPU doesn't write directly to memory but requests a read or write operation from its paired memory unit, which updates the main memory store when it can get access, as seen in Figure 1.1.

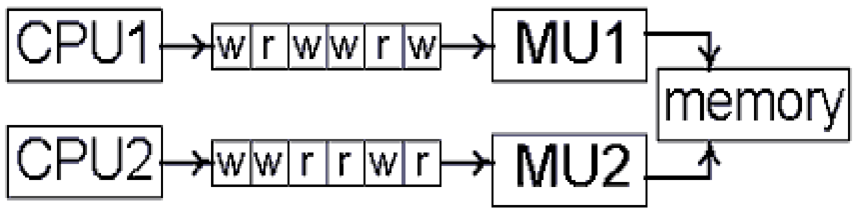

Figure 1.1: The memory unit

To solve memory coherency the bright engineer noticed some method to optimize memory operations with the hardware or operation system such as relaxed memory, synchronize. The visibility of a memory modification is guaranteed only when the modifying thread releases a lock that is subsequently acquired by the examining thread.

For the threading, it’s depends on the number of CPU within SMP (Symmetric Multiprocessor). Some compiler can run with over threading, such as POSIX thread, while the Omni OpenMP the number of threading is limited to the number of CPU in SMP.

1.2 Parallel Programming Concepts and Terminology

1.2.1 Parallel Programming Paradigms

The term “parallel programming” refers to writing a program that takes advantage of parallel processing, in which multiple processors take part in executing a single program. From programmers’ point of view, there are two major paradigms in doing so. The shared-memory programming model loosely targets a shared memory architecture, in which multiple processors share single memory space. The communication between processors takes place through reading and writing in this memory space. The notion of “shared” and “private” data becomes important. Shared data are visible to all processors participating in parallel execution. Communication between processors takes place in the form of reading and writing to share data. Private data, on the other hand, is local to each processor and cannot be accessed by other processors.

The other form of parallel processing targets message-passing architecture. Processors do not share memory; instead, they explicitly send and receive messages. All data are private, and the only way to acquire the data that are not in the local memory is to request and receive them from the processor that has them. This thesis assumes a shared-memory programming paradigm. Note, that this paradigm is viewed from a software perspective. At the architecture level a shared-memory software paradigm can be implemented with distributed-memory architecture, as long as there are hardware or system software mechanisms that enable the program to access shared data as if it were placed in a common memory. In this case, the “only” effect noticed by the program is that access to some memory is faster than to other memory. We will describe software techniques that deal with such *non-uniform memory access* behavior.

1.2.2 Parallelization Constructs

In order to tell the underlying machine that a program should be executed in parallel, we need some form of programming language constructs. These constructs control data sharing, synchronization, and so on. The two paradigms offer different sets of parallel constructs to achieve this.

In our shared memory model, a programmer inserts “directives” into the code. These directives do not affect the program semantics. They dictate how the parallel processors shall share work and data.

The directives usually target program sections with repetitive execution pattern, mainly loops. A programmer, who wants certain loops to be executed in parallel, inserts appropriate directives before these sections. The machine code generating compilers generate parallel executable code based on these directives. An alternative way of expressing programs in the shared-memory model is to use *threads*. In this scheme the programmer packages program sections that can execute concurrently into subroutines and *spawns* these subroutines as parallel activities, called threads. In the view of this document, threads parallelism is at a lower level than directive parallelism. In fact, the compiler will translate a directive-parallel program into a thread-parallel program as an intermediate compilation step. Advanced parallel programmers sometimes prefer threads parallelism because it can offer more direct control over the parallel program execution. Usually, this comes at the cost of a higher programming effort, however.

In the message-passing model, the constructs typically come in the form of library of functions. The library includes functions for sending and receiving messages, synchronizing execution, and so on. The Message Passing Interface (MPI) is an important standard that is implemented in the form of such libraries. The parallel programmer's task in the message-passing model is to incorporate these functions into the algorithm. Programmers need to devise ways to split data, communicate, and synchronize, and write or modify the program based on the idea.

1.2.3 OpenMP Directive Language

In the past there have been many different sets of directives. They have recently been standardized in the form of the OpenMP directive language, which is supported by most manufacturers of shared-memory multiprocessors. All application code in this thesis are written in OpenMP. OpenMP is the result of an industry-wide effort to resolve compatibility issue in the shared memory-programming model. It embraces many existing directive languages and adds a few new concepts for more expressiveness. What follows is an example of OpenMP directives applied to a code section that computes PI.

```
/* calculate the interval size */
   w=1.0/n;
   sum=0.0;
#pragma omp parallel private (x) shared(w,sum)
{
#pragma omp for
   for (i=1;i<=n;i++)
   {
      x=w*((double)i-0.5);
#pragma omp critical
{
      sum=sum+f(x);
}
   }
}
   pi=w*sum;
```

Lines starting with *#pragma omp* indicates directives. Directive *parallel* indicates that the loop has no loop-carried dependencies and may be executed in parallel. Directives *private* and *shared* tell the compiler that the following variables in the parenthesis are private or shared, respectively. Directive *reduction(+: sum)* indicates that the variable *Sum* is a summation reduction variable (refer to the reduction technique section)[], and requires a special care for parallel execution.

Examining the details of OpenMP is beyond the scope of this guideline and readers should refer to the OpenMP group for more information.

1.3 Motivation

How to reduce overhead and to improve performance, parallel computing is necessary. For the powerful analysis and making fast and right decision, parallel computing will give a very significant support. But timing is the most important thing for parallel computing. To develop a good parallel implementation requires understanding of where run-time is spent and comparing this to some realistic best possible time. The overhead analysis is a way of comparing achieved performance with achievable performance. In the parallel programming models message passing (MPI, PVM) and threading (POSIX thread, OpenMP), we select the OpenMP Application Program Interface (API) as my application parallel language. Why using OpenMP? The reasons are stated as follows:

- Good for loop parallelization.
- Parallelization mainly compiler directives.
- Portable and scalable model for C/C++ and FORTRAN.
- New Standardized for jointly defined and endorsed by a group of major computer hardware and software vendors
- Easy to translate from sequential program into parallel program.

1.4 Contributions

By using OpenMP with multithreads programming on SMP (Symmetric Multiprocessors) systems to analyze the speedup and improvement the performance, two data communication styles are used: Independent model and Nearest Neighbor model [1]. Two distinct models based on the span of the data stencil used to update the next point. For Independent model the update algorithm requires the data only from the previous time step or initial conditions and the general algorithm, i.e. Twin Primes and Matrix Multiplication computing. But Nearest Neighbor model the update algorithm is similar to the method of approximating derivatives, which used central differences. First derivatives calculated using central differences only use values from neighboring points. The method can be used to solve several classes of equations, such as Laplace's Equation computing. As we analyze the overhead, some important factors need to be taken into account, which include computation algorithm, available memory of the system and the data decomposition or partition method. As the experiment results show that the super linear speed up is possible. Many vendors develop their machine with huge numbers of CPUs and build in large memory in the SMP system. Therefore the trend is clear and the influence will make the super high performance computing comes true.

1.5 Structure of This Thesis

This thesis starts with the brief introduction of threading in Multiprocessor; parallel programming concepts and OpenMP directive language. Chapter 2 gives a short definition of the parallel terminology, overhead and efficiency notation. It will support a deeper analysis later in this thesis. Chapter 3 is described the parallel programming environments and the framework of Omni OpenMP compiler system. In chapter 4 we implement three programs for evaluation the overhead in two data communication styles. Also we discussion the out come between the static and dynamic scheduling methods. Finally we make a brief conclusion and future work in chapter 5.

Chapter 2

Background

2.1 Overhead Analysis

Overhead analysis [2] [3] is a technique to provide developers with more information about the execution of their code, specifically to help determine the maximum performance possible. It is an extended view of Amdahl's Law, as we now explain. Assume T_s is the time spent by a serial implementation of a given algorithm and T_p is the time spent by a parallel implementation of the same algorithm on p threads. Then for perfect parallelization we would have $T_p = T_s /p$. Amdahl's Law introduces as a measure of the fraction of parallelized code in the parallel implementation, and states that as below:

$$T_p = (1 - \alpha) T_s + \alpha \frac{T_s}{p}$$

Thus, the best time for a parallel implementation is restricted by the fraction $(1\text{-}\alpha)$ of the unparalleled code. Let use to rearrange above equation to give new equation:

$$T_p = \frac{T_s}{p} + \frac{p-1}{p}(1-\alpha)T_s$$

The first term is the time for an ideal parallel implementation. The second term can be considered as an overhead or degradation of the performance. In this case it is an overhead due to unparallel code. However, this model is too simplistic in that it takes no account of any of the various factors affecting performance, such as how well the parallelized code has been implemented.

Let us therefore consider equation to be a specific form as follow.

$$T_p = \frac{T_s}{p} + \sum_i O_i$$

The O_i is an overhead for each possible overhead. The efficiency we defined as below:

$$E_p = \frac{T_s}{p \times T_p}$$

In order to calculate the overhead our function noted as below:

$$O_p = T_p - \frac{T_s}{p}$$

In this thesis we will use above definition to evaluate the efficiency between different communication styles.

2.1.1 Communication Styles

As mentioned in “Beowulf Cluster Design for Scientific PDE Models” [1], B. McGarvey and other authors classified four styles of point update methodology: Independent, Nearest Neighbor, Quasi-Global and Global. That will take most influence of the communication behavior among processors.

Independent: in Figure 2.1 the update algorithm requires the data only from the previous time step or initial conditions and the general algorithm, i.e. co-located models.

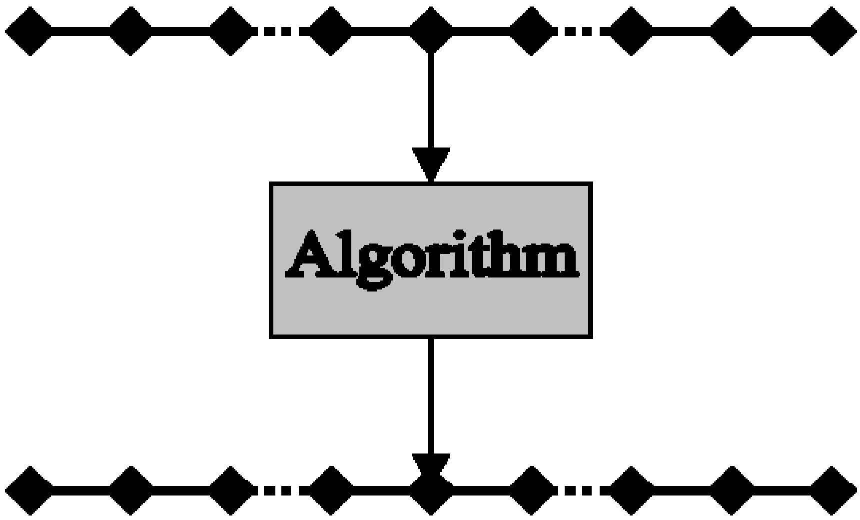

Figure 2.1: Independent point update methodology

Nearest Neighbor: in Figure 2.2 a method of approximating derivatives is to use central differences. First derivatives calculated using central differences only use values from neighboring points. This method can be used to solve several classes of equations, such as Maxwell's electromagnetic equations and Laplace's Equation.

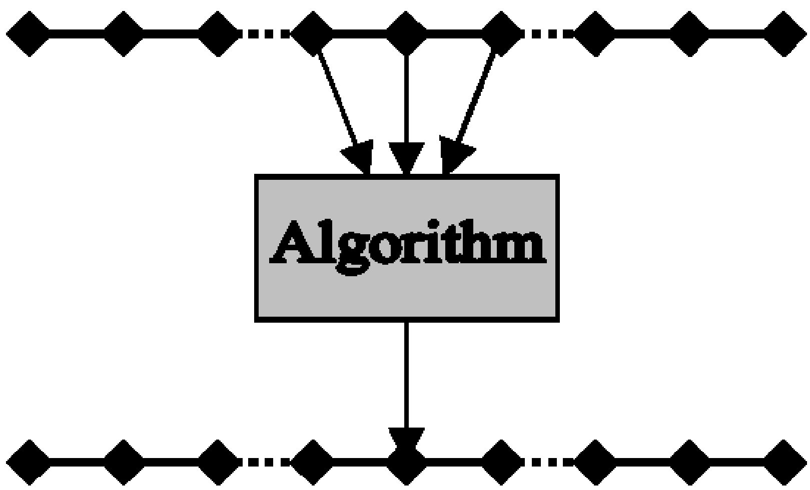

Figure 2.2: Nearest Neighbor point updated methodology

Quasi-Global: in Figure 2.3 a method for discrimination result in the need for more than just the adjacent points, the scheme determines the number of points required from the original point. One such scheme is the Battle-LeMarie Multi-Resolution Time-Domain (MRTD) method [13].

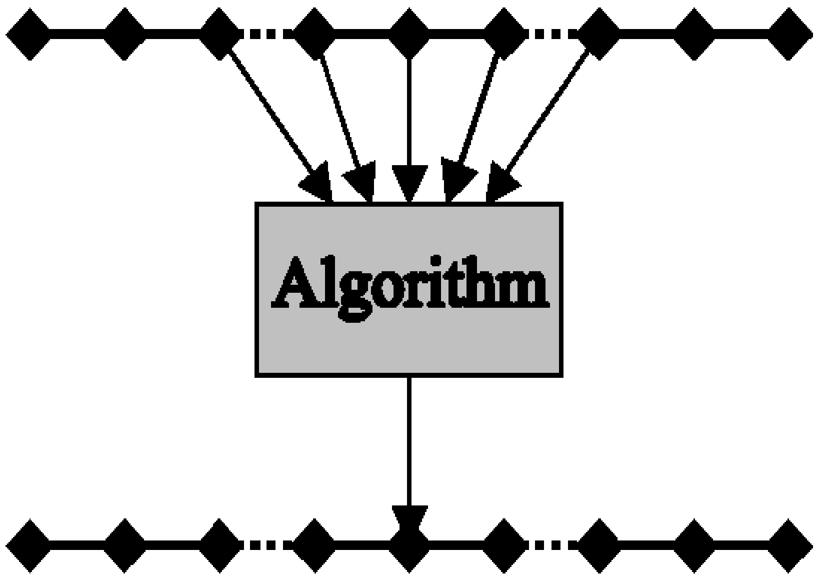

Figure 2.3: Quasi-Global point updated methodology

Global: in Figure 2.4 the global basis function requires that all the data points in the space be know to update any single point, This is generally the limiting case of data interaction; update any point depends on the data from every point on the grid [13]. Matrix inversion loosely fits into this Category.

