TASK PARALLEL IMPLEMENTATION OF NAS PARALLEL BENCHMARKS

Shashi Kumar Nanjaiah
B.E, Visveswaraiah Technological University, Karnataka, India, 2005

PROJECT

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY, SACRAMENTO

SPRING
2010
TASK PARALLEL IMPLEMENTATION OF NAS PARALLEL BENCHMARKS

A Project

by

Shashi Kumar Nanjaiah

Approved by:

__________________________________, Committee Chair
Dr. Chung E Wang

__________________________________, Second Reader
Dr. Ted Krovetz

___________________________________
Date
Student: Shashi Kumar Nanjaiah

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the Project.

__________________________, Graduate Coordinator       ________________
Dr. Cui Zhang                                   Date

Department of Computer Science
Abstract

of

TASK PARALLEL IMPLEMENTATION OF NAS PARALLEL BENCHMARKS

by

Shashi Kumar Nanjaiah

The multi-core era brings new challenges to the programming community. Parallelization requirements of applications in mainstream computing and applications in emergent fields of high performance computing, such as informatics, must be explored. With parallelism now ubiquitous, programmability, composability, and reuse need to be closely examined in applications developed using existing parallel programming tools. Of the available programming models, task parallelism is the most promising and is able to parallelize regular and irregular applications alike [1, 2, 3]. In order to widely adopt task parallelism it is necessary for a tool to be configurable and extensible without any runtime penalties. Recently, PFunc [4], a novel library for expressing shared-memory task parallelism in C and C++ has been released to the open-source community. PFunc is unique in its heavy use of generic programming - a programming paradigm for developing efficient and reusable software libraries. PFunc can be used as is and does not require any configuration or extensions. It introduces the ability to deliver multiple task completion notifications and the notion of task groups and also provides production-grade exception handling and performance monitoring mechanisms to assist software developers. However, to validate PFunc’s utility, it is necessary to demonstrate its ability
to deliver parallel performance for a variety of applications. In this thesis, we present a
task parallel version of three NAS parallel benchmarks: Gaussian random variates (EP),
integer sort (IS), and conjugate gradient (CG) [5]. These three benchmarks are then
compared against their OpenMP counterparts.

_______________________, Committee Chair
Dr. Chung E Wang

_______________________
Date
ACKNOWLEDGMENTS

Appreciation is due to many people who have helped, often without knowing they were doing so, for inspiration, exhortation, and consolation throughout the time it has taken to create this work. Although the list of individuals I wish to thank extends beyond the limits, I am thankful to Dr. Chung E Wang, with the Department of Computer Science, California State University, Sacramento, for giving me the opportunity and guiding me throughout the project. I wish to thank Dr. Ted Krovetz for being the Second Reader. I also thank Dr. Cui Zhang for her ideas and suggestions during the course of the project. I would personally like to thank my friends, Pradeep Manjuraj, Chetan Krishna, Melanie Dybvig, Pramukh Jadhav, Deepak Gujjar and Shwetha Jain for their invaluable support and friendship that helped to get through some of the tough times.

There are four chief perpetrators who managed to have me submit my thesis. First and foremost, my mother Laxmi, her faith in my capabilities is unnerving. Second, my father Nanjaiah, his expectations in my achievements is boundless. Thirdly, the most beautiful women in my life Devika, always soothing with her limitless love. They all made me believe getting masters degree is just what most sane people opt for – which now I think is a complete lie. Amma sorry for taking so long. I am very thankful to you all for your love and encouragement. All I can say is, it would take one more thesis to express my deep love, particularly in the days when I spent more time with my computer than with you. Finally the fourth perpetrator and my primary advisor, Dr. Prabhanjan Kambadur. He is one of the most significant persons in my life and more than a friend over the years. I am fortunate to work with such an esteemed scholar. He is the one who made me sit down and write this thesis today and definitely he’s been the driving force behind the completion of this project.
TABLE OF CONTENTS

Page

Acknowledgements ........................................................................................................ vi
List of Tables .................................................................................................................. x
List of Figures ................................................................................................................ xi

Chapter

1. BACKGROUND AND INTRODUCTION ............................................................... 1
   1.1 A Need to Validate Performance of Parallelized Applications ....................... 1
   1.2 Motivation ......................................................................................................... 2
   1.3 Roadblocks ........................................................................................................ 2
   1.4 Project Organization ......................................................................................... 4

2. OVERVIEW OF TASK PARALLELISM AND NAS PARALLEL
   BENCHMARKS ............................................................................................................. 5
   2.1 Task Parallelism ................................................................................................ 5
   2.2 NAS Parallel Benchmarks ................................................................................ 7
   2.3 NAS Parallel Benchmark Components ............................................................. 8

3. OVERVIEW OF PFUNC ......................................................................................... 10
   3.1 Introduction ..................................................................................................... 10
   3.2 Software Architecture ..................................................................................... 11
   3.3 Components of PFunc ..................................................................................... 12
6.2 Problem Definition........................................................................................................ 27
6.3 Sequential Key Generation Algorithm........................................................................ 28
6.4 Parallel Implementation............................................................................................... 29
6.5 Results......................................................................................................................... 30

7. IMPLEMENTATION OF COIN-CHANGING PROBLEM ...................................... 32
7.1 Introduction................................................................................................................. 32
7.2 Examples of Backtracking......................................................................................... 32
7.3 Coin Changing Problem......................................................................................... 33
7.4 Divide & Conquer v.s. Backtracking........................................................................... 35
7.5 Disadvantages of Depth-First Search/Backtracking ............................................... 41
7.6 Parallel Implementation............................................................................................... 41
7.7 Results......................................................................................................................... 43

8. CONCLUSION................................................................................................................. 45

Bibliography.................................................................................................................... 46
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Table 1: Results of EP Benchmark for Class = W</td>
<td>18</td>
</tr>
<tr>
<td>2.</td>
<td>Table 2: Results of EP Benchmark for Class = A</td>
<td>19</td>
</tr>
<tr>
<td>3.</td>
<td>Table 3: Results of CG Benchmark for Class = W</td>
<td>25</td>
</tr>
<tr>
<td>4.</td>
<td>Table 4: Results of CG Benchmark for Class = A</td>
<td>25</td>
</tr>
<tr>
<td>5.</td>
<td>Table 5: Results of IS Benchmark for Class = W</td>
<td>30</td>
</tr>
<tr>
<td>6.</td>
<td>Table 6: Results of IS Benchmark for Class = A</td>
<td>30</td>
</tr>
<tr>
<td>7.</td>
<td>Table 7: Greedy algorithm Example for 66-Cents</td>
<td>34</td>
</tr>
<tr>
<td>8.</td>
<td>Table 8: Results of Coin-Changing Problem for 654</td>
<td>43</td>
</tr>
<tr>
<td>9.</td>
<td>Table 9: Results of Coin-Changing Problem for 932</td>
<td>43</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Figure 1: An Overview of the Software Architecture of PFun...</td>
<td>11</td>
</tr>
<tr>
<td>2.</td>
<td>Figure 2: An Overview of Different Components of PFun at Runtime...</td>
<td>12</td>
</tr>
<tr>
<td>3.</td>
<td>Figure 3: Comparison of PFun and OpenMP for EP Benchmark for Class W...</td>
<td>19</td>
</tr>
<tr>
<td>4.</td>
<td>Figure 4: Comparison of PFun and OpenMP for EP Benchmark for Class A...</td>
<td>19</td>
</tr>
<tr>
<td>5.</td>
<td>Figure 5: Comparison of PFun and OpenMP for CG Benchmark for Class W...</td>
<td>26</td>
</tr>
<tr>
<td>6.</td>
<td>Figure 6: Comparison of PFun and OpenMP for CG Benchmark for Class A...</td>
<td>26</td>
</tr>
<tr>
<td>7.</td>
<td>Figure 7: Comparison of PFun and OpenMP for IS Benchmark for Class W...</td>
<td>31</td>
</tr>
<tr>
<td>8.</td>
<td>Figure 8: Comparison of PFun and OpenMP for IS Benchmark for Class A...</td>
<td>31</td>
</tr>
<tr>
<td>9.</td>
<td>Figure 9: Occurrences of the 16-Cent Sub-Problem</td>
<td>36</td>
</tr>
<tr>
<td>10.</td>
<td>Figure 10: Pruned State-Space Tree for 66-Cents.</td>
<td>38</td>
</tr>
<tr>
<td>11.</td>
<td>Figure 11: Best-First Search with Bound Pruning</td>
<td>40</td>
</tr>
<tr>
<td>12.</td>
<td>Figure 12: Coin-Changing Problem for 654</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1
BACKGROUND AND INTRODUCTION