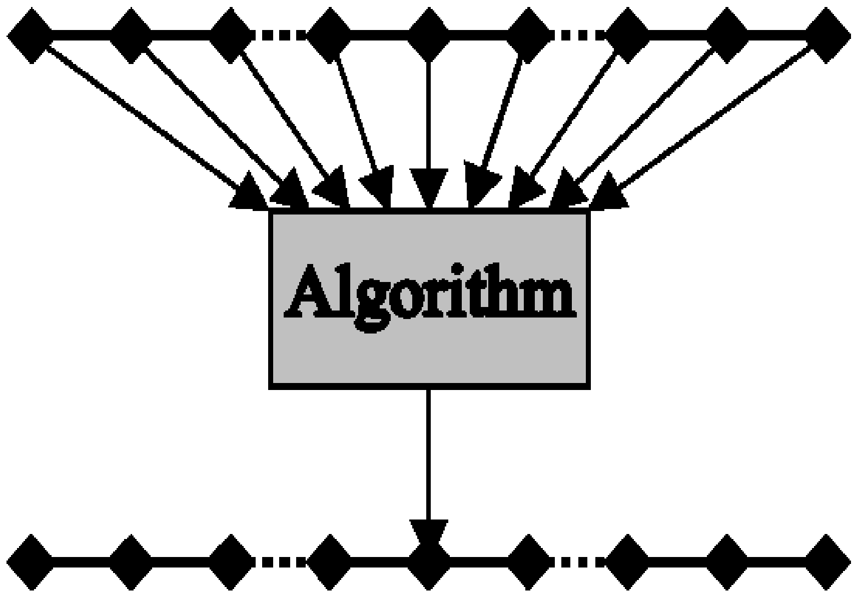

Figure 2.4: Global point updated methodology

2.1.2 Static Scheduling and Dynamic Scheduling

In the Omni OpenMP compiler, the default loop scheduling is static schedule with block scheduling. It means the scheduling with chunk size N, which may cause load imbalance when the execution time of each remote call is changed. Altering the schedule of the parallel DO loop from A simple to an interleaved distribution (by giving the SCHEDULE clause argument (STATIC, 1), meaning cyclic scheduling with chunk size 1. It should give more satisfactory load balance in the loop [2].

Dynamic scheduling adjusts the schedule during execution and is especially suitable whenever the number of iterations is uncertain or iteration may take a different amount of time. Although it is more suitable for load balancing between processors, runtime overhead is the cost. In the experiment we compare the dynamic and static scheduling only the Matrix Multiplication get best performance. Because we create the matrix by dynamic allocate memory.

2.2 A Methodology for Optimization

In order to porting and tuning the performance of these three application programs to a parallel machine. We start by identifying the most time-consuming code section of the program, optimize its performance using several recipes, and then repeat this process with the next most important code section. The most important program sections for parallel execution in our programming paradigm are loops. Hence we profile the program execution time on a loop-by-loop basis. We do this by inserting the program code with calls to timer functions. The timing profile not only allows us to identify the most important code sections, but also to monitor the program's performance improvements as we convert it from a serial to a parallel program. If we are not satisfied with the resulting performance we will modify the parallel code by hand again.

Chapter 3

Programming Environment and Applications

3.1 OpenMP Overview

3.1.1 Platforms of OpenMP

There are many vendors and groups implement their machine to support OpenMP, such as Compaq, Fujitsu, HP, IBM, Intel, KAI (KAI Software Lab is part of the Intel team of technology leaders), SGI, Sun and RWCP (Real Word Computing Partnership in Japan). These vendors or groups support the platforms of OpenMP not only for UNIX system but also in Windows NT/2000 platforms. Reference list as below:

- Sun Solaris 5.6 (SPARC and x86).
- Linux 2.2.7 (redhat-6.0, x86 SMP or above version) with Linux-threads
- IRIX 6.5 (Origin 2000) with POSIX threads
- IBM AIX with POSIX threads
- HPUX with POSIX threads
- KAP/Pro with POSIX threads.

3.1.2 Framework of Omni OpenMP Compiler System

In this thesis we select Omni OpenMP [5] to setup the compiler system, which is developed at RWCP (Real World Computing Partnership) [6]. So far it is available on many SMP (shared memory processor) platforms such as Sun Solaris, Red Hat Linux and IRIX 6.5 (Origin 2000). The SMP cluster version is under development. But it is available on SCASH [11] to implement the cluster-enabled Omni OpenMP under Score Cluster System Software for software distributed shared memory system. Regarding to grid computing there is A Grid RPC Facility for Cluster and Global Computing in OpenMP [12].

The framework of Omni OpenMP compiler system [7] shown in Figure 3.1

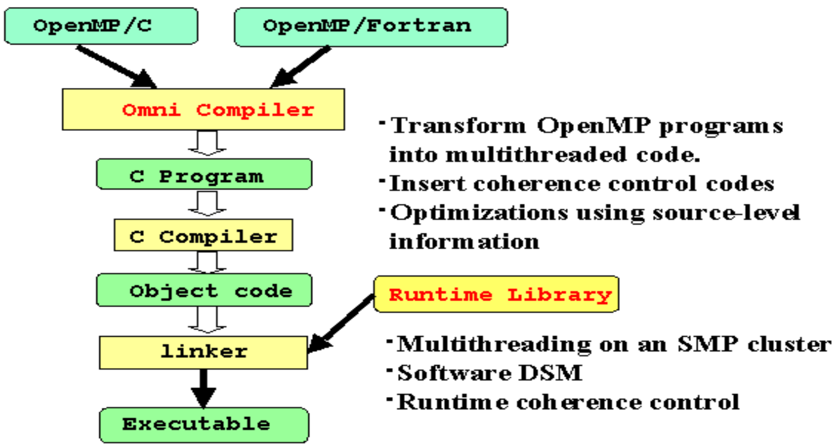

Figure 3.1: Omni OpenMP compiler system

3.2 Sun Fire 6800 Server

Sun Fire[tm] 6800 server is a powerful, highly available solution. Scaling up to 24 processors and featuring the redundantly configurable Sun[tm] Fire plane interconnect; this server delivers impressive total system performance. Fully hardware redundancy and a variety of advanced mainframe-class availability features, such as hot CPU upgrades and Dynamic Reconfiguration, provide maximum uptime. With Solaris Resource Manager[tm] software and Dynamic System Domains, this system has the flexibility to accommodate changing resource requirements where continuous uptime and availability are the key technologies. By using Dynamic Reconfiguration (DR), we can perform the following functions:

- Configure CPU/Memory boards into a running domain.
- Install new CPU/Memory boards in a domain.
- Hot-Swap CPU/Memory boards.
- Hot-Swap an I/O Assembly.
- Hot-Swap a PCI Card.
- Move a CPU/Memory board between Dynamic System Domains.
- Initiate parallel DR operations

3.3 System Description and Experimental Setup

3.3.1 System Description

In our study system we configure On the Sun Fire[tm] 6800 as a domain which with 8 CPU, 8GB main memory and setup by Solaris 8 (5.8) operation system. This NUMA machine built from 4-processor building blocks (“quads”) interconnected with a fast switch that delivers 9.6GB/sec. In each quad it is a UMA SMP. The Omni OpenMP is used for programs compiler system.

This studies we have chosen two styles of data communication code, the Twin Primes number problem which is called for independent style, Matrix Multiplication also for Independent style but we create matrix size by dynamic memory allocate, and Laplace’s Equation for the Nearest Neighbor update style.

These three programs are modified by manual from sequential code to parallel codes. In order to calculate the execution time some special codes were added. Such as time function, parallel begin and parallel loop end.

To compare the overhead and performance between different schedule type, we modified each program into parallel dynamic schedule and parallel static schedule model. In next section the result will point out the behavior between these two models.

3.3.2 Experimental Setup

According to the Omni OpenMP installation guideline [10] we setup the experimental system step by step. First, we download the Omni distribution, “Omni-1.4.tar.gz”, second, we download yacc to make the parser, third, we download Java Development Kit (JDK version 1.3.1) and finally we use Solaris thread as the thread library.

Since the Java development Kit (JDK) is preinstalled and the GNU bison or yacc also preinstalled too. On the Sun Fire[tm] 6800 system Omni OpenMP can be installed easily and safety without any modification. Following are our installed step and running commands.

- Get the Omni distribution,"Omni-1.4.tar.gz"
- Prepare following software to install and use the Omni compiler:
    - GNU bison or yacc to make the parser.
    - Java Development Kit, JDK (later than 1.1.3, 1.2.2 prefferred) or Kaffe (1.0.5 or later). (If you have no JDK, or don't want to use JDK, you can install using 'jexc'. See the Omni OpenMP Compiler Installation Notes.)
    - Thread library, Solaris thread or POSIX thread library, On SGI IRIX, sproc is also used.
- Unpack the distribution file.
- gzip -dc Omni-1.4.tar.gz | tar xvf -
- Change the current directory to Omni directory.
- cd Omni-1.4
- Run "configure".
- % configure

- Run "make" to compile the sources.
- % make
- Run "make install" to install the compiler.
- % make install

Make sure that the command path includes '*install_directory/bin*'. Suggest to use root user to install the Omni OpenMP.

The compiler command as bellow:

- Run omcc [driver-options] [compiler-options] filename

  Example compiler the default output file is 'a.out' :
- Run omcc prime_2_omp_t –lm

  Example run 8 means 8 threads:
- Run ./a.out 8

Chapter 4

Experimental Results and Discussion

4.1 Twin Primes

It's a very old fact that the set of primes is infinite and a much more recent and famous result (by Jacques Hadamard (1865-1963) and Charles-Jean de la Vallee Poussin (1866-1962)) that the law rules the density of primes

$$p(n) \sim \frac{n}{\log(n)}$$

A couple of primes (p,q) are said to be twins if q = p+2. Except for the couple (2,3), this is clearly the smallest possible distance between two primes. For Example (3,5), (5,7), (11,13), (17,19), (29,31),...(419,421),... are Twin Primes.

Through Table 4.1 to 4.2 and Figure 4.1 we find out the efficiency will reach 80% while upper bound is over 1000K. This result is caused from the density and prime gap become larger. Beside this, we used Modulo 30 algorithm to sieving Twin Primes in Twin Primes program [8]. No wonder it takes this feature.

As the Table 4.3 and Figure 4.2 shown that the overhead of dynamic schedule is depended on the threads number. The more threads and iterations get more overheads. It comes out bad performance than by using static schedule. The reason is default scheduling is static and running a simple round-robin block scheduling. It may cause load imbalance but in this Independent Model the data communication is fewer, each processor received only an integer of upper bound.

Table 4.1: The elapsed time (seconds) of Twin Primes by dynamic schedule with difference upper bound

Number of threads	1	2	4	8
1K	0.0497	0.0480	0.0483	0.0487
10K	0.0527	0.0531	0.0560	0.0612
100K	0.1014	0.1068	0.1239	0.1838
1000K	0.8894	0.7746	0.8625	1.3834
10000K	16.8861	10.4647	7.9914	12.4649
100000K	352.8218	193.5303	106.9369	99.9079

Table 4.2: The efficiency of Twin Primes by dynamic schedule in each upper bound

Number of threads		1	2	4	8
Efficiency E_p	1K	100.0000	51.4583	25.5694	12.6797
Efficiency E_p	10K	100.0000	49.6234	23.5268	10.7639
Efficiency E_p	100K	100.0000	47.4719	20.4600	6.8961
Efficiency E_p	1000K	100.0000	57.4092	25.7792	8.0362
Efficiency E_p	10000K	100.0000	80.6800	52.8258	16.9336
Efficiency E_p	100000K	100.0000	91.1541	82.4836	44.1436

Through the results of above tables we make the chart as Figure 4.1. It shows us that the efficiency is increased by the upper bound, but it is downside by the number of threads

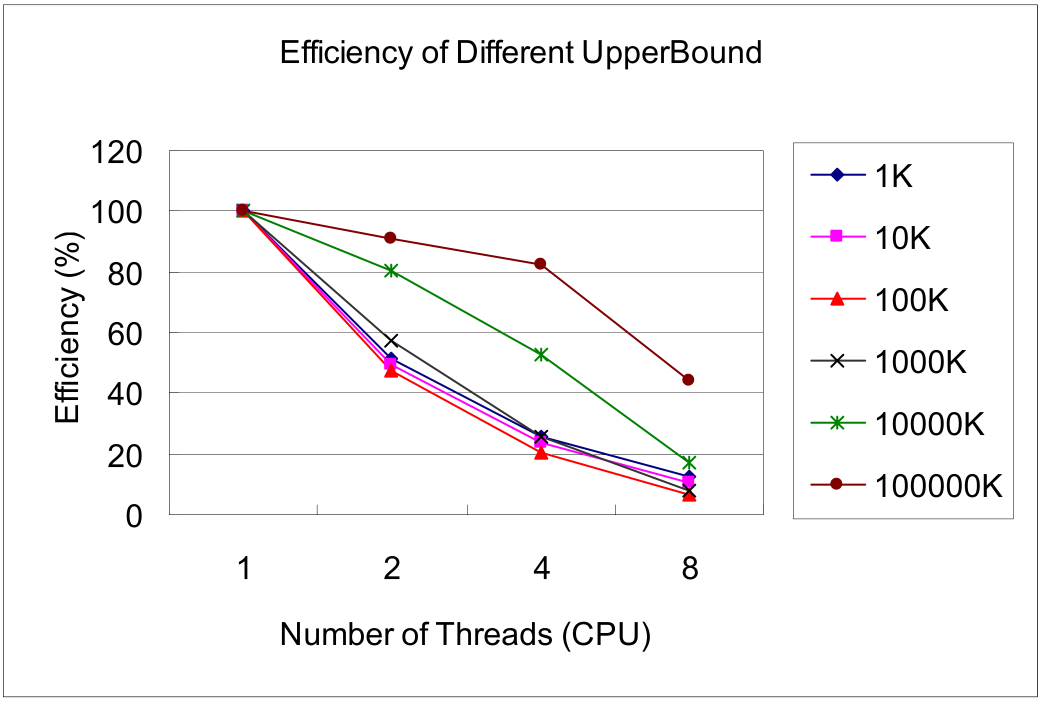

Figure 4.1: The efficiency of Twin Primes with different upper bound by dynamic schedule

Table 4.3: The overhead of Twin Primes by dynamic schedule in each upper bound

Number of threads		1	2	4	8
Total overhead/s	1K	0.0000	0.0233	0.0360	0.0425
Total overhead/s	10K	0.0000	0.0268	0.0428	0.0546
Total overhead/s	100K	0.0000	0.0561	0.0986	0.1711
Total overhead/s	1000K	0.0000	0.3299	0.6402	1.2722
Total overhead/s	10000K	0.0000	2.0217	3.7699	10.3541
Total overhead/s	100000K	0.0000	17.1194	18.7315	55.8052

The overhead results show as Figure 4.2 It shows us that the upper bound and the number of threads increase the overhead