1.1 A Need to Validate Performance of Parallelized Applications

Recent trends in hardware and software have brought new challenges to the parallel programming community. The ubiquitous nature of multi-core and many-core architectures has ushered in a new era of high performance computing wherein the computing machinery is essentially hybrid in nature. The increasing presence of hybrid computing environments has precipitated the need to address the long-standing issue of having a uniform and efficient mechanism of delivering parallelism across shared-and-distributed-memory machines. Of the plethora of programming models that are available, task parallelism is the most promising when it comes to meeting the new challenges in parallel programming. The task parallel programming model is sufficiently high-level and generic, thereby enabling it to parallelize regular and irregular applications with equal aplomb.

Task parallelism offers performance comparable to that of OpenMP, although it is considered as a much higher level concept than OpenMP (i.e., sans abstraction penalties). To validate this claim, it is essential to compare the performance of applications parallelized using OpenMP with the corresponding task parallel version. In this thesis, we aim to characterize the performance of NAS parallel benchmarks parallelized using the task parallelism paradigm with its OpenMP counterpart on shared memory platforms.
1.2 Motivation

The reason for choosing NAS parallel benchmarks is twofold. First, they are an accepted industry standard benchmark suite. Second, as there are standard implementations of NAS parallel benchmarks available for OpenMP, these benchmarks lend themselves as a fair comparison with the task parallel version.

The software and hardware architecture that is proposed to validate the parallel performance rely upon a simple yet sufficient architecture using any multi core parallel processor on a Linux (Ubuntu 8.10) platform. NAS benchmarks will be parallelized using PFunc as the parallel tool. PFunc, short for Parallel Functions, is a lightweight and portable library that provides C and C++ APIs to express task parallelism on shared memory architectures. Some of the highlights of PFunc include:

- Custom Priorities for tasks
- Custom Scheduling for tasks
- Task Groups for collective communication
- Nested Parallelism
- Generic
- Open Source

1.3 Roadblocks

The NAS Parallel benchmarks (NPB) lay down specifications for a set of computational kernels. The implementation of these kernels is completely in the hands of developers. NPB implementations have traditionally been in FORTRAN; however, there
is an OpenMP version of NPB 2.3 implemented in C. Moreover, the parallelization of the serial versions of the NPB benchmarks has been carried out in one of the two models: SPMD and data parallel. Some of the reasons that make implementation of a task parallel version of NPB challenging are:

1. Language Barrier: As mentioned before, most of the open-source NPB implementations are written in FORTRAN. Furthermore, these implementations have been written not for readability, but for performance; therefore, it makes understanding the code, a difficult proposition. Our task parallel implementation makes use of PFunc, which requires that the base language be C++ for maximum performance. This is because the C interface to PFunc is given merely as a convenience and each C-based call requires pointer dereferencing - causing additional overhead. Consequently, developing the C++ versions of NPB requires additional effort.

2. Programming Model Barrier: All of the open-source implementations of NPB use the SPMD model or the data parallel model for expressing parallelism. In our project, we plan to use task parallelism. Changing the model of parallelism used requires a complete rethink of the organization of the benchmark. Task parallelism delivers good performance when the programmers express their application as a divide-and-conquer mechanism; this is completely different from either the SPMD or the data parallel models.
1.4 Project Organization

Before starting out, this section provides a brief roadmap of the Project Report. In the following pages of the report, we will look at an overview of task parallelism and NAS parallel benchmarks in Chapter 2. Chapter 3 provides a brief introduction to the PFunc tool. Chapters 4, 5 and 6, documents the PFunc implementation of the three NAS parallel benchmarks (EP, CG and IS). Each of these chapters also provides a comparison of the implementation in PFunc versus the results obtained using OpenMP. Chapter 7 describes implementation of the supplement backtracking problem (coin-changing) using PFunc. We finally summarize the report, based on the test results.
2.1 Task Parallelism

Task parallelism is a process of breaking up a problem into smaller independent tasks and executing these tasks in parallel. For example, you are listening to music on itunes and running a word-processing program at the same time on a single-core processor. Even though these applications run on separate threads, since it is a single-core processor, they still share the same processor. On the other hand if you are running the same two programs on a dual-core machine, these two applications can run independent of one another. Although they share some resources, such as memory, disk space and so on, dual-core machines can handle the two parallel tasks more efficiently.

In parallel computing environment we describe a problem as a collection of tasks. Executing each of these tasks individually in parallel can solve a problem efficiently. In order to break up a problem into tasks and execute them concurrently, we need to identify the dependencies with the data. This assumption is valid for systems with a shared memory space, where parallel executions of tasks have small working sets and infrequent access to shared global data structures [6]. Data Parallel Algorithm Strategy pattern is used when the distribution of data is crucial for execution efficiency. When the data distribution becomes essential for the efficient execution, each task is stated in terms of single stream of tasks and applied to each independent element of a data structure, thus the solution involves efficient execution of tasks with data distribution.
There are many ways to define a task. Tasks can be independent of one another or they may share data. Sometimes a task is known at the beginning of execution or can be generated during the computation. For example a task can be defined in many ways for Monte Carlo computations, as it relies on random or pseudo-random number generator algorithms. In dynamic programming, a task can be independent or can pool resources by sharing the data. In computations such as backtracking branch and bound serves as a good example where tasks can be known in the beginning stage of the execution or during the computation [6].

We also need to consider the overheads involved with task parallelism. Generating greater number of smaller tasks gives us the opportunity for load balancing but greater number of small tasks brings upon management overheads, known as “Task granularity”. A fewer number of larger tasks may reduce the management overhead but also the opportunities for load balancing becomes an overhead; known as “Task interaction”. Computations such as dynamic programming and backtrack branch and bound experience these overheads because the solution time grows exponentially with the number of decision nodes and the storage requirements quickly become excessive. These overheads can be overcome by incorporating a solution that can be arrived at in three steps [6]:

1. Understanding application concurrency characteristics.
2. Understanding implementation platform characteristics.
3. Using a flexible Parallel Algorithm Strategy Pattern to map structural and computation patterns onto implementation strategy patterns.
One may itemize the steps 1 and 2 depending on the situation. For example, first pick a platform and check whether the application can be efficiently mapped to the platform. Also one can also choose to render all levels of parallelism first and then pick a platform to construct the application.

2.2 NAS Parallel Benchmarks

To support ‘the role of United States as a leader in aeronautical technology’, the Numerical Aerodynamics Simulation (NAS) started a program in 1984. The main aim was to provide aerospace research and development community, a high-performance system that is capable of simulating an entire aerospace vehicle. This requires a development of computer system that performs complex scientific computations at a greater rate than current generation supercomputers [7]. To achieve this high-level of performance, the architecture of computer systems will probably be different to that of a shared memory multiprocessing supercomputers of today [8].

There are many reasons for which the performance evaluation of these systems for scientific computations is very difficult. Firstly, limited information is available regarding the performance of algorithms related to computational aerophysics. Secondly, the kernel benchmarks (LINPACK) used by the traditional vector supercomputers are inappropriate for the performance evaluation of highly parallel machines. Taking this into consideration, scientists from NAS Applied Research Branch and NASA Ames Research Center came up with a benchmarking approach for highly parallel super computers [8]:
1. In advanced parallel systems, algorithms and software approaches frequently change and the new approach will be different from the conventional approach.

2. Benchmarks should not be any architecture-specific (e.g., message passing code) rather it must be generic to all parallel architecture.

3. In order to verify the performance and correctness of the results, it is important to keep a generic small set of both input and output data sets. Also, the nature of the computation and the expected results must be specified in enormous detail.

4. The benchmarks must be able to accommodate new systems with memory size and run time requirements with increased power.

5. The benchmark must be readily distributable [8].

With this benchmarking approach, NAS Ames Research Center came up with eight benchmarks and the main idea of these benchmarks is to specify the problems algorithmically. NAS parallel benchmark provides implementers with the flexibility to solve various problems in ways appropriate to their specific system. Also, it provides implementers the freedom of choosing their own data structure, processor allocation, algorithm, and memory usage.

2.3 NAS Parallel Benchmark Components

NAS benchmarks consist of two major components:

- Parallel kernel benchmarks
  - Embarrassingly Parallel (EP) - Gaussian random variates.
  - MultiGrid (MG) - 3-dimensional discrete Poisson equation.
- Conjugate Gradient (CG) - Iterative solver for linear systems.
- Fast Fourier Transforms (FFT) - 3-D partial differential equation.
- Integer Sort (IS)- Bucket sort

- Simulated application benchmarks.
  - LU Solver (LU) Lower-upped symmetric Gauss-Seidel.
  - Pentadiagonal solver (SP) System of nonlinear equations
  - Block tridiagonal solver (BT) System of nonlinear equations

To illustrate the task parallelism, NAS benchmarks will be parallelized using PFunc. In our project, we implement the following three kernel benchmarks:

1. Embarrassingly Parallel Benchmark
2. Conjugate Gradient Benchmark
3. Integer Sort Benchmark
Chapter 3

OVERVIEW OF PFUNC

3.1 Introduction

In task parallelism, the existing solutions execute tasks asynchronously, as all the parallel tools provide `spawn()` and `sync()` functions, which facilitates asynchronous parallel execution. It is also important for the users to get little details such as task scheduling policy and task priority rights for efficient parallelization of the applications. It is unfortunate that these little details are concealed from users and in turn users tend to explore task parallel tools that satisfy their application needs, restricting the advantages over a set of applications. To adopt task parallelism and not to limit the power of parallelization for a set of applications, it is necessary for a tool to be configurable and extensible without any runtime penalties [9]. Recently, PFunc [4], a novel library for expressing shared-memory task parallelism in C and C++ has been released to the open-source community. PFunc makes it possible to develop efficient and reusable software libraries. PFunc has advantages over the existing task parallelism implementations (Cilk, Threading Building, Block and OpenMP) because it has all the commonalities and includes additional features such as customizing task scheduling policy, task stealing policy, task priorities, and task affinities. It also has the feature of notifying completion of multiple tasks and provides production-grade exception handling and performance monitoring mechanisms to help software developers [9].
3.2 Software Architecture

The goal of PFunc is to allow users to execute tasks on shared memory. In order to achieve this goal there should be an interaction with various low-level components and user applications. PFunc has this capability to allow users to interact with low-level components and hence are allowed to execute tasks on parallel processors [9]. Figure 1 gives an overview of the software architecture of PFunc. Setting task affinities is an important attribute where in this capability is currently provided by the operating systems. PFunc allows users to map individual threads to specific processors which in turn make PFunc interface with the operating system [9].

![Figure 1: An Overview of the Software Architecture of PFunc](image)

In PFunc many concurrent data structures and algorithms are implemented using custom synchronization primitives, which gives maximum efficiency because they are
built on top of the processor-specific atomic operations. Also PFunc has the feature of capturing hardware statistics of users applications execution through “performance profiler” which interfaces with Performance Application Programming Interface (PAPI) [9].

3.3 Components of PFunc

PFunc’s components can be categorized into three groups: Figure 2 shows an overview of different components of PFunc at runtime.

![Figure 2: An Overview of Different Components of PFunc at Runtime](image)

3.3.1 User-Provided Components

In Figure 2, the User-provided component consists of: work, task queue, and task predicates. The logical grouping of task queue and task predicate components is called
the task scheduler. This is responsible for selecting the next task that is to be executed by each thread. Task queue and task predicate components can be chosen by the users based on the scheduling policies (cilkS, prioS, lifoS and fifoS) during compile time. If users implements their own scheduling policy then task queue and task predicate components must be defined. [9].

3.3.2 Generated Components

In Figure 2, the Generated component: attribute, task, and task manager, are automatically created during compile-time, using the information provided by user-provided components. Each task is associated with an attribute; users can control the execution of the task by specifying the sub-attributes (priority, queue_number, num_waiters, grouped, level, nested) of the task’s attribute at the time of spawning [9].

3.3.3 Fixed Components

In Figure 2, group, thread manager, exception handler, and performance profiler constitutes fixed components and are non-configurable. [9].

PFunc components are implemented as C++ classes; hence one or more objects of these classes will be interacting with each other at runtime. A portable task parallel library is a complex piece of software and to build a customizable and extensible task parallel library, it is important to organize groups of related functions and data into components [9]. In the coming chapters, to illustrate task parallelism, three NAS parallel benchmarks (EP, IS and CG) will be implemented using PFunc.
4.1 Introduction

Embarrassingly Parallel (EP) is one of the NAS parallel kernel benchmarks, which evaluates an integral by means of pseudorandom tests. This problem is typical of many Monte Carlo applications; since it requires almost no communication. This benchmark provides estimates of the higher achievable limits for floating point performance.

4.2 Problem Definition


“Set \( n = 2^{28} \) and \( s = 271828183 \). Generate the pseudorandom floating-point values \( r_j \) in the interval \((0, 1)\) for \( 1 \leq j \leq 2n \) using the scheme described below. Then for \( 1 \leq j \leq n \) set \( x_j = 2r_{2j-1}^2 \) and \( y_j = 2r_{2j-1}^2 \). Thus \( x_j \) and \( y_j \) are uniformly distributed on the interval \((-1, 1)\).