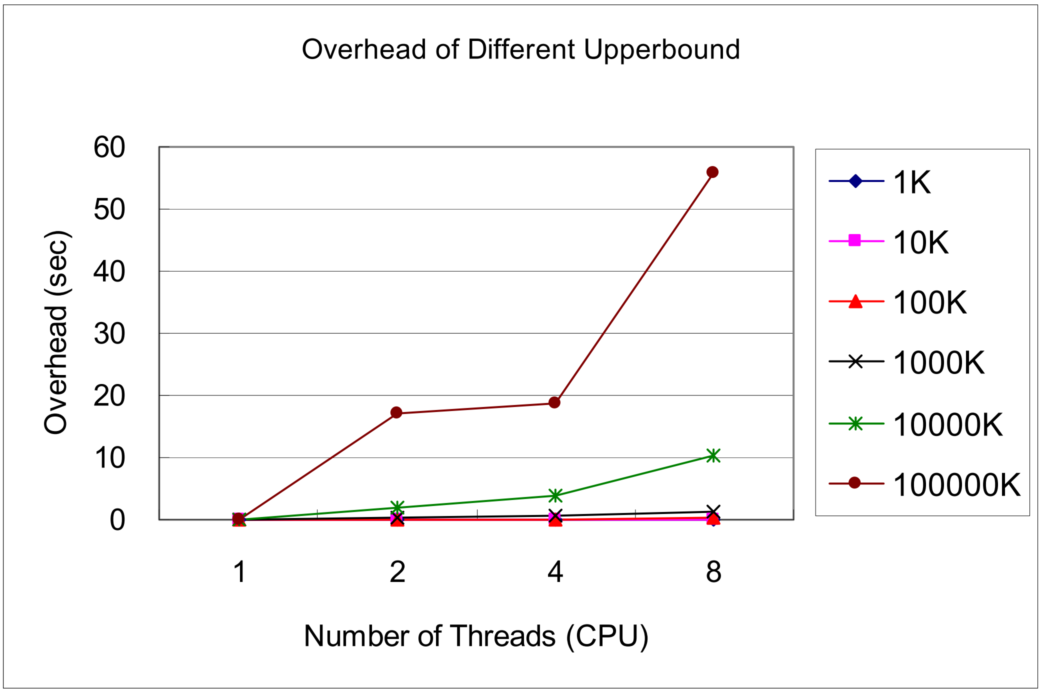

Figure 4.2: The overhead of Twin Primes with different upper bound by dynamic schedule

Table 4.4 is the elapsed time of Twin Primes by static schedule. The efficiency refer Table 4.5 and Figure 4.3, it is easy to find out the efficiency will reach 80% while upper bound is over 1000K. That's means the same model (Independent model) will give us the same performance ratio.

Table 4.4: The elapsed time of Twin Primes by static schedule in each upper bound

Number of threads	1	2	4	8
1K	0.0494	0.0495	0.0495	0.0497
10K	0.0527	0.0528	0.0512	0.0505
100K	0.1014	0.0786	0.0648	0.0579
1000K	0.8894	0.5141	0.2663	0.1374
10000K	16.8861	9.9592	5.2601	2.6720
100000K	352.8218	213.7587	113.4414	57.8288

Table 4.5: The efficiency of Twin Primes by static schedule in each upper bound

Number of threads		1	2	4	8
Efficiency E_p	1K	100.0000	49.8990	24.9495	12.4245
Efficiency E_p	10K	100.0000	49.9053	25.7324	13.0446
Efficiency E_p	100K	100.0000	64.5038	39.1204	21.8912
Efficiency E_p	1000K	100.0000	86.4989	83.4879	80.9142
Efficiency E_p	10000K	100.0000	84.7761	80.2558	78.9940
Efficiency E_p	100000K	100.0000	82.5280	77.7542	76.2643

Through the results of above tables we make the chart as Figure 4.3. It shows us that the efficiency is increased by the upper bound, but it is downside by the number of threads

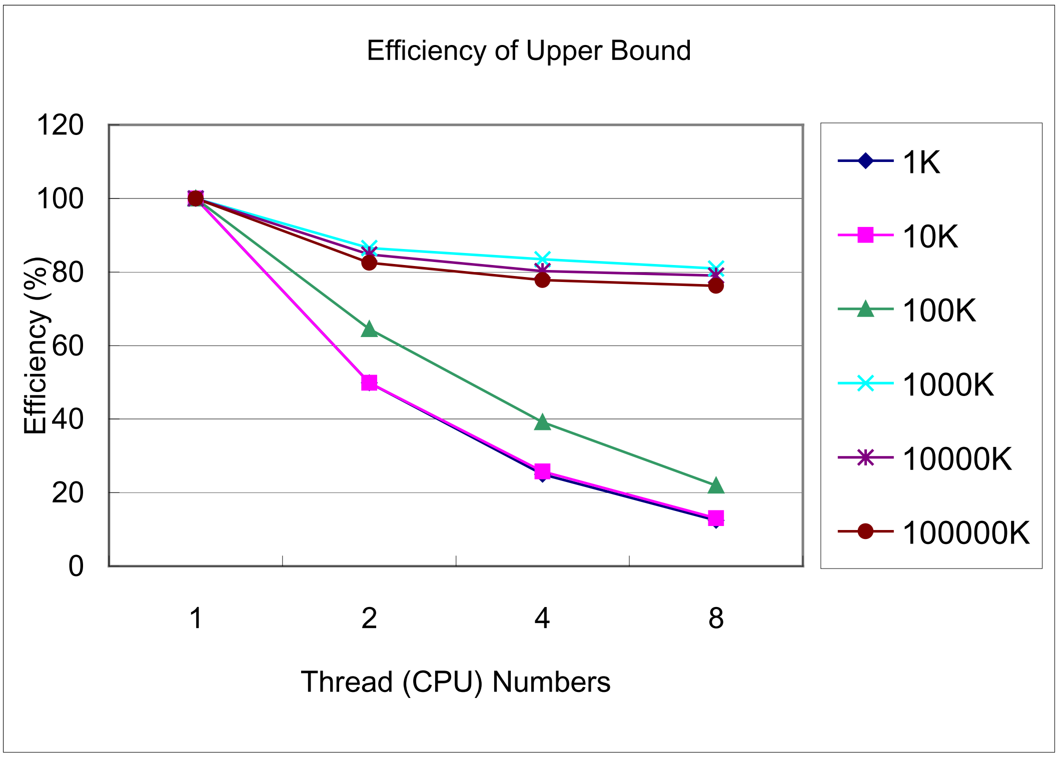

Figure 4.3: The efficiency of Twin Primes with different upper bound by static schedule

Table 4.6: The overhead of Twin Primes by static schedule in each upper bound

Number of threads		1	2	4	8
Total overhead/s	1K	0.0000	0.0248	0.0372	0.0435
Total overhead/s	10K	0.0000	0.0265	0.0380	0.0431
Total overhead/s	100K	0.0000	0.0279	0.0395	0.0452
Total overhead/s	1000K	0.0000	0.0694	0.0440	0.0262
Total overhead/s	10000K	0.0000	1.5162	1.0386	0.5613
Total overhead/s	100000K	0.0000	37.3479	25.2359	13.7261

The overhead result is shown as Table 4.6 and Figure 4.4. It tells us that the upper bound over 1000K the overhead will decrease by the number of threads.

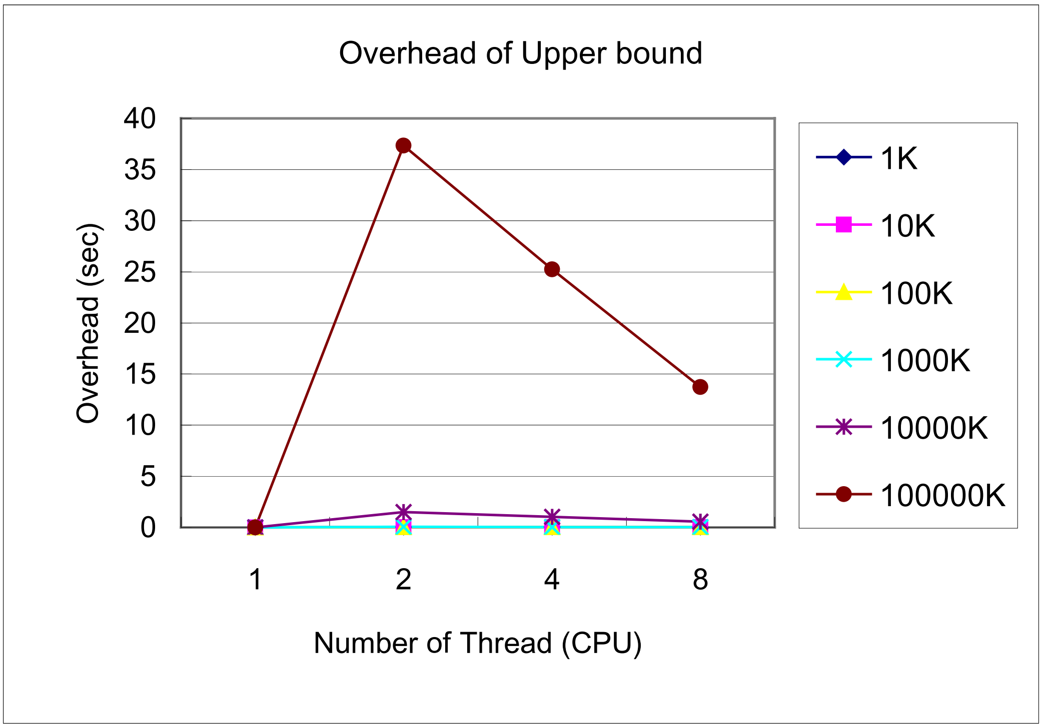

Figure 4.4: The overhead of Twin Primes with different upper bound by static schedule

Table 4.7: The efficiency ratio of Twin Primes by dynamic vs. static schedule

Number of threads	1	2	4	8
1K	-	103.1249	102.4846	102.0540
10K	-	99.4351	91.4287	82.5161
100K	-	73.5955	52.3001	31.5017
1000K	-	66.3699	30.8778	9.9318
10000K	-	95.1683	65.8218	21.4366
100000K	-	110.4523	106.0825	57.8824

While we compare the efficiency ratio results of dynamic vs. Static schedule see the Table 4.7 and Figure 4.5. It shows that when the upper bound over 10000K, the dynamic's efficiency ratio will better than static's, but once the number of threads reach 8 the ratio will down to 60%. That means dynamic scheduling will take idler threads for waiting synchronize.

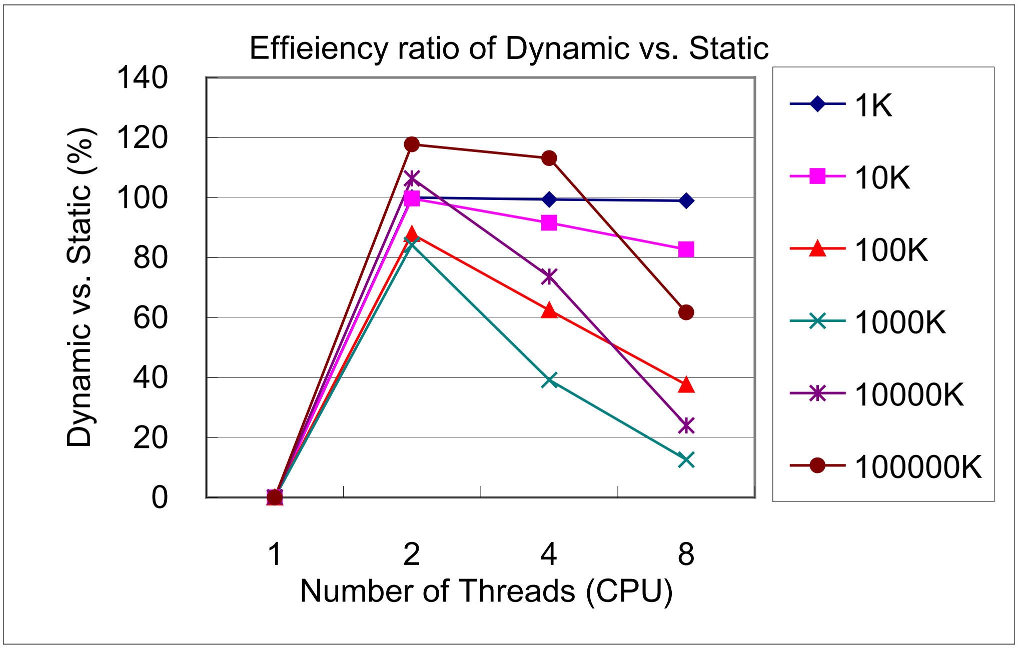

Figure 4.5: The efficiency ratio of Twin Primes by dynamic vs. static schedule

Table 4.8: The overhead ratio of Twin Primes by dynamic vs. static schedule

Number of threads	1	2	4	8
1K	-	93.9516	96.7742	97.7011
10K	-	101.1321	112.6316	124.3736
100K	-	201.0753	249.6203	378.5398
1000K	-	475.3602	1455.0000	4855.7252
10000K	-	133.3399	362.9790	1844.6642
100000K	-	45.8377	74.2256	406.5627

While we compare the overhead ratio results of dynamic vs. Static schedule see the Table 4.8 and Figure 4.6. It shows that when the upper bound over 10000K, the dynamic's overhead ratio will bad than static's, once the number of threads over 4 the ratio will up quickly, but will reach smoothly while the iteration is bigger than 1000K. That means dynamic scheduling will take idler threads for waiting synchronize.

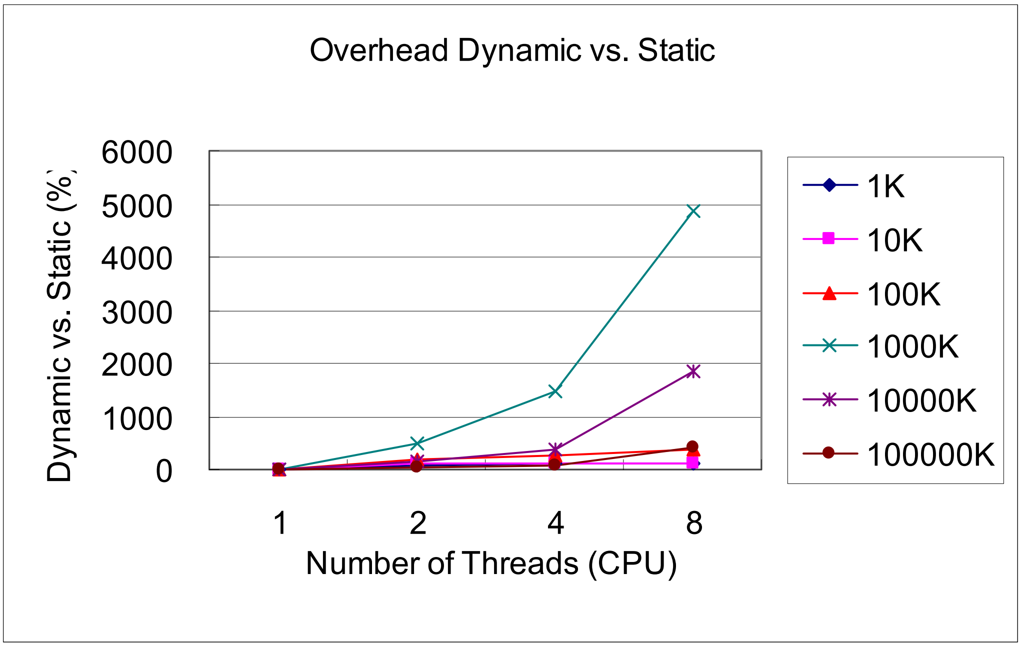

Figure 4.6: The overhead ratio of Twin Primes by dynamic vs. static schedule

4.2 Matrix Multiplication

The matrix operation derives a resultant matrix by multiplying two input matrices, **a** and **b**, where matrix **a** is a matrix of N rows by P columns and matrix **b** is of P rows by M columns. In the experiment we create matrix sizes by dynamic memory allocate. To initial two matrixes value into 5.0 and 2.0 that we can easily validate the final value.