Next set \( k = 0 \), and beginning with \( j = 1 \), test to see if \( t_j = x_j^2 + y_j^2 \leq 1 \). If not, reject this pair and proceed to the next \( j \). If this inequality holds, then set \( k \leftarrow k + 1 \); \( X_k = x_j \sqrt{(-2 \log t_j) / t_j} \) and \( Y_k = y_j \sqrt{(-2 \log t_j) / t_j} \), where \( \log \) denotes the natural logarithm. Then \( X_k \) and \( Y_k \) are independent Gaussian deviates with mean zero and variance one. Approximately \( n\pi / 4 \) pairs will be constructed in this manner.
Finally, for $0 \leq l \leq 9$ tabulate $Q_l$ as the count of the pairs $(X_k, Y_k)$ that lie in the square annulus $l \leq \max(|X_k|, |Y_k|) < l + 1$, and output the ten $Q_l$ counts. Each of the ten $Q_l$ counts must agree exactly with reference values.

The $2n$ uniform pseudorandom numbers $r_j$ mentioned above are to be generated according to the following scheme: Set $a = 5^{13}$ and let $x_0 = s$ be the specified initial “seed”. Generate the integers $x_k$ for $1 \leq k \leq 2n$ using the linear congruential recursion $x_{k+1} = ax_k (\mod 2^{46})$ and return the numbers $r_k = 2^{-46}x_k$ as the results. Observe that $0 < r_k < 1$ and the $r_k$ are very nearly uniformly distributed on the unit interval” [10].

4.3 Generating a Random Number

Numbers that are chosen randomly similar to the probable outcome from a roll of dice are termed as random numbers. Numbers that are chosen at random are useful in many different kinds of applications such as simulation, sampling, Numerical analysis, Decision-making and so forth. There are two ways that computers can generate random numbers:

1. Creating a device that monitors a completely random natural event and sending the results to the computer. This approach is very difficult for common people because creating a device and connecting to the machine is not a good idea.

2. Create a formula that generates a pseudo-random number. A good pseudorandom generator has the characteristics of not to repeat any numbers in a sequence, a good numeric distribution of numbers and lack of predictability.
To generate random numbers given in this benchmark, we have used linear congruential (LCG) random number generator. It's quite common to use a LCG algorithm and the numbers generated as a result of this algorithm returns a list of sequential numbers within a specified range. Because of the modulus arithmetic involved this will generate every number in the range once before repeating. The algorithm not only computes the desired answer to a problem, it is also intended to combine well with the internal operations of a digital computer.

4.3.1 Linear Congruential Random Number Generator

A linear congruential random number generator is a method for algorithmically generating a pseudo-random sequence of numbers. The method of random number generation by linear congruential method works by computing each successive random number from the previous, starting with a seed.

The linear congruential method uses four magic numbers [11]:

\[ m, \text{ the modulus; } m > 0. \]

\[ a, \text{ the multiplier; } 0 \leq a < m. \]

\[ c, \text{ the increment; } 0 \leq c < m. \]

\[ X_0, \text{ the starting value; } 0 \leq X_0 < m. \]

The desired sequence of random numbers \( (X_n) \) is then obtained by setting

\[ X_{n+1} = (aX_n + c) \mod m, \ n > 0. \]
This is called a linear congruential sequence. The sequence generated is not always random for all choices of \( m, a, c \) and \( X_0 \), the principles of choosing magic numbers should be appropriate.

The conditions for a maximum period “\( m \)” in a linear congruential sequence are given by Knuth in Volume 2 "Seminumerical Algorithms", "The Art of Computer Programming" series. These conditions are presented as “Theorem A” of section 3.2.1.2 on page 16.

4.3.2 Choice of Multiplier

We have to choose the multiplier “\( a \)” so as to give the period of maximum length “\( m \)”. A long period is essential for any sequence that is to be used as a source of random numbers. For example when \( a = c = 1 \), the sequence is simply \( X_{n+1} = (X_n + 1) \mod m \), with the period length \( m \). Since only \( m \) different values are possible, the period cannot be longer than \( m \). To achieve the maximum length \( m \) we need to investigate all the possible choices of “\( a \)”, “\( c \)” and \( X_0 \). All the values of the parameters can be characterized in a simple way, when \( m \) is the product of distinct primes, only \( a = 1 \) will produce the full period, but when \( m \) is divisible by a high power of some prime there is considerable latitude in the choice of “\( a \)”.

4.3.3 Theorem A

A linear congruential sequence defined by \( m, a, c \) and \( X_0 \) has period length \( m \) if and only if [11]:

1. \( c \) is relatively prime to \( m \).
2. \( b = a-1 \) is a multiple of \( p \), for every prime \( p \) dividing \( m \).

3. \( b \) is a multiple of 4, if \( m \) is a multiple of 4.

### 4.4 Parallel Implementation

As the benchmark name suggests, parallelizing this particular benchmark is simple. The iteration range outer loop that generates the Gaussian pairs is known beforehand (e.g., from 0 to 100). When executing in parallel, the outer loop iteration space is broken down into the number of threads and each thread calculates a pre-specified number of Gaussian pairs independent of one another. At the end, all the values are summed up sequentially. As there are no dependencies between tasks, we expect a near linear speedup.

### 4.5 Results

The experiments were conducted on a four-socket, quad-core (16 cores total) AMD 8356 processor running Linux kernel 2.6.24. For compilation, we used GCC 4.3.2 with:”-O3-omit-frame-pointer-funroll-loops″. The same compiler was used for the OpenMP implementation as well (using the extra flag “-fopenmp” to compile). To collect hardware metrics, we used PF unc's integration with PAPI.

<table>
<thead>
<tr>
<th>No_rnds</th>
<th>N</th>
<th>No_Gaussian_Pairs</th>
<th>Threads</th>
<th>OMP_CPU_time</th>
<th>OMP_Mops</th>
<th>PF unc_CPU_time</th>
<th>PF unc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>67108864</td>
<td>2^25</td>
<td>26354769</td>
<td>1</td>
<td>8.0355</td>
<td>8.35</td>
<td>8.0778</td>
<td>8.31</td>
</tr>
<tr>
<td>67108864</td>
<td>2^25</td>
<td>26354769</td>
<td>2</td>
<td>4.0627</td>
<td>16.52</td>
<td>4.0902</td>
<td>16.41</td>
</tr>
<tr>
<td>67108864</td>
<td>2^25</td>
<td>26354769</td>
<td>4</td>
<td>2.0493</td>
<td>32.75</td>
<td>2.077</td>
<td>32.31</td>
</tr>
<tr>
<td>67108864</td>
<td>2^25</td>
<td>26354769</td>
<td>8</td>
<td>1.0411</td>
<td>64.46</td>
<td>1.0868</td>
<td>61.75</td>
</tr>
<tr>
<td>67108864</td>
<td>2^25</td>
<td>26354769</td>
<td>16</td>
<td>0.535</td>
<td>125.43</td>
<td>0.812</td>
<td>92.64</td>
</tr>
</tbody>
</table>

Table 1: Results of EP Benchmark for Class = W
Table 2: Results of EP Benchmark for Class = A