From Table 4.9 to 4.10 and Figure 4.6, it shows the result in dynamic and static schedule model. Since we are used dynamic memory allocate to create matrix size. The memory is fixed at initial region. So the more iteration size will comes good efficiency. It is near to reach 100% efficiency. Specifically in dynamic schedule it comes out supper linear speed up performance (if we select the dynamic elapsed time of one thread as the T_s). Note that it will be possible for an overhead to be negative and thus relate to an improvement in the parallel performance. In our case, for a certain processors it may be possible that the data fits into cache when it does not for the serial implementation. In such case, the overhead due to data access would be negative.

Table 4.9: The elapsed time of Matrix Multiplication by dynamic schedule

Number of threads	1	2	4	8
128	0.1684	0.0894	0.0453	0.0232
256	1.4040	0.7429	0.3704	0.1864
512	11.3228	5.9034	2.9489	1.4684
1024	94.4611	48.0127	23.8313	11.9161
2048	1742.7566	880.4999	438.2902	218.7415
4096	16853.7397	8579.0476	4287.6328	2141.1396

Table 4.10: The efficiency of Matrix Multiplication by dynamic schedule

Number of threads		1	2	4	8
Efficiency E_p	128	100.0000	94.1834	92.9360	90.7328
Efficiency E_p	256	100.0000	94.4945	94.7624	94.1524
Efficiency E_p	512	100.0000	95.9007	96.0341	96.3872
Efficiency E_p	1024	100.0000	98.3710	99.0935	99.0898
Efficiency E_p	2048	100.0000	98.9640	99.4056	99.5900
Efficiency E_p	4096	100.0000	98.2262	98.2695	98.3786

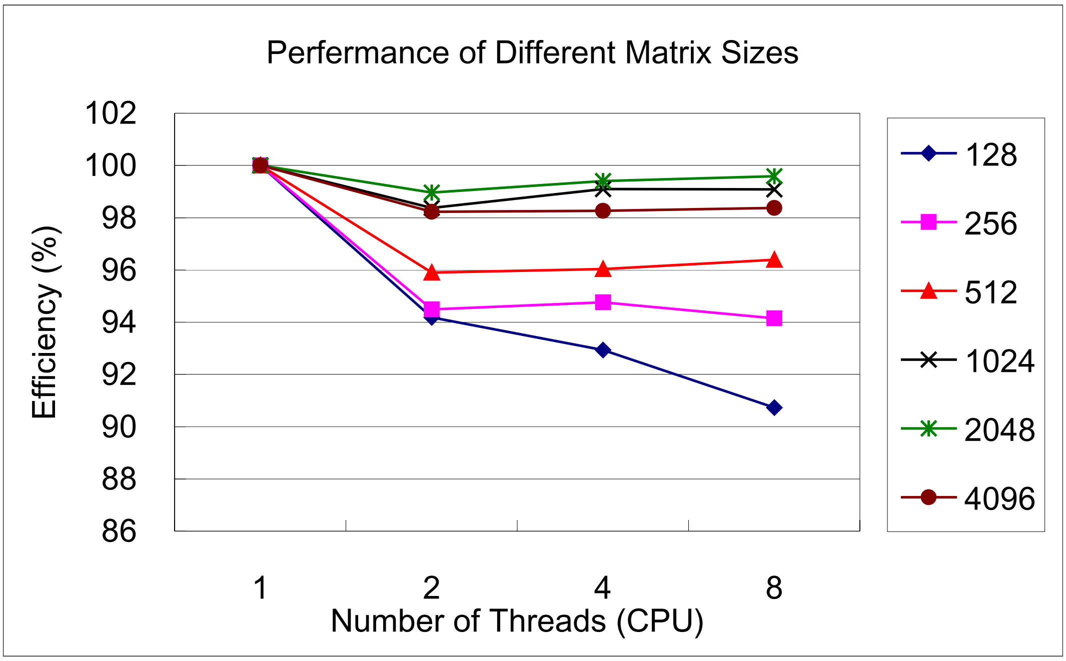

Figure 4.7: The efficiency of Matrix Multiplication by dynamic schedule

The overhead result is shown as Table 4.11 and Figure 4.8. It make the same result as Twin Primes.

Table 4.11: The overhead of Matrix Multiplication by dynamic schedule

Number of threads		1	2	4	8
Total overhead/s	128	0.0000	0.0052	0.0032	0.0022
Total overhead/s	256	0.0000	0.0409	0.0194	0.0109
Total overhead/s	512	0.0000	0.2420	0.1169	0.0530
Total overhead/s	1024	0.0000	0.7822	0.2160	0.1085
Total overhead/s	2048	0.0000	9.1216	2.6011	0.8969
Total overhead/s	4096	0.0000	152.1777	74.1979	34.7221

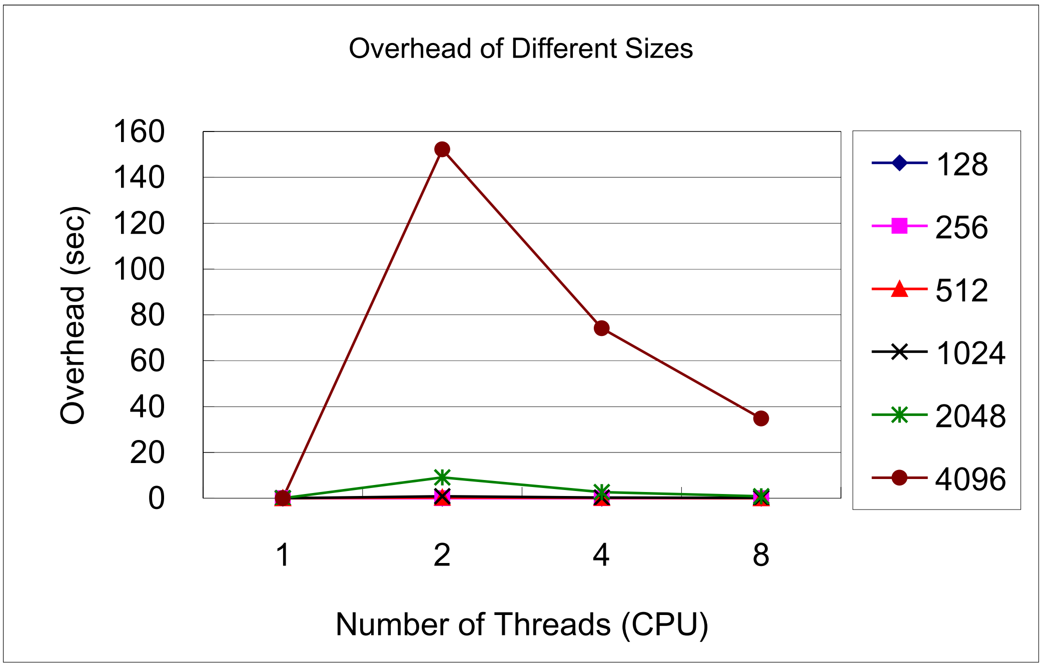

Figure 4.8: The overhead of Matrix Multiplication by dynamic schedule

During Table 4.12 to 4.14 and Figure 4.9 to 4.10, we found that the performance and overhead shown the same curve outline as dynamic's.

Table 4.12: The elapsed time of Matrix Multiplication by static schedule

Number of threads	1	2	4	8
128	0.1684	0.0902	0.0456	0.0233
256	1.4040	0.7449	0.3726	0.1872
512	11.3228	5.8813	2.9489	1.5073
1024	94.4611	48.3024	24.5135	12.0478
2048	1742.7566	869.3907	436.3162	217.9633
4096	16853.7397	8456.3749	4228.3545	2108.8773

Table 4.13: The efficiency of Matrix Multiplication by static schedule

Number of threads		1	2	4	8
Efficiency E_p	128	100.0000	93.3481	92.3246	90.3433
Efficiency E_p	256	100.0000	94.2408	94.2029	93.7500
Efficiency E_p	512	100.0000	96.2610	95.9917	93.8997
Efficiency E_p	1024	100.0000	97.7810	96.3358	98.0066
Efficiency E_p	2048	100.0000	98.4968	99.1256	99.2142
Efficiency E_p	4096	100.0000	99.6511	99.6472	99.8976

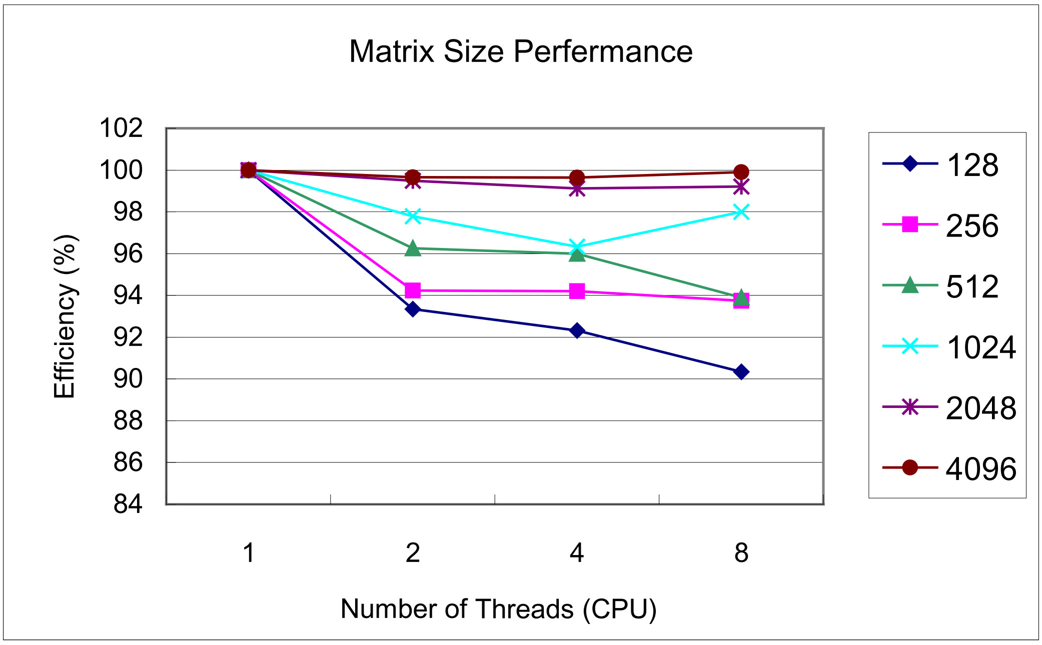

Figure 4.9: The efficiency of Matrix Multiplication by static schedule

Table 4.14: The overhead of Matrix Multiplication by static schedule

Number of threads		1	2	4	8
Total overhead/s	128	0.0000	0.0060	0.0035	0.0023
Total overhead/s	256	0.0000	0.0429	0.0216	0.0117
Total overhead/s	512	0.0000	0.2199	0.1182	0.0920
Total overhead/s	1024	0.0000	1.0719	0.8982	0.2402
Total overhead/s	2048	0.0000	13.2014	3.8150	1.7127
Total overhead/s	4096	0.0000	29.5050	14.9196	2.1598

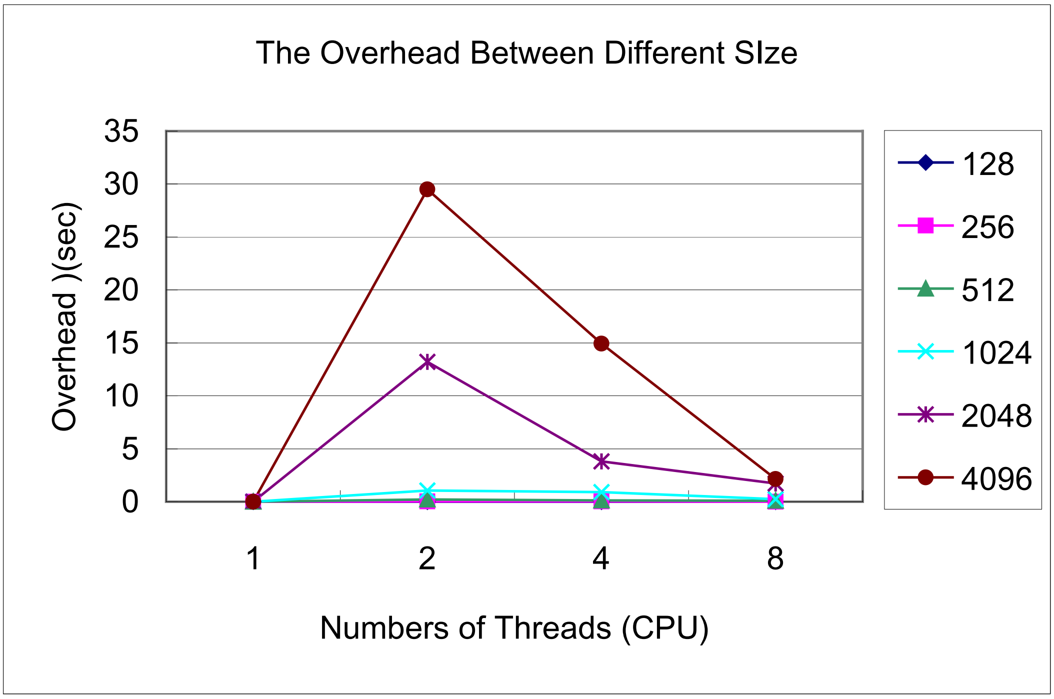

Figure 4.10: The overhead of Matrix Multiplication by static schedule

The ratio of efficiency present as slow down style. When the matrix size over 2048 dynamic has bad efficiency than static's. This may help us to know even dynamic memory allocated, static scheduling will take good performance when matrix size is larger.

Table 4.15: The efficiency ratio of Matrix Multiplication by dynamic vs. static schedule

Number of threads	1	2	4	8
128	100.0000	100.8948	100.6623	100.4311
256	100.0000	100.2692	100.5939	100.4292
512	100.0000	99.6257	100.0441	102.6491
1024	100.0000	101.0700	102.8626	101.1052
2048	100.0000	99.7392	99.5496	99.6443
4096	100.0000	98.5701	98.6175	98.4795

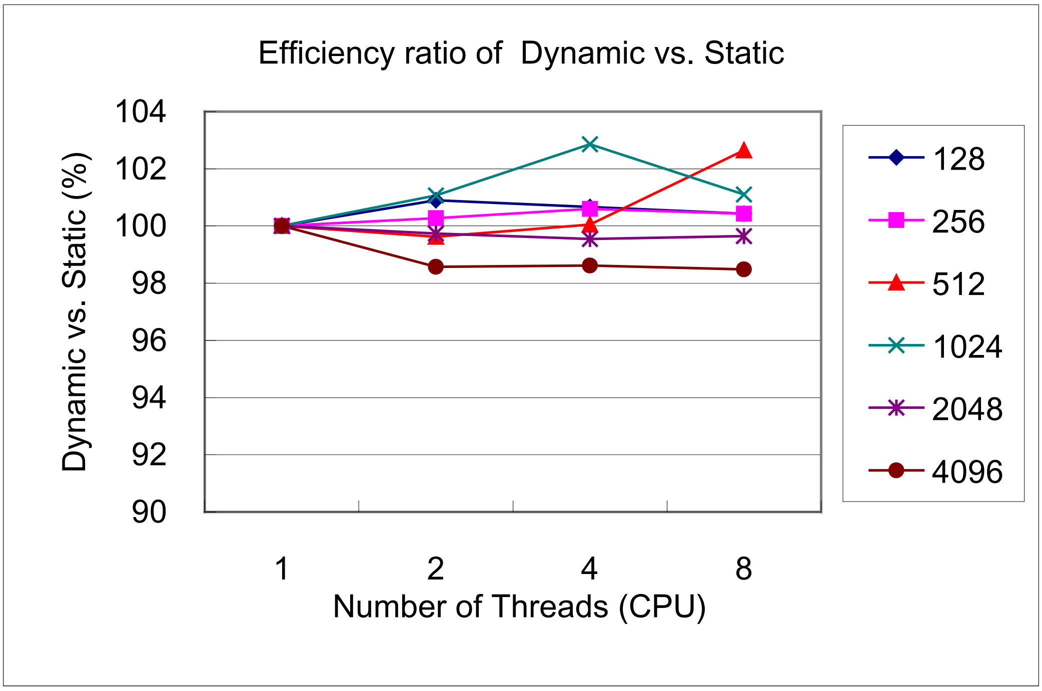

Figure 4.11: The efficiency ratio of Matrix Multiplication by dynamic vs. static schedule

Due to dynamic scheduling with chunk 1, for a larger number of processors it will take more overhead, it only get better overhead ratio on 1024 matrix size.

Table 4.16: The overhead of Matrix Multiplication by dynamic vs. static schedule

Number of threads		1	2	4	8
Total overhead/s	128	0.0000	86.6667	91.4286	97.7778
Total overhead/s	256	0.0000	95.3380	89.8148	93.1624
Total overhead/s	512	0.0000	110.0500	98.9002	57.6400
Total overhead/s	1024	0.0000	52.2834	24.0474	45.1777
Total overhead/s	2048	0.0000	133.6400	414.8485	755.6024
Total overhead/s	4096	0.0000	515.7684	497.3192	1607.6265

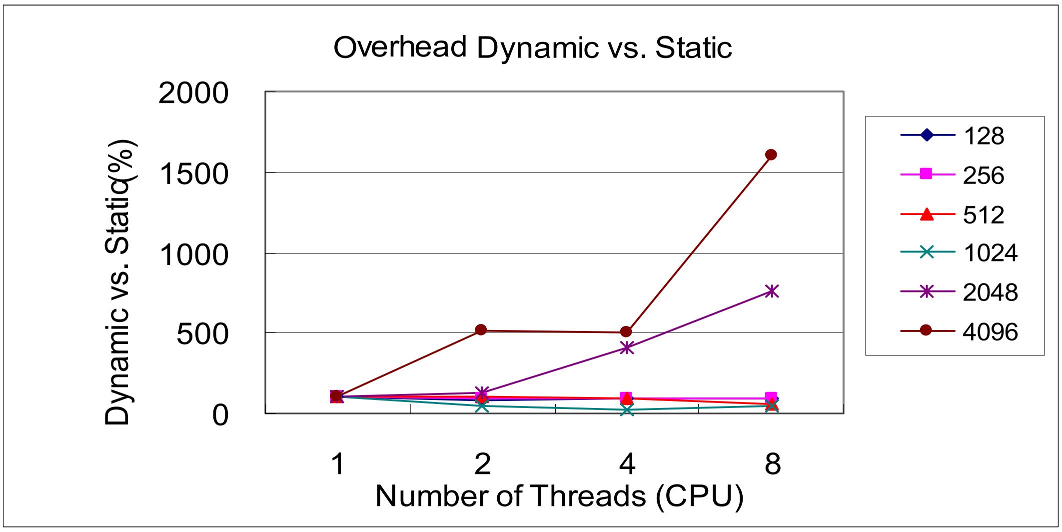

Figure 4.12: The overhead of Matrix Multiplication by dynamic vs. static schedule

4.3 Laplace's Equation

To solve Laplace's Equation we used OpenMP with explicit Jacobin iteration method [9]. It starts with an initial guess at the solution and iterates to a final solution by averaging over the values of the four nearest neighbors. This is called *successive over-relaxation.* Define a square gird consisting of points (x,y), and use Jacobin iteration to compute the value of u(x,y) at all the grid points.

In this style we select square matrix size 4000 as our evaluation program. Table 4.17 is shown that the result in dynamic schedule model. In static schedule the efficiency range from 80% to 100% see Table 4.18. It has little difference between iteration sizes. In dynamic schedule the efficiency are mostly the same between different iteration sizes. The curves almost overlap together refer Figure 4.13. This feature bring us to explore the behavior in Nearest Neighbor Model. The elapsed time and overhead in each loop should be same. Dynamic scheduling take nearest points with the same updated timing. The overhead increased by the iteration size.

Table 4.17: The elapsed time of Laplace's Equation by dynamic schedule

Number of threads	1	2	4	8
20	100.63	200.038	401.434	800.432
40	56.472	112.108	224.707	449.38
80	31.159	62.028	125.99	249.632
160	16.235	32.369	64.675	129.153

Table 4.18: The efficiency of Laplace's Equation by dynamic schedule

Number of threads		1	2	4	8
Efficiency E_p	20	100.0000	89.0973	80.7391	77.4792
Efficiency E_p	40	100.0000	89.2166	80.6241	77.2491
Efficiency E_p	80	100.0000	89.3239	79.6559	77.5868
Efficiency E_p	160	100.0000	89.0596	80.1612	77.4694

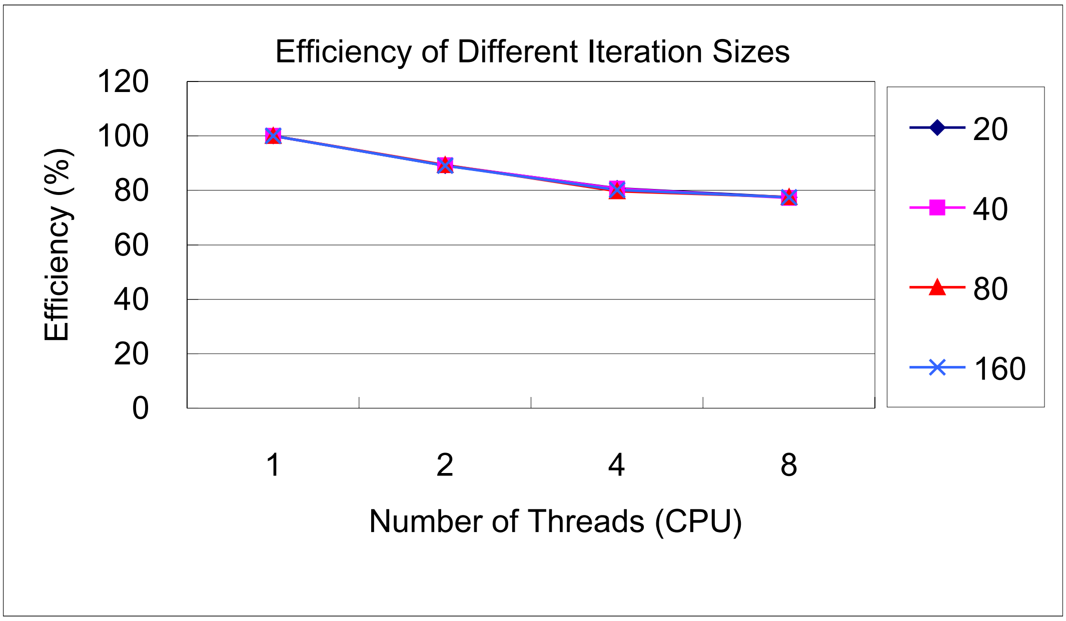

Figure 4.13: The efficiency of Laplace's Equation by dynamic schedule

Table 4.19: The overhead of Laplace's Equation by dynamic schedule

Number of threads		1	2	4	8
Total overhead/s	20	0.0000	6.1570	6.0015	3.6563
Total overhead/s	40	0.0000	12.0890	12.0185	7.3643
Total overhead/s	80	0.0000	23.9900	25.6315	14.4958
Total overhead/s	160	0.0000	49.1640	49.5240	29.0990

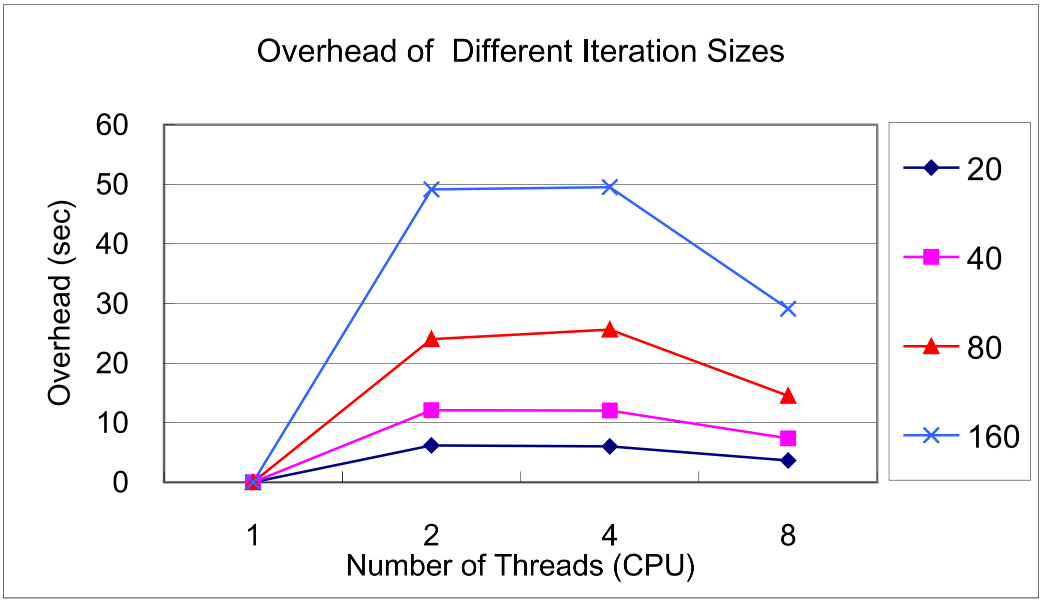

Figure 4.14: The overhead of Laplace's Equation by dynamic schedule

While in static scheduling the efficiency are mostly the same too (see Figure 4.15), but there are some gap in each iteration size. The only different is overhead increased by the number of threads without any slow down even it was ran on 8 threads.

Table 4.20: The elapsed time of Laplace's Equation by static schedule

Number of threads	1	2	4	8
20	100.63	200.038	401.434	800.432
40	50.704	100.66	203.913	406.489
80	26.558	51.844	105.133	208.573
160	13.695	27.287	55.018	108.641

Table 4.21: The efficiency of Laplace's Equation by static schedule

Number of threds p		1	2	4	8
Efficiency E_p	20	100.0000	99.2328	94.7266	91.8492
Efficiency E_p	40	100.0000	99.3632	96.4615	91.6361
Efficiency E_p	80	100.0000	98.4327	95.4586	91.2052
Efficiency E_p	160	100.0000	98.4568	95.9415	92.0960

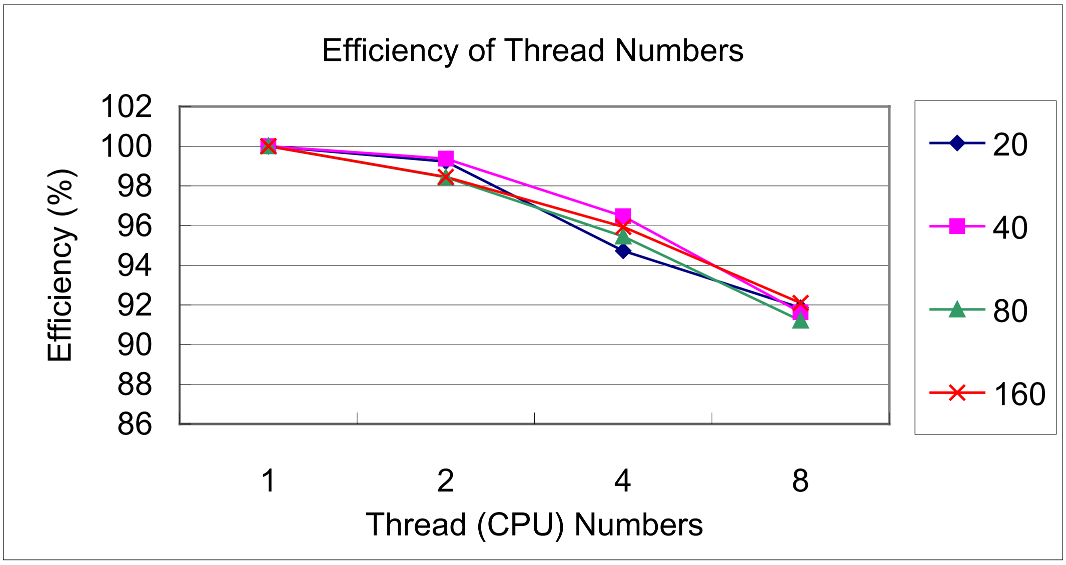

Figure 4.15: The efficiency ratio of Laplace's Equation by static schedule

Table 4.22: The overhead of Laplace's Equation by static schedule

Number of threads		1	2	4	8
Total overhead/s	20	0.0000	0.3890	1.4005	1.1163
Total overhead/s	40	0.0000	0.6410	1.8345	2.2823
Total overhead/s	80	0.0000	3.1960	4.7745	4.8388
Total overhead/s	160	0.0000	6.2730	8.4650	8.5870

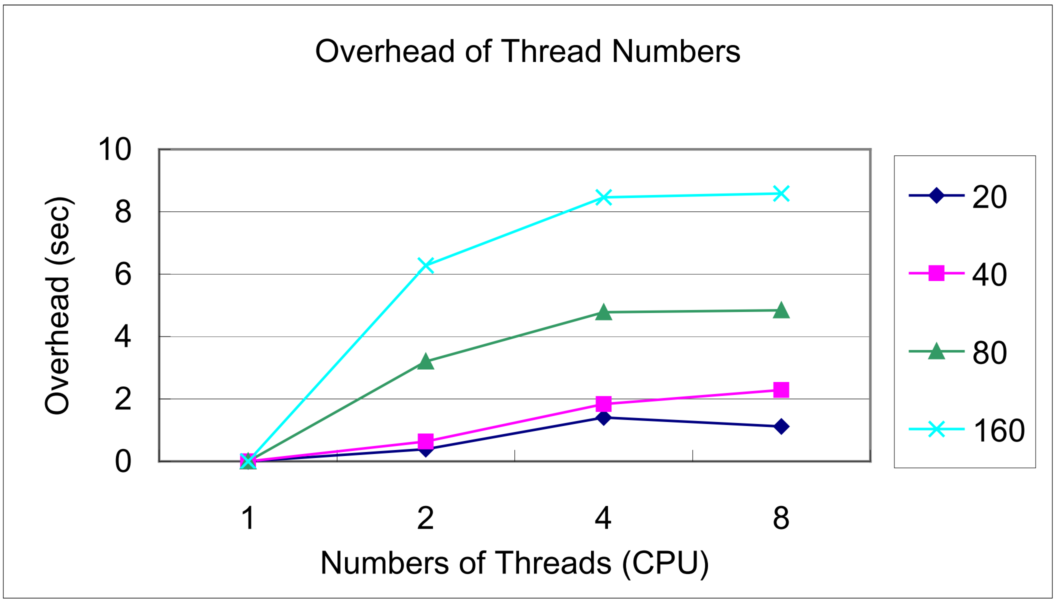

Figure 4.16: The overhead of Laplace's Equation by static schedule

The dynamic's performance is less than static's around 84% to 91 %.