<table>
<thead>
<tr>
<th>No_rnds</th>
<th>N</th>
<th>No_Gaussian_Pairs</th>
<th>Threads</th>
<th>OMP_CP U_time</th>
<th>OMP_Mops</th>
<th>PFunc_CP U_time</th>
<th>PFunc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>1</td>
<td>64.0531</td>
<td>8.38</td>
<td>64.3944</td>
<td>8.34</td>
</tr>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>2</td>
<td>32.0549</td>
<td>16.75</td>
<td>32.454</td>
<td>16.54</td>
</tr>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>4</td>
<td>16.1909</td>
<td>33.16</td>
<td>16.2534</td>
<td>33.03</td>
</tr>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>8</td>
<td>10.8128</td>
<td>49.65</td>
<td>10.9154</td>
<td>49.23</td>
</tr>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>16</td>
<td>8.1016</td>
<td>66.27</td>
<td>8.2996</td>
<td>64.69</td>
</tr>
<tr>
<td>536870912</td>
<td>2^28</td>
<td>210832767</td>
<td>1</td>
<td>4.0701</td>
<td>131.91</td>
<td>4.1018</td>
<td>133.63</td>
</tr>
</tbody>
</table>

Figure 3: Comparison of PFunc and OpenMP for EP Benchmark for Class W

Figure 4: Comparison of PFunc and OpenMP for EP Benchmark for Class A
The Tables 1 and 2, provides the results of an execution of the EP benchmark for class W and class A, using OpenMP and PFunc. We can see that there is hardly any difference between the execution times for OpenMP and PFunc. This can be verified, by mapping the results on to a graph, with “Number of Threads” on the X-axis and “Execution Time” on the Y-axis as shown in Figures 3 and 4 (respectively). This verifies the theory that task parallelism indeed subsumes data parallelism.
Chapter 5

IMPLEMENTATION OF CONJUGATE GRADIENT BENCHMARK

5.1 Introduction

In a system of linear equations whose matrix is symmetric and positive definite, then conjugate gradient method algorithm is used to obtain a numerical solution. Since conjugate gradient method is iterative, it can be applied to sparse matrix systems that are too large to handle. There are many direct methods to compute eigenvalues and eigenvectors in a sparse matrix, but the direct methods become inefficient when we solve for all the eigenvalues and eigenvectors when only one of them is desired. Conjugate gradient method is much faster and used for computing large or small eigenvalues. The proposed benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros [5]. This kernel benchmark tests irregular long distance communication by employing unstructured matrix communication.

5.2 Problem Definition

The problem definition is given in "The NAS Parallel Benchmarks," Technical Report RNR-94-007 NASA Ames Research Center March 1994. “The CG benchmark computes an iterative approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. Within the iteration loop, the core procedure is to solve a linear system of equations via the conjugate gradient method (CGM). This
The kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix-vector multiplication.

The inner iteration of the CGM computes the product of a sparse matrix with a vector:

\[ Y = Ax \]

and then uses the result \( y \) to update the \( x \) vector. Unless \( P \), the number of processors, is very large, the cost of the update is insignificant. The size \( N \) of this problem is the length of \( x \). Another parameter, \( k \), measures the sparsity of the matrix; each row and column of \( A \) has about \( k \) nonzero elements.” [10]

The inverse power method is to be implemented as follows [12]:

\[ x = [1, 1, \ldots 1]^T; \]

(Start timing here)

DO \( it = 1, niter \)

\[ \text{Solve the system } Az = x \text{ and return } ||r||, \text{ as described below} \]

\[ \zeta = \lambda + 1/(x^T z) \]

Print \( it, ||r||, \) and \( \zeta \)

\[ x = z / ||z|| \]

ENDDO

(Stop timing here)

“A” denotes sparse matrix of order \( n \), roman letters are vectors, \( x_j \) is the \( j^{th} \) component of the vector \( x \) and superscript “T” indicates the usual transpose operator. Lower case Greek letters are scalars. \( ||x|| \) is the Euclidean norm of a vector \( x \), \( ||x|| = \sqrt{x^T x} \)
5.3 Inverse Power Method

The inverse power method is used to find largest eigenvalue and this method is similar to power method which is used to find smallest eigenvalue. Consider a system $Ax = \lambda x$ for which we want to find the eigenvalues. Power method provides a simple iterative procedure to find $v_1$ and $\lambda_1$ for a given matrix $A$, where $\lambda_1$ is the largest eigenvalue and $v_1$ is the corresponding eigenvector for $\lambda_1$. The assumption is

$$|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$$

The power method is a simple iterative procedure. It starts with an initial guess $x_0$ and generated sequence of approximations $x_0$ to generate sequence of approximations $x_k$ which hopefully converges as $k \to \infty$

5.3.1 Power Method Algorithm

The algorithm for power methods is simple [13].

- Choose initial guess $x_0$.

- DO $k = 1$ to $m$
  - Set $y_k = Ax_{k-1}$
  - Set $\alpha_k = \text{the largest element of } y_k \text{ (in absolute value)}$
  - Set $x_k = y_k / \alpha_k$

- End DO

The iterative step is performed $m$ times and in each step we pre-multiply the previous approximation $x_{k-1}$ by $A$ and the next approximations are obtained by dividing...
the resulting vector $y_k$. Thus for each $k > 0$, $x_k$ has the property that its largest element is equal to one.

### 5.3.2 Inverse Power Method Algorithm

The inverse power method deals with $A^{-1}$ rather than $A$. By doing slight modification in the power method algorithm we can find largest eigenvalue. Thus the algorithm would be [13]

- Choose initial guess $x_0$.
- DO $k = 1$ to $m$
  - Set $y_k = A^{-1}x_{k-1}$ or $Ay_k = x_{k-1}$
  - Set $\alpha_k$ = the largest element of $y_k$ (in absolute value)
  - Set $x_k = y_k / \alpha_k$
- End DO

Since the eigenvalues of $A^{-1}$ are $1/\lambda_n$, ..., $1/\lambda_1$ the inverse power method should converge to $1/\lambda_n$ which is the largest eigenvalue of $A^{-1}$

### 5.4 Parallel Implementation

In the CG benchmark, there are three operations that can be executed in parallel. First, the parallel dot product ($a = V1.V2; a =$ scalar, $V1$, $V2$=vectors). Second, the sparse matrix-vector multiply ($V2=A.V1; A =$ sparse matrix, $V1$, $V2$=vectors). Third, the SaXpbY operation ($S = a*X + b*Y; S$, $X$, $Y =$ vectors; $a$, $b =$ scalars). The parallelization
for these three kernel operations is straightforward --- they each have an outer-loop
whose iteration space is divided up amongst tasks and executed in parallel. For each
iteration of the Eigen value algorithm, there are 2 parallel dot products and 2 parallel
SaXpbY operations. In addition, there is a call to the Conjugate Gradient method, which
in turn is iterative and has 1 sparse matric-vector operation, 2 parallel dot products, and 3
parallel SaXpbY operations. Therefore, we expect good parallel efficiency.

5.5 Results

The experiments were conducted in the same environment as used for EP
benchmark. That is, quad-core (16 cores total) AMD 8356 processor running Linux
kernel 2.6.24 and compiled using GCC 4.3.2 with: ”-O3 -fomit-frame-pointer -funroll-
loops”.