Table 4.23: The efficiency ratio of Laplace's Equation by dynamic vs. static schedule

Number of threds p		1	2	4	8
Efficiency E_p	20	100.0000	89.7861	85.2338	84.3548
Efficiency E_p	40	100.0000	89.7884	83.5816	84.2998
Efficiency E_p	80	100.0000	90.7462	83.4455	85.0684
Efficiency E_p	160	100.0000	90.4555	83.5522	84.1181

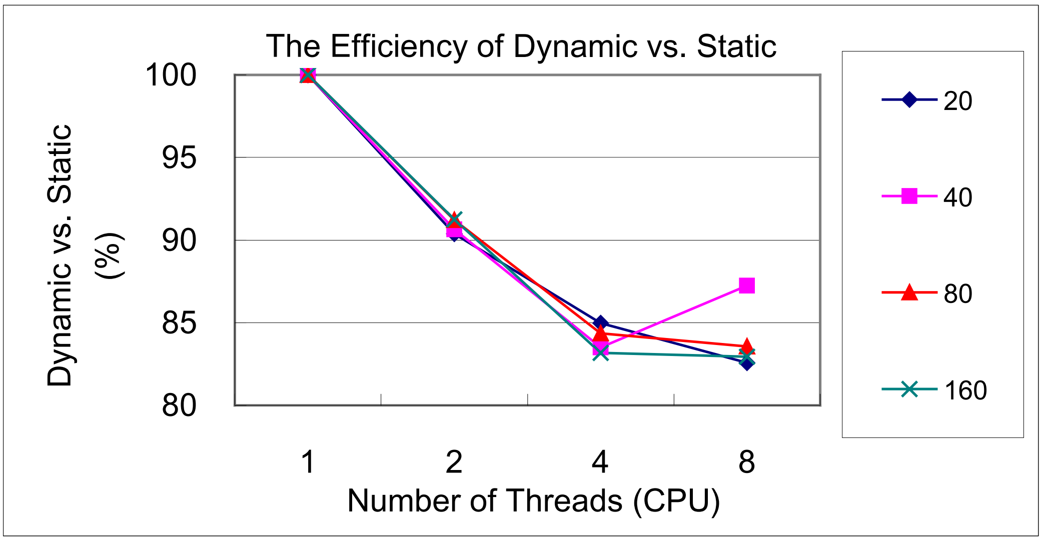

Figure 4.17: The efficiency ratio of Laplace's Equation by dynamic vs. static schedule

Table 4.24: The overhead ratio of Laplace's Equation by dynamic vs. static schedule

Number of threds p		1	2	4	8
Total overhead/s	20	0.0000	1582.7763	428.5255	327.5521
Total overhead/s	40	0.0000	1885.9594	655.1376	322.6772
Total overhead/s	80	0.0000	750.6258	536.8416	299.5774
Total overhead/s	160	0.0000	783.7398	585.0443	338.8727

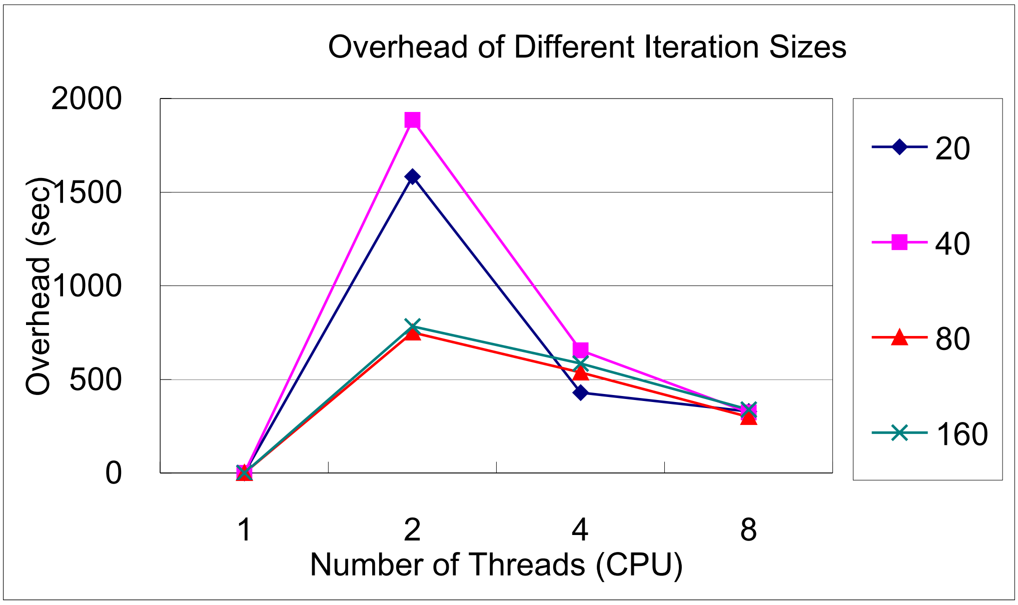

Figure 4.18: The overhead ratio of Laplace's Equation by dynamic vs. static schedule

4.4 Experimental of Twin Primes by Pthread Code

Table 4.25: The elapsed time (seconds) of Twin Primes by Pthread code in each upper bound

Number of threads	1	2	4	8
1K	0.0494	0.0523	0.0525	0.0532
10K	0.0527	0.0541	0.0565	0.0541
100K	0.1014	0.0794	0.0664	0.0604
1000K	0.8894	0.5094	0.2912	0.1752
10000K	16.8861	8.8294	4.6841	2.4168
100000K	352.8218	188.9959	100.4106	50.7818

Table 4.26: The efficiency of Twin Primes by Pthread in each upper bound

Number of threads		1	2	4	8
Efficiency E_p	1K	100.0000	47.2275	23.5238	11.6071
Efficiency E_p	10K	100.0000	48.7061	23.3186	12.1765
Efficiency E_p	100K	100.0000	63.8539	38.1777	20.9857
Efficiency E_p	1000K	100.0000	87.2971	76.3550	63.4548
Efficiency E_p	10000K	100.0000	95.6241	90.1244	87.3369
Efficiency E_p	100000K	100.0000	93.3411	87.8448	86.8475

Through the results of above tables we make the chart as Figure 4.19. It shows us that the efficiency is increased by the upper bound, but it is downside by the number of threads. This take the same feature as coding by OpenMP.

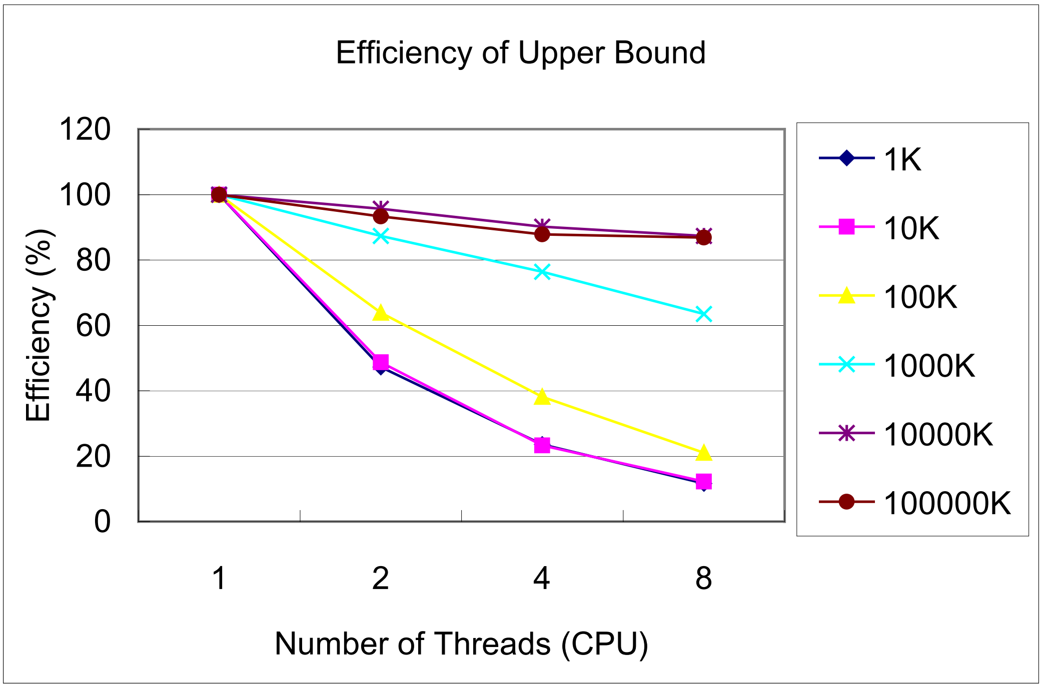

Figure 4.19: The efficiency of Twin Primes with different upper bound by Pthread

Table 4.27: The overhead of Twin Primes by Pthread in each upper bound

Number of threads		1	2	4	8
Total overhead/s	1K	0.0000	0.0276	0.0402	0.0470
Total overhead/s	10K	0.0000	0.0278	0.0433	0.0475
Total overhead/s	100K	0.0000	0.0287	0.0411	0.0477
Total overhead/s	1000K	0.0000	0.0647	0.0687	0.0640
Total overhead/s	10000K	0.0000	0.3864	0.4626	0.3060
Total overhead/s	100000K	0.0000	12.5850	12.2052	6.6791

The overhead results show as Figure 4.20 It shows us that the upper bound and the number of threads increase the overhead

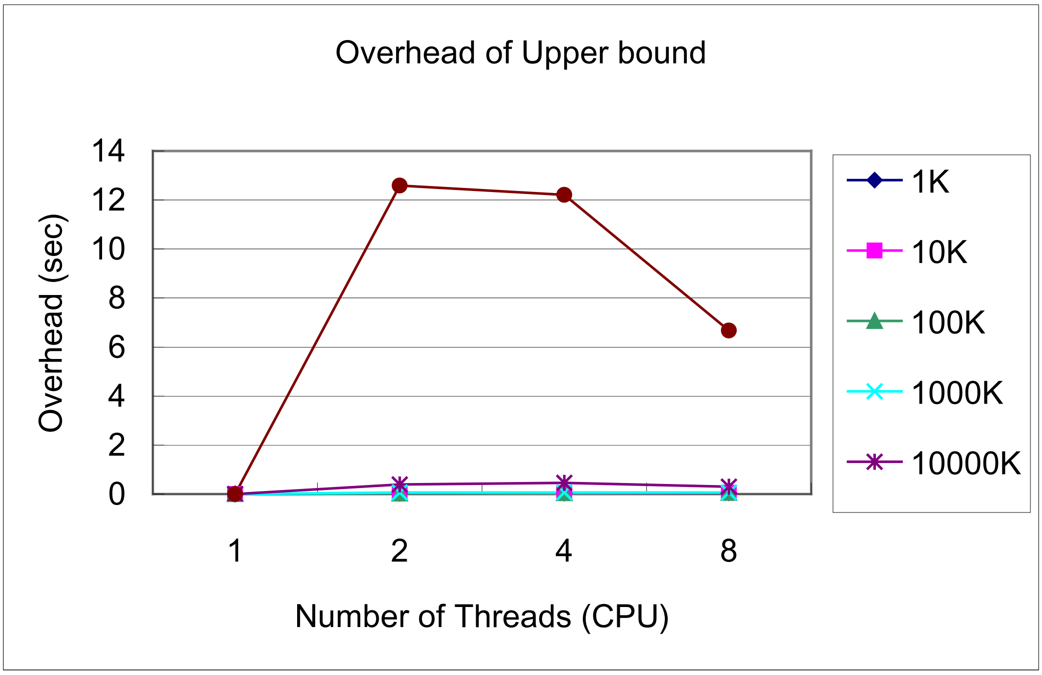

Figure 4.20: The overhead of Twin Primes with different upper bound by Pthread

4.5 Experimental of Matrix Multiplication by Pthread Code

The performance curves are as same as OpenMP's, But the start overhead(2 threads) is higher than other threads. The best performance and overhead are shown on 4 threads.

Table 4.28: The elapsed time of Matrix Multiplication by Pthread

Number of threads	1	2	4	8
128	0.1684	0.0941	0.0475	0.0246
256	1.4040	0.7693	0.3858	0.1934
512	11.3228	6.0868	3.0491	1.5277
1024	94.4611	49.3074	24.5992	12.6769
2048	1742.7566	875.6255	436.5643	219.3900
4096	16853.7397	8448.4465	4215.6423	2115.0166

Table 4.29: The efficiency of Matrix Multiplication by Pthread

Number of threads		1	2	4	8
Efficiency E_p	128	100.0000	89.4793	88.6316	85.5691
Efficiency E_p	256	100.0000	91.2518	90.9798	90.7446
Efficiency E_p	512	100.0000	93.0111	92.8372	92.6458
Efficiency E_p	1024	100.0000	95.7880	96.0002	93.1429
Efficiency E_p	2048	100.0000	99.5150	99.7995	99.2956
Efficiency E_p	4096	100.0000	99.7446	99.9476	99.6076

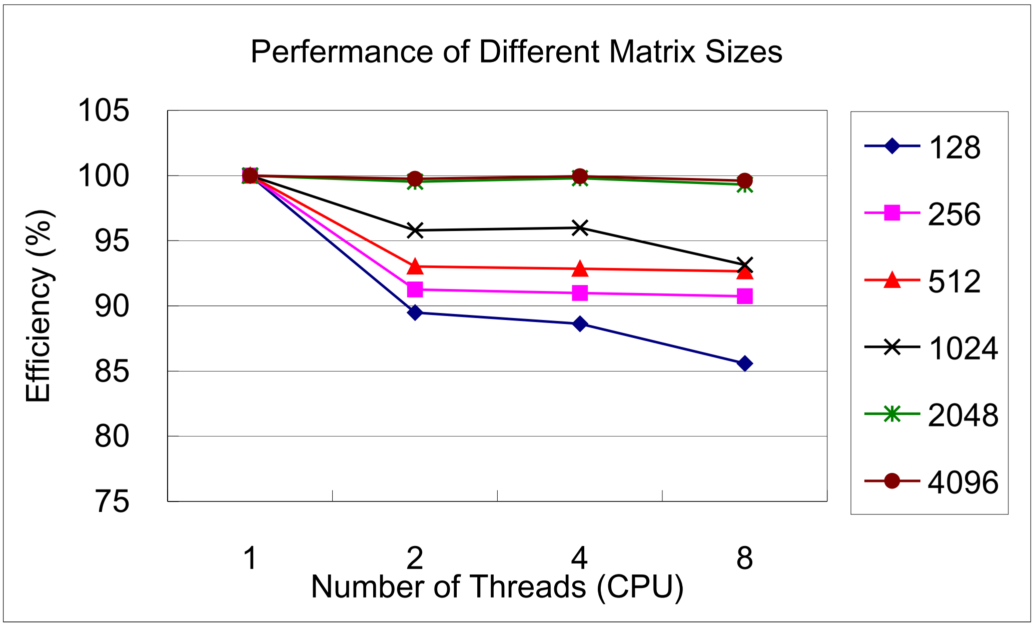

Figure 4.21: The efficiency of Matrix Multiplication by Pthread

Table 4.30: The overhead of Matrix Multiplication by Pthread

Number of threads		1	2	4	8
Total overhead/s	128	0.0000	0.0099	0.0054	0.0036
Total overhead/s	256	0.0000	0.0673	0.0348	0.0179
Total overhead/s	512	0.0000	0.4254	0.2184	0.1124
Total overhead/s	1024	0.0000	2.0769	0.9839	0.8693
Total overhead/s	2048	0.0000	4.2472	0.8752	1.5454
Total overhead/s	4096	0.0000	21.5766	2.2074	8.2991

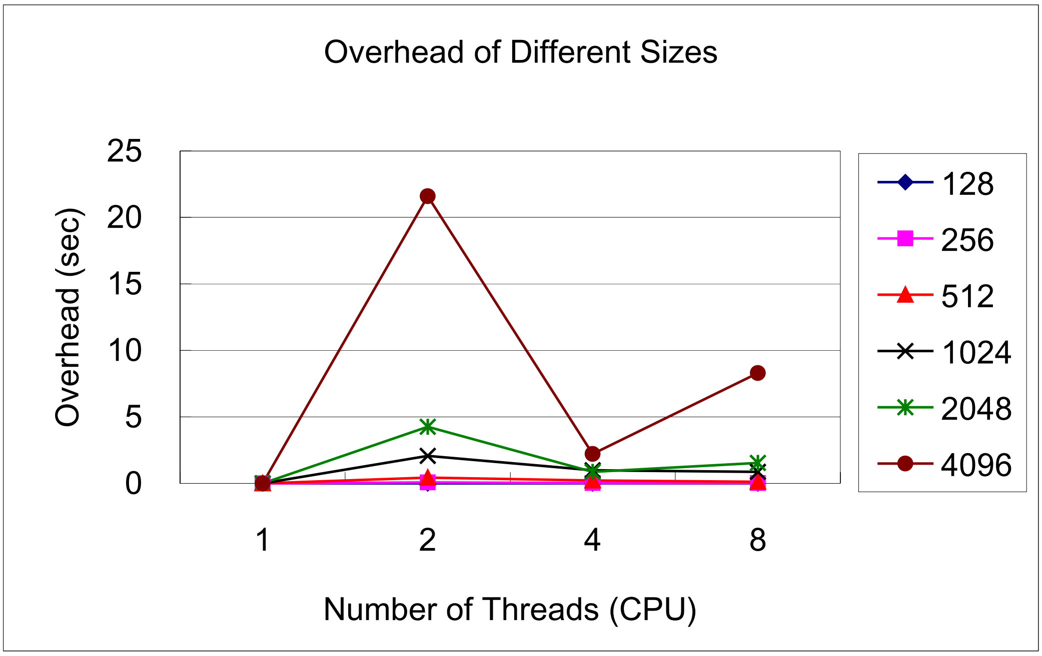

Figure 4.22: The overhead of Matrix Multiplication by Pthread

4.6 Experimental of Laprice's Eauation by Pthread Code

Table 4.31: The elapsed time of Laplace’s Equation by Pthread

Number of threads	1	2	4	8
20	100.63	54.018	27.316	14.899
40	200.038	107.916	54.711	29.603
80	401.434	215.618	109.474	57.945
160	800.432	430.102	218.073	116.516

Table 4.32: The efficiency of Laplace’s Equation by Pthread

Number of threds p		1	2	4	8
Efficiency E_p	20	100.0000	93.1449	92.0980	84.4268
Efficiency E_p	40	100.0000	92.6823	91.4067	84.4669
Efficiency E_p	80	100.0000	93.0892	91.6734	86.5981
Efficiency E_p	160	100.0000	93.0514	91.7619	85.8715

As shown on Table 4.31 to 4.33 the curves almost overlap together, it is similar as OpenMP’s.

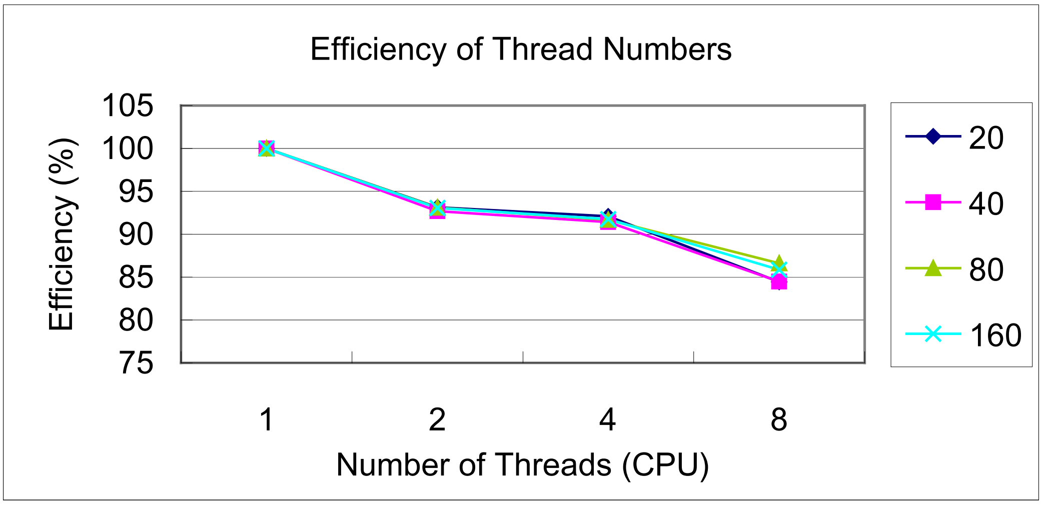

Figure 4.23: The efficiency of Laplace's Equation by Pthread

Table 4.33: The overhead of Laplace's Equation by Pthread

Number of threads		1	2	4	8
Total overhead/s	20	0.0000	3.7030	2.1585	2.3203
Total overhead/s	40	0.0000	7.8970	4.7015	4.5983
Total overhead/s	80	0.0000	14.9010	9.1155	7.7658
Total overhead/s	160	0.0000	29.8860	17.9650	16.4620

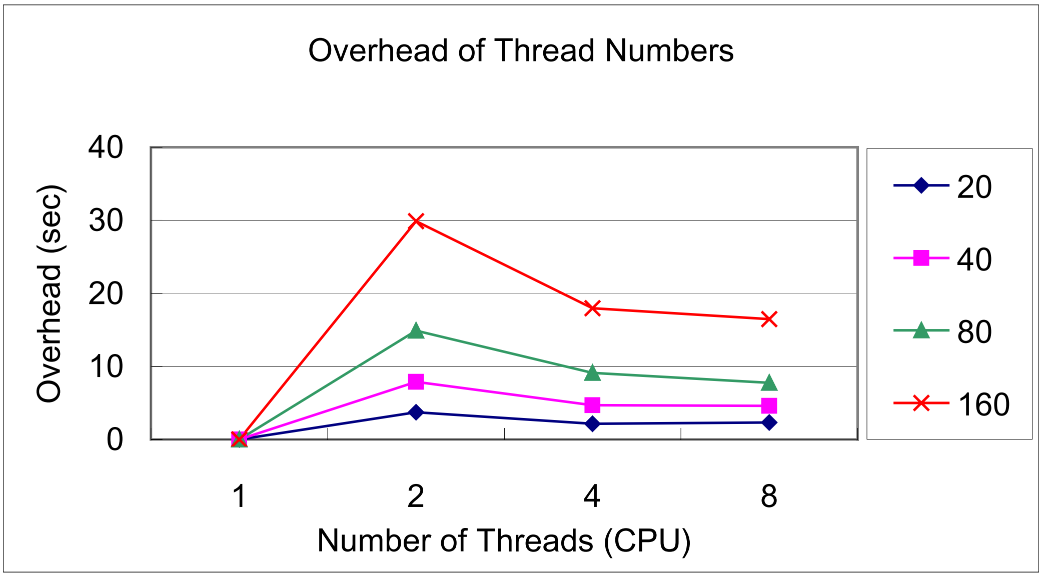

Figure 4.24: The overhead of Laplace's Equation by Pthread

4.7 Experimental OpenMP vs. Pthread

This section we will discuss the performance, which one is the better one between OpenMP and Pthread. In Twin Primes program the OpenMP have good performance until the upper bound is over 10000K. That's means Pthread can handle good threading on large iterations.

Table 4.34: The efficiency ratio of Twin Primes by Pthread vs. static schedule

Number of threads	1	2	4	8
1K	-	94.6462	94.2857	93.4211
10K	-	97.5970	90.6196	93.3451
100K	-	98.9925	97.5903	95.8636
1000K	-	100.9228	91.4564	78.4223
10000K	-	112.7961	112.2964	110.5614
100000K	-	113.1023	112.9776	113.8770

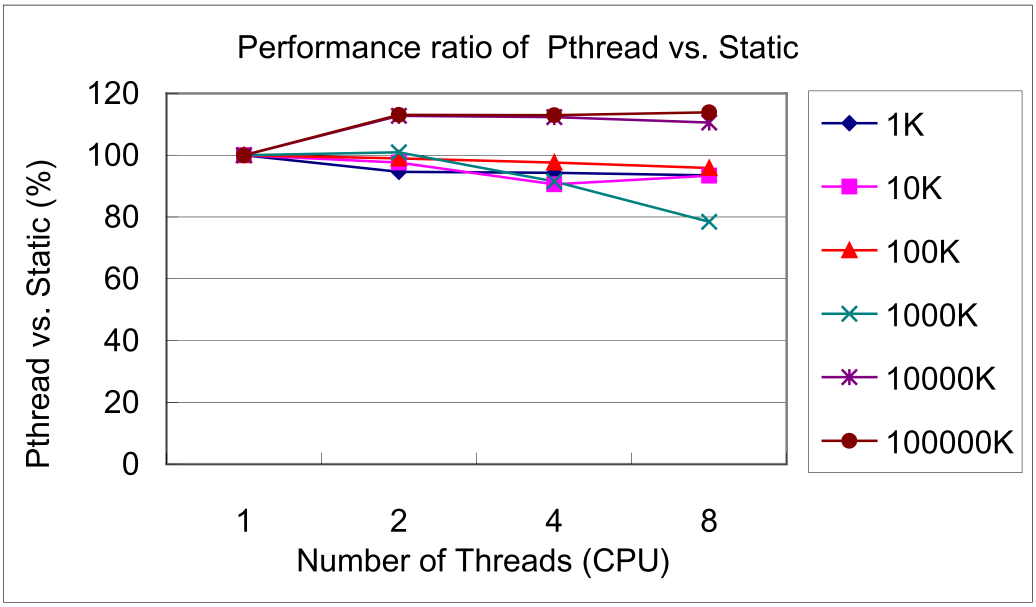

Figure 4.25: The efficiency ratio of Twin Primes by Pthread vs. static schedule

Table 4.35: The overhead ratio of Twin Primes by Pthread vs. static schedule

Number of threads	1	2	4	8
1K	-	111.2903	108.0645	108.0460
10K	-	104.9057	113.9474	108.2005
100K	-	102.8674	104.0506	105.5310
1000K	-	93.2277	156.1364	244.2748
10000K	-	25.4848	44.5407	54.5163
100000K	-	33.6967	48.3644	48.6599

While we compare the overhead ratio results of Pthread vs. Static schedule see the Table 4.35 and Figure 4.26. It shows that when the upper bound over 10000K, the Pthread's overhead ratio is better than static's, in this case means that the more iteration Pthread will spend more time for lock and synchronization.