<table>
<thead>
<tr>
<th>Size</th>
<th>Threads</th>
<th>OMP_Time</th>
<th>OMP_Mops</th>
<th>PFunc_Time</th>
<th>PFunc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>1</td>
<td>1.44</td>
<td>291.59</td>
<td>1.32</td>
<td>318.26</td>
</tr>
<tr>
<td>7000</td>
<td>2</td>
<td>0.69</td>
<td>606.69</td>
<td>0.19</td>
<td>530.17</td>
</tr>
<tr>
<td>7000</td>
<td>4</td>
<td>0.19</td>
<td>2269.75</td>
<td>0.61</td>
<td>694.94</td>
</tr>
<tr>
<td>7000</td>
<td>8</td>
<td>0.11</td>
<td>3942.62</td>
<td>0.55</td>
<td>768.95</td>
</tr>
<tr>
<td>7000</td>
<td>16</td>
<td>0.09</td>
<td>4665.73</td>
<td>0.4</td>
<td>1013.1</td>
</tr>
</tbody>
</table>

Table 3: Results of CG Benchmark for Class = W

<table>
<thead>
<tr>
<th>Size</th>
<th>Threads</th>
<th>OMP_Time</th>
<th>OMP_Mops</th>
<th>PFunc_Time</th>
<th>PFunc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>14000</td>
<td>1</td>
<td>5.27</td>
<td>284.02</td>
<td>4.84</td>
<td>309.41</td>
</tr>
<tr>
<td>14000</td>
<td>2</td>
<td>2.74</td>
<td>545.41</td>
<td>2.79</td>
<td>535.91</td>
</tr>
<tr>
<td>14000</td>
<td>4</td>
<td>1.8</td>
<td>831.08</td>
<td>2.07</td>
<td>721.56</td>
</tr>
<tr>
<td>14000</td>
<td>8</td>
<td>1.68</td>
<td>889.37</td>
<td>2.18</td>
<td>685.6</td>
</tr>
<tr>
<td>14000</td>
<td>16</td>
<td>1.69</td>
<td>884.17</td>
<td>2.18</td>
<td>686.13</td>
</tr>
</tbody>
</table>

Table 4: Results of CG Benchmark for Class = A
The Tables 3 and 4, provide the results of an execution of the CG benchmark for class W and class A respectively, using OpenMP and PFunc. This can be verified, by mapping the results on to a graph, with “Number of Threads” on the X-axis and “Execution Time” on the Y-axis as shown in Figures 5 and 6 respectively. The performance of OpenMP and PFunc are nearly identical leading credence to the theory that task parallelism indeed subsumes data parallelism.
Chapter 6
IMPLEMENTATION OF INTEGER SORT BENCHMARK

6.1 Introduction

As English dictionary describes; sorting is categorizing of things having some common features or type, it is common for computer programmers to use this word in a special sense of sorting things in ascending or descending order. Sorting is important in many applications like solving the togetherness problem, sequential accessing on large files, finding all the matches in a sequential file and, so forth. For wider perspective we also find that sorting algorithms make an interesting research of how to solve computer-programming problems in the real world.

Fast integer sorting is crucial for solving problems in many domains, and as such, is used in several benchmarks such as NAS and SPLASH [14]. Integer sorting is a subclass of the sorting problem where the element is polynomially bounded in the number of elements to be sorted. Integer Sort (IS) is one of the NAS parallel kernel benchmarks, which performs sorting operations that is important in “particle method” codes. This benchmark is to test long distance communication performance.

6.2 Problem Definition

Sort N keys in parallel. The keys are generated by the sequential key generation algorithm given below and initially must be uniformly distributed in memory [12].

A sequence of keys, \( \{K_i \mid i = 0,1, \ldots, N-1\} \), will be said to be sorted if it is arranged in non-descending order, that is, \( K_i \leq K_{i+1} \leq K_{i+2} \ldots \). The rank of a particular
key in a sequence is the index value \( i \) that the key would have if the sequence of keys were sorted. Ranking then is the process of arriving at a rank for all the keys in a sequence. Sorting is the process of permuting the keys in a sequence to produce a sorted sequence. If an initially unsorted sequence, \( K_0, K_1, \ldots K_{N-1} \) ranks \( r(0), r(1), \ldots, r(N-1) \), the sequence becomes sorted when it is rearranged in the order \( K_{r(0)}, K_{r(1)}, \ldots K_{r(N-1)} \). Sorting is said to be stable if equal keys retain their original relative order. In other words, a sort is stable only if \( r(i) < r(j) \) whenever \( K_{r(i)} \) and \( i < j \) [12].

6.3 Sequential Key Generation Algorithm

The sequential key generation algorithm is given by D.E Knuth in Volume 3 "Sorting and searching", "The Art of Computer Programming" series. These conditions are presented as "Algorithm Q" of section 5.2.2 on page 126.

Records \( R_1, R_2, \ldots R_N \) are arranged in place; after sorting is complete, their keys will be in order, \( K_1 \leq K_2 \leq \ldots \leq K_N \). Each keys is assumed to be an \( m \)-bit binary number, e.g. \( (a_1, a_2 \ldots a_m) \) the \( i^{th} \) most significant bit, \( a_i \) is called "bit i" of the key. An auxiliary stack with room for at most \( m-1 \) entries is needed for temporary storage. This algorithm essentially follows the radix exchange partitioning procedure [12].

1. \([\text{Initialize}]\) Set the stack empty and set \( l \leftarrow 1, r \leftarrow N, b \leftarrow 1\)
2. \([\text{Begin new stage}]\) We now wish to sort the subfile \( R_l \leq \ldots \leq R_r \) on bit \( b \); from the nature of the algorithm, we have \( l \leq r \). If \( l = r \), go to step 10 (since a one word file is already sorted) Otherwise set \( i \leftarrow l, j \leftarrow r\)
3. \([\text{Inspect } K_i \text{ for } 1]\) Examine bit \( b \) of \( K_i \). If it is a 1, go to step 6.
4. [Increase i] Increase i by 1. If \( i \leq j \), return to step 3; otherwise go to step 8.

5. [Increase \( K_{j+1} \) for 0] Examine bit b of \( K_{j+1} \). If it is 0 go to step 7.

6. [Decrease j] Decrease j by 1. If \( i \leq j \) go to step 5; otherwise go to step 8.

7. [Exchange \( R_i, R_{j+1} \)] Interchange records \( R_i \leftrightarrow R_{j+1} \); then go to step 4.

8. [Test special cases] (At this point partitioning stage has been completed; \( i = j+1 \), bit b of keys \( K_i,...K_j \) is 0, and bit b of keys \( K_i,...K_r \) is 1) Increase b by 1. If \( b > m \), where m is the total number of bits in the keys, go to step 10. (In such a case, the subfile \( R_1,...R_r \) has been sorted. This test need not be made if there is no chance of having equal keys present in the file.) Otherwise if \( j < l \) or \( j = r \), go back to step 2 (all bits examined were 1 or 0 respectively.) Otherwise if \( j = l \), increase \( l \) by 1 and go to step 2 (there was only one 0 bit).

9. [Put on stack] Insert the entry \( (r, b) \) on top of the stack; then set \( r \leftarrow j \) and go to step 2.

10. [Take off stack] If the stack is empty we are done sorting; otherwise set \( l \leftarrow r+1 \), remove the top entry \( (r_r, b_r) \) of the stack, set \( r \leftarrow r_r \), \( b \leftarrow r_r \), and return to step 2.

6.4 Parallel Implementation

Like the other NAS parallel benchmarks, the IS benchmark is also data parallel, a paradigm that can be easily simulated using task parallelism. Like the other cases, we divide the outer iteration space into N chunks, where N is the number of threads. Therefore, we are creating 1 task per thread in the system. When all the tasks finish
computing the frequencies of their partition of the data, the frequencies are added
\textit{(reduced)} using the “+” reduction operator

6.5 Results

The IS Benchmarks were executed on the same environments as used for the previous benchmarks. Here again, we can see that the results obtained from this execution are in acceptance with the theory that task parallelism subsumes data parallelism. The Tables 5 and 6 below, provide us with the results for execution of the IS Benchmark and depicts very close execution times for both OpenMP and PFunc implementation. The Figures 7 and 8 are a result of plotting these points on a graph with “Number of Threads” on the X-axis and “Execution time” on the Y-axis

<table>
<thead>
<tr>
<th>Size</th>
<th>Threads</th>
<th>OMP_Time</th>
<th>OMP_Mops</th>
<th>PFunc_Time</th>
<th>PFunc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1048576</td>
<td>1</td>
<td>0.12</td>
<td>84.52</td>
<td>0.1</td>
<td>88.52</td>
</tr>
<tr>
<td>1048576</td>
<td>2</td>
<td>0.07</td>
<td>144.79</td>
<td>0.07</td>
<td>144.63</td>
</tr>
<tr>
<td>1048576</td>
<td>4</td>
<td>0.04</td>
<td>286.69</td>
<td>0.04</td>
<td>287.1</td>
</tr>
<tr>
<td>1048576</td>
<td>8</td>
<td>0.03</td>
<td>322.09</td>
<td>0.03</td>
<td>322.14</td>
</tr>
<tr>
<td>1048576</td>
<td>16</td>
<td>0.04</td>
<td>264.49</td>
<td>0.05</td>
<td>259.64</td>
</tr>
</tbody>
</table>

Table 5: Results of IS Benchmark for Class = W

<table>
<thead>
<tr>
<th>Size</th>
<th>Threads</th>
<th>OMP_Time</th>
<th>OMP_Mops</th>
<th>PFunc_Time</th>
<th>PFunc_Mops</th>
</tr>
</thead>
<tbody>
<tr>
<td>8388608</td>
<td>1</td>
<td>1.14</td>
<td>73.9</td>
<td>1.16</td>
<td>71.83</td>
</tr>
<tr>
<td>8388608</td>
<td>2</td>
<td>0.59</td>
<td>141.42</td>
<td>0.47</td>
<td>179.52</td>
</tr>
<tr>
<td>8388608</td>
<td>4</td>
<td>0.38</td>
<td>221.73</td>
<td>0.47</td>
<td>180.31</td>
</tr>
<tr>
<td>8388608</td>
<td>8</td>
<td>0.38</td>
<td>223.35</td>
<td>0.54</td>
<td>154.5</td>
</tr>
<tr>
<td>8388608</td>
<td>16</td>
<td>0.54</td>
<td>153.99</td>
<td>0.56</td>
<td>150.23</td>
</tr>
</tbody>
</table>

Table 6: Results of IS Benchmark for Class = A
Figure 7: Comparison of PFunc and OpenMP for IS Benchmark for Class W

Figure 8: Comparison of PFunc and OpenMP for IS Benchmark for Class A
Chapter 7

IMPLEMENTATION OF COIN-CHANGING PROBLEM

7.1 Introduction

Backtracking is a general algorithm for finding all (or some) solutions to computational problem that incrementally builds candidates to the solutions, and abandons each partial candidate “c” ("backtracks") as soon as it determines that “c” method cannot be used to arrive at valid solution [12]. Backtracking is an exhaustive search where it can be applied to problems that use the concept of “partial candidate solution”. The solution can be explained with the concept of “survival-of-the fittest” where you distinguish between good and bad solutions. This concept can be used in backtracking for solving problems that have limited solutions in a confined amount of time.

7.2 Examples of Backtracking

Backtrack search is recognized as a basic algorithmic technique in computer science. It can be used to search for values which instantiate a set of variables subject to a set of constraints [15]. Some of the good running examples of backtracking are listed below. Problem solving technique and PFunc implementation of the coin-changing problem is explained in further sections of the chapter.
• Eight Queens Problem: The eight queens puzzle also known as N queens problem is representing the positions of the eight queens on a [8 X 8] chessboard, and the constraints require that none of the queens can attack each other. Consider an [N X N] chessboard and finding a solution when N is 3, 4 or 5, is reasonably straight forward. The problem becomes extremely hard when the value of N becomes relatively bigger.

• Job-Assignment problem: In an [N X N] cost matrix where each row contains the cost of assigning a person to each job. The problem is to find the minimum assignment of all N people to the N jobs, such that each person is assigned exactly one job and no job has two people assigned to it [16].

• Traveling Salesperson Problem: Is defined as, in a weighted, directed graph G = (V, E) finding the minimum cost (tour) of simple cycle that visits every vertex in the graph.

• Coin Changing Problem: In a given set of coins, make change with the fewest number of coins assuming that an unlimited supply of each type of coins is available [16].

7.3 Coin Changing Problem

The problem is to generate change, in terms of a given set of coins that sum up to a given amount. The aim here is to generate a result with the fewest number of coins assuming that an unlimited supply of each type of coins is available. When asked to solve
this problem, we find the obvious result sets using greedy algorithm by repeatedly giving
back the largest coin first that is less or equal to the change remaining until the desired
sum is obtained. For example we need to make change for 66-cents; optimal solution
would be five coins: two quarters, a dime, a nickel and a penny.

The above example holds good for the coin set; \{1, 5, 10, 25\} cents. Does the
greedy method apply for all given set of coins? Let us consider one more example with
the coin set; \{1, 5, 10, 12, 25\} cents, to make change for 66-cents. The solution for this
scenario is described in Table 7. We can see that, by returning the largest coin first, that is
less than or equal to the remaining change the solution is seven coins.

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Coins to be returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>Return the largest coin less $\leq$ 66 cents, that is, 25-cent coin</td>
</tr>
<tr>
<td>-25</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
</tr>
<tr>
<td>-25</td>
<td>Return the largest coin less $\leq$ 41 cents, that is, 25-cent coin</td>
</tr>
<tr>
<td>41</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td>Return the largest coin less $\leq$ 16 cents, that is, 12-cent coin</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>Return the largest coin less $\leq$ 4 cents, that is, 1-cent coin</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Return the largest coin less $\leq$ 3 cents, that is, 1-cent coin</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Return the largest coin less $\leq$ 2 cents, that is, 1-cent coin</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Return the largest coin less $\leq$ 1 cents, that is, 1-cent coin</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Greedy algorithm Example for 66-Cents
Now the question arises is this the best solution? Greedy algorithm does not give the optimal solution for a different set of coin types. As we see from the above example, making change for 66-cents with coin types of \{1, 5, 12, 10, 25\} cents results in seven coins. Clearly, a better five-coin solution exists, that is, to return 25, 25, 10, 5, 1} cent coins. What is the best way to find all the possible solutions and come up with the best one?

7.4 Divide & Conquer v.s. Backtracking

In order to optimize the coin-changing problem we can investigate other approaches like divide and conquer or dynamic programming techniques. The dependencies for using these techniques lie in the initial set of a recursive relationship that would allow solving the problem by breaking it into smaller workable results sets or problems. In this example making change for 66-cents, the problem can be broken down by returning the highest value of coin that can be used not exceeding 66 which in the current coin set \{1, 5, 10, 12, 25\}, would be \{25\} The technique would further identify the suitable options using the same approach for breaking down 41 cents (66 - 25) and return the next coin back based on the coin set.