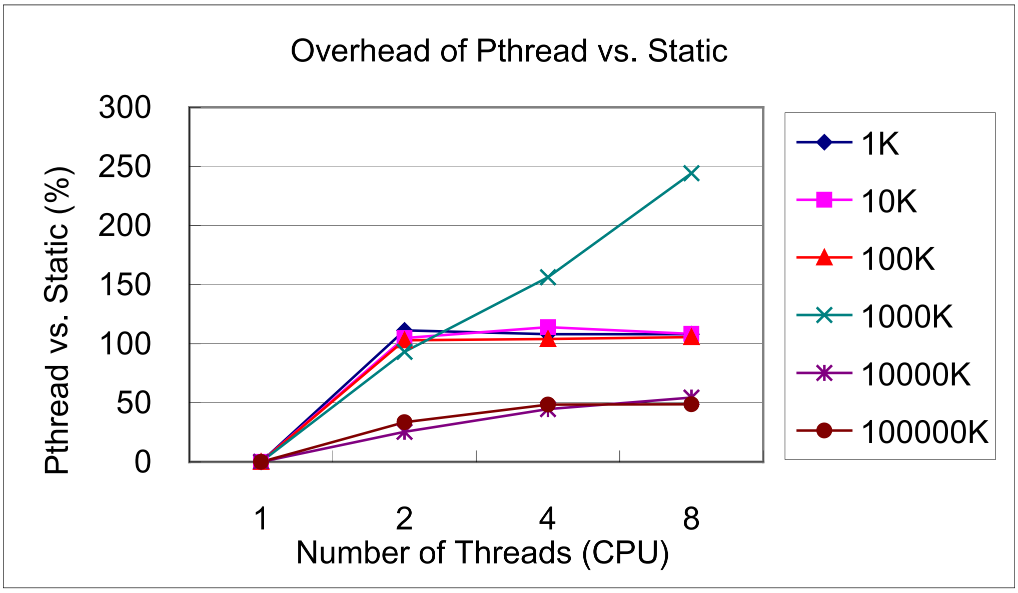

Figure 4.26: The overhead ratio of Twin Primes by Pthread vs. static schedule

Table 4.36: The efficiency ratio of Matrix Multiplication by Pthread vs. static schedule

Number of threads	1	2	4	8
128	100.0000	95.8555	96.0000	94.7154
256	100.0000	96.8283	96.5786	96.7942
512	100.0000	96.6238	96.7138	98.6647
1024	100.0000	97.9618	99.6516	95.0374
2048	100.0000	101.0337	100.6798	100.0820
4096	100.0000	100.0938	100.3015	99.7097

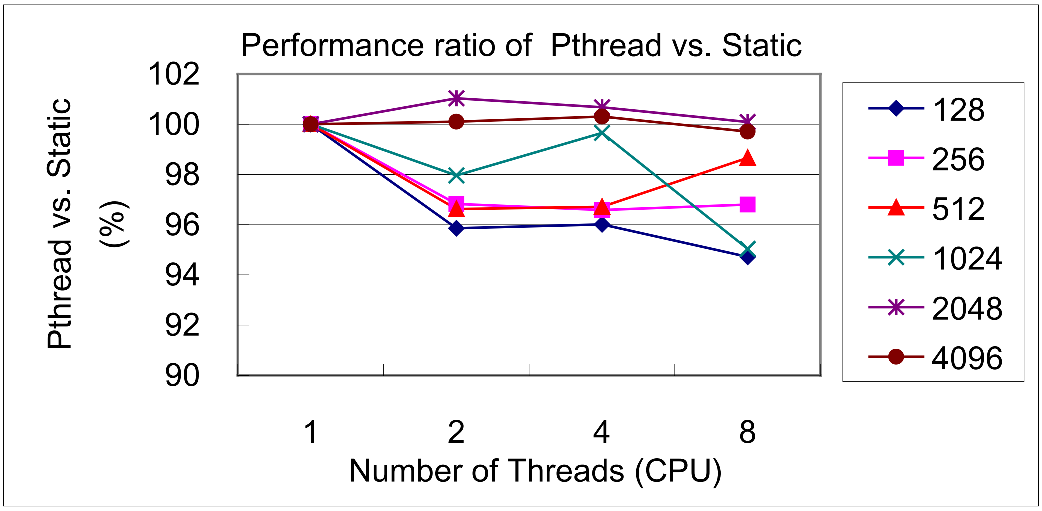

Figure 4.27: The efficiency ratio of Matrix Multiplication by Pthread vs. static schedule

Table 4.37: The overhead of Matrix Multiplication by Pthread vs. static schedule

Number of threads		1	2	4	8
Total overhead/s	128	0.0000	165.0000	154.2857	160.0000
Total overhead/s	256	0.0000	156.8765	161.1111	152.9915
Total overhead/s	512	0.0000	193.4516	184.7716	122.2403
Total overhead/s	1024	0.0000	193.7678	109.5383	361.9633
Total overhead/s	2048	0.0000	32.1723	22.9410	90.2318
Total overhead/s	4096	0.0000	73.1285	14.7953	384.2467

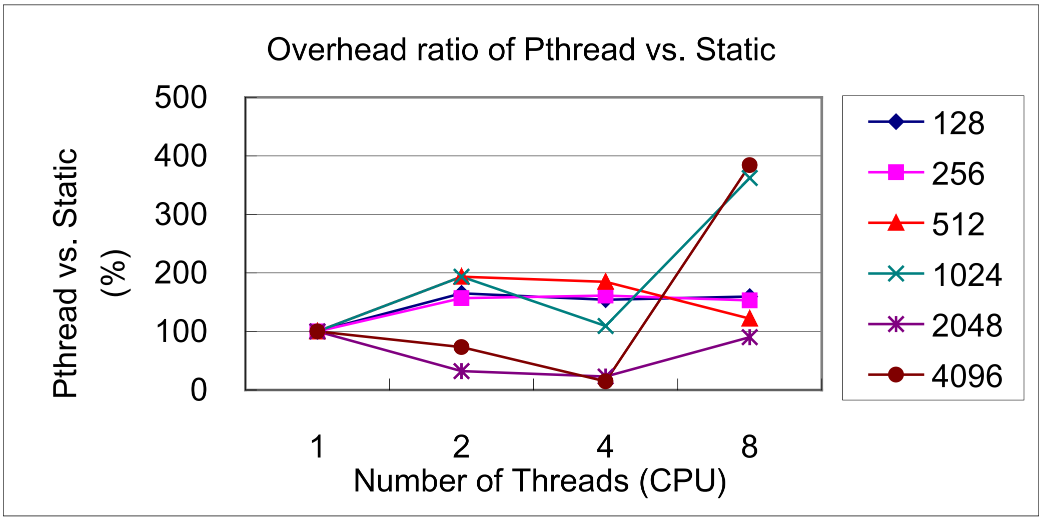

Figure 4.28: The overhead of Matrix Multiplication by Pthread vs. static schedule

For the Matrix Multiplication, the Pthread also get good performance by large parallel regions, such as matrix size over 2048. One the other hand the overhead is reduced by the parallel regions sizes. It has an out order overhead comes from running on 8 threads. That may be runtime environment is changed.

Most the same as run on the Pthread and OpenMP, no matter they are dynamic or static scheduling, the curves of performance are nearly to overlap. But this time, the OpenMP show the power of performance better than Pthread.. On the other hand the overhead of OpenMP is less than Pthread's.

Table 4.38: The efficiency ratio of Laplace's Equation by Pthread vs. static schedule

Number of threds p		1	2	4	8
Efficiency E_p	20	100.0000	93.8650	97.2250	91.9189
Efficiency E_p	40	100.0000	93.2763	94.7598	92.1764
Efficiency E_p	80	100.0000	94.5715	96.0347	94.9487
Efficiency E_p	160	100.0000	94.5099	95.6436	93.2413

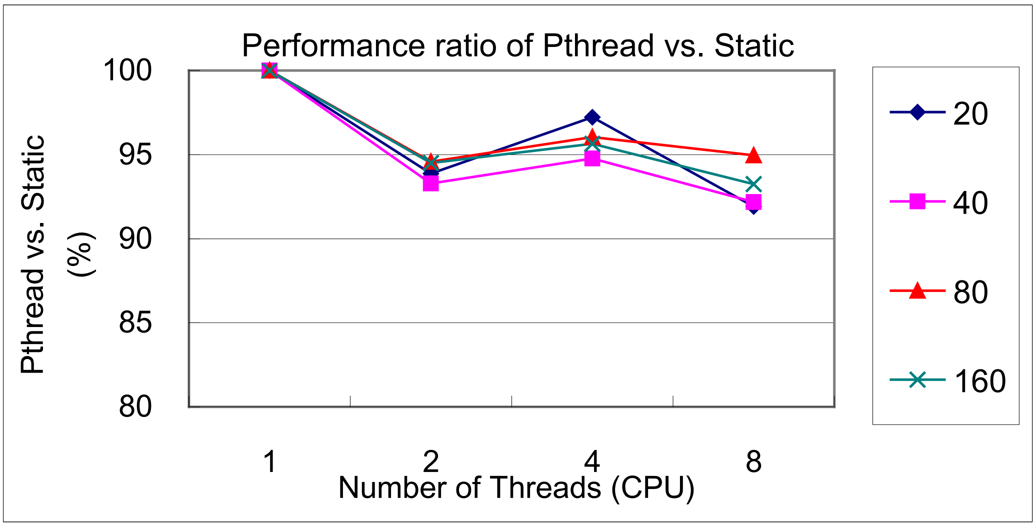

Figure 4.29: The efficiency ratio of Laplace's Equation by Pthread vs. static schedule

Table 4.39: The overhead ratio of Laplace's Equation by Pthread vs. static schedule

Number of thrreds p		1	2	4	8
Total overhead/s	20	0.0000	951.9280	154.1235	207.8656
Total overhead/s	40	0.0000	1231.9813	256.2824	201.4810
Total overhead/s	80	0.0000	466.2390	190.9205	160.4919
Total overhead/s	160	0.0000	476.4228	212.2268	191.7084

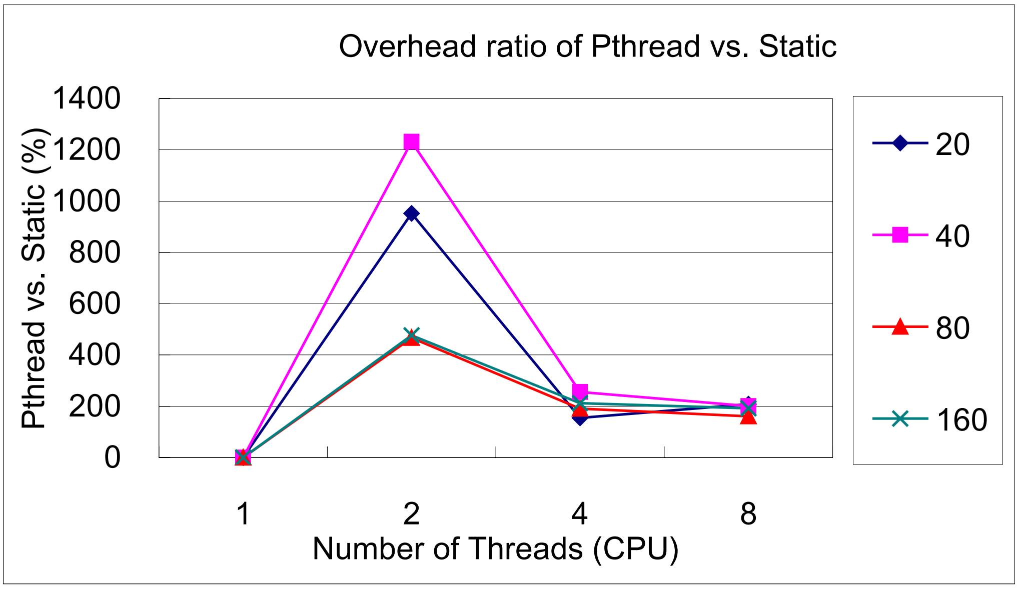

Figure 4.30: The overhead ratio of Laplace's Equation by Pthread vs. static schedule

Chapter 5

Conclusion and Future Work

OpenMP [14] is an API aimed for portable shared memory parallel programming which defines directives/pragmas, functions, and environment variables as an interface to the system. Currently, language bindings exist for Fortran, C, and C++. OpenMP performs reasonably well on all SMP systems. The overhead for starting up a parallel region was fairly high and programs which fork a parallel region for every Fine-grained parallel loop might have performance problems. Data placement and processor locality of data in non-UMA systems is an important aspect. There are tools available to gather information on the memory performance [15] which might help to optimize data locality although there are no language constructs in OpenMP to guide the compiler in generating processor locality. The static scheduling scheme has the lowest overhead as every thread is able to calculate from the known loop boundaries and the number of participating threads its own iteration space. Neither communication nor synchronized access to a global variable is necessary. On the other hand, static scheduling works only if there is an equal amount of work in the blocks the threads work on (leaving the aspect of data communication out of the discussion). Therefore, the cases with increasing and decreasing amount of work in the iterations for the static loop do not perform well. The dynamic scheduling scheme is usually implemented with atomic accesses to a shared variable which holds the loop count. For a (default) chunk size of 1 this means that every iteration involves an atomic access to the variable. As the OpenMP install guide mentioned, OpenMP have to call thread library, therefore we compare the performance with POXIS thread. However, only iteration size large enough the Pthread's performance will better than

OpenMP. Through the result of this thesis we find out some characters in overhead behavior. First, the same data communication style will take same overhead ratio. Second, the supper linear speed up is possible in some special problem and algorithm designed. Third, the dynamic schedule comes out bad performance than static schedule, unless we can rearrange chunk size and design a powerful machine or to prefix the data into dynamic memory or cache memory. In future we will study our case on PC cluster SMP system that will take more challenge in overhead reduction.

References

[1] B. McGarvey, R. Cicconetti, N. Bushyager, E. Dalton, M. Tentzeris, "Beowulf Cluster Design for Scientific PDE Models," Proc. of the 2001 Annual Linux Showcase, Oakland, CA, November 2001.

[2] M.K. Bane and G.D. Riley: "Automatic Overheads Profiler for OpenMP Codes" Research Thesiss presented in EWOMP2000.

[3] Y.Charlie Hu, Honghui Lu, Alan L. Cox and Willy Zwaenepoel "OpenMP for Networks of SMPs" Journal of Parallel and Distributed Computing,

[4] The OpenMP forum OpenMP C and C++ Application Program Interface, Version 2.0, http://www.OpenMP.org, DEC. 2002.

[5] Omni RCWP OpenMP Compiler projects, http://phase.etl.go.jp/Omni/omni.htm

[6] The Real World Computing Partnership, http://www.rwcp.or.jp/lab/pdslab/dist.

[7] Mitsuhisa sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP compiler for SMP cluster. The 1st European Workshop on OpenMP (EWOMP'99)

[8] Thomas R. Nicely. "Counts of twin prime pairs and Brun's constant to 3*10^5" http://www.trincely.net/twins/tabpi2.html.

[9] Solution of Laplace's Equation by using Jacobi Iteration, http://www.dartmouth.edu/~rc/classes/intro_mpi/laplaces_eqn.html.

[10] Omni OpenMP installation guideline, http://phase.etl.go.jp/Omni/Omni-doc/Install.html

[11] Omni/SCASH: Cluster-enabled Omni OpenMP on software distributed shared memory system SCASH, http://phase.etl.go.jp/Omni/Omni-doc/omni-scash.html

[12] OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP, http:// ninf.apgrid.org/papers/wompat01sato/WOMPAT01.pdf

[13] E.Tentzeris, R.Robertson, A.Cangellaris and L.P.B.Katehi, "Space and time adaptive gridding using MRTD",*Proc. of the 1997 IEEE Symposium on Microwave Theory and Technique*s, pp.337-340, Denver, CO.

[14] Leonardo Dagum and Ramesh Menon OpenMP: "An industry-standard API for shared-memory programming" *IEEE Computational Science and Engineering*, pages 46–55, January 1998.

[15] PCL – the performance counter library version1.2. http://www.fz-juelich.de/zam/PCL/, July 1999