Thus the recursive relationship FewestNumberOfCoins for the coin-changing problem would be [16]

\[
\text{FewestNumberOfCoins (change)} = \begin{cases} 
\min (\text{FewestNumberOfCoins(change - coin)}) & \text{if } \text{coin} \in \text{CoinSet} \\
1 & \text{if } \text{change} \in \text{CoinSet} 
\end{cases}
\]
The recursive divide-and-conquer algorithm is relatively simple and easy to implement. The drawback of using the divide and conquer method attributes to the number of redundant calculations it performs when it encounters the same sub-problem which makes it impractical.

Figure 9: Occurrences of the 16-Cent Sub-Problem.
For example the recursion tree for 66-cents with coins \{1, 5, 10, 12, 25\} leads to the 16-cent sub-problem many times as shown in the Figure 9. Each time the 16-cent sub-problem is encountered it performs the same set of calculations for breaking down the 16 cents

Problems encountered during divide and conquer method can be overcome using backtracking. It uses state-space/search-space tree that shows the recursive calls performed during backtracking. Since backtracking uses state-space/search-space tree, it provides an enhancement over depth-first search by minimizing the search space. In the coin-changing problem in order to improve the efficiency, we stop processing the branches of the tree that either cannot lead to a solution or cannot lead to a better solution than we have found already, that is, we want to cut down or “prune” the search tree. The criteria for pruning are [16]:

- Giving back a coin that is worth more than the amount of change.
- Giving back one less coin than the best solution and having more change to return.

The biggest problem with backtracking is that it searches the state-space tree in a depth-first search pattern, which might be slow, as shown in Figure 10. In order to improve the efficiency we need to process the tree that would only result in the successful processing and ignore all other trees that do not lead to a better solution. Keeping this objective in mind, we can now only estimate the choice that would lead to the best solution and cannot pinpoint to the best choice.
Figure 10: Pruned State-Space Tree for 66-Cents.
During the processing of the estimated best choice, if the result sets further down the tree aren’t promising, we can then backtrack to an earlier node that does. Thus the suggested approach would then be to not follow an identified order since this would result in searching less number of result trees and reach the best solution with less number of iterations. If the estimate made when evaluating a node’s potential is a bound on the best solution possible for any node derived from it, then we can use these bounds when pruning the tree. This type of algorithm is known as a best-first search with branch-and-bound pruning [16].

For the coin-change problem, we want to calculate the bound in an efficient manner. More often the bound calculation is done using a greedy algorithm and for this problem the greedy calculation is the algorithm previously described (giving back the largest possible coin). Since greedy calculation is only useful in arriving at the best-first search, we can combine this with bound calculation to prune state-space tree and thus improve the efficiency. Figure 11 illustrates coin-change problem for 66-cent using best-first search with bound pruning for coin set \{1, 5, 10, 12, 25\}

\[
\text{bound} = \left( \frac{\text{Number of coins returned to reach this}}{\text{value of largest \leq change remaining}} \right) + \left( \frac{\text{Change remaining at this node}}{\text{value of largest \leq change remaining}} \right)
\]
Figure 11: Best-First Search with Bound Pruning.
7.5 Disadvantages of Depth-First Search/Backtracking

- Backtracking uses depth first search method, so there is a probability that it may go down the left most path forever. In this case even a finite graph can become an infinite tree.
- Imposing a cutoff depth on the search beforehand would be a good solution, but the problem is cutoff depth is hardly known in advance.
- If the cutoff depth (d) is known beforehand, there may be a possibility where the algorithm fails to find a solution if the chosen depth is less than d. when the cutoff depth is greater than d, then the execution time is large and the first solution found may not be optional.
- Since backtrack is a depth first search, there is no guaranteed solution. If there is more than one solution, then there is no guarantee to find a minimal solution.

7.6 Parallel Implementation

The parallel implementation of the coin-changing algorithm is fairly straightforward. The sequential algorithm consists of a iterative loop that uses recursion to explore deeper into the tree. In other words, the sequential algorithm is a depth-first search of the solution space. To parallelize our algorithm, we simply spawn the recursive function using PFunc instead of calling it sequentially. The code example given below explains the parallelization process:
• Sequential Version

```c
void build (int k) {
    while (more choices for x[k]) {
        x[k] = the next choice;
        if ( x[1..k] satisfies the criterion function ) {
            if ( k is the last component)
                if (x is better than b) b = x;
            else if (bound(k) build(k+1))
        }
    }
}
```

• Parallel Version

```c
void build (int k) {
    while (more choices for x[k]) {
        x[k] = the next choice;
        if ( x[1..k] satisfies the criterion function ) {
            if ( k is the last component)
                if (x is better than b) b = x;
            else if (bound(k) build(k+1))

                SPAWN build(k+1)

                SYNC
        }
    }
```
7.7 Results

We ran our experiments on the same environments as used for the previous benchmarks. For compilation, we used GCC 4.3.2 with: `"\texttt{-O3 -fomit-frame-pointer -funroll-loops}"`. To collect hardware metrics, we used PFunc's integration with PAPI. The results are shown in Tables 8 and 9 for problem sizes of 654 and 932 respectively.

<table>
<thead>
<tr>
<th>Time(sec)</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.07615</td>
<td>Sequential</td>
</tr>
<tr>
<td>3.26109</td>
<td>1</td>
</tr>
<tr>
<td>4.3443</td>
<td>2</td>
</tr>
<tr>
<td>3.83123</td>
<td>4</td>
</tr>
<tr>
<td>4.09478</td>
<td>8</td>
</tr>
<tr>
<td>4.91364</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 8: Results of Coin-changing problem for 654

<table>
<thead>
<tr>
<th>Time(sec)</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.4565</td>
<td>Sequential</td>
</tr>
<tr>
<td>16.2396</td>
<td>1</td>
</tr>
<tr>
<td>18.6048</td>
<td>2</td>
</tr>
<tr>
<td>19.367</td>
<td>4</td>
</tr>
<tr>
<td>20.2617</td>
<td>8</td>
</tr>
<tr>
<td>24.0073</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 9: Results of Coin-changing problem for 932
As can be seen in Figure 12, the sequential version outperforms the parallel version --- even with 16 threads (1 thread per core). The primary reason for this is that the amount of computation in each node is insignificant when compared to the overhead of spawning the tasks. In fact, this case study proves that it is not only necessary to have parallel machines, but also necessary to choose algorithms that can be efficiently parallelized. One possible way to efficiently parallelize this solution is to first explore the tree breadth-wise in a parallel fashion and then search depth-first in a sequential manner.
Chapter 8

CONCLUSION

Task parallelism is an important parallel programming paradigm that is emerging as a feasible solution in today’s ubiquitous multi-core environment. Through this study, we have demonstrated that it is possible to efficiently parallelize data parallel algorithms (EP, IS, and EP) using the task parallel model. Our results in these three benchmarks prove that the task parallel model indeed encompasses the data parallel constructs such as `foreach`. In this study, we have also examined the parallelization of a backtracking algorithm (Coin-change) using task parallelism. Our solution to this problem followed the depth-first search pattern, which is inherently difficult to parallelize. Furthermore, because of the lack of computations in the tasks themselves, a majority of the time was spent in spawning the tasks. Therefore, the parallel implementation of Coin-change was not as efficient as its sequential version. In fact, this case study proves that it is important to choose the right algorithm if case parallelism needs to be exploited. For example, a naïve recursive Fibonacci implementation cannot be efficiently parallelized for the very same reason --- the amount of computation in each function call is insignificant when compared to the actual task spawning and synchronization overheads. Instead, people often use matrix exponentiation to solve this problem.
BIBLIOGRAPHY


[7] Background Information about some of the TOP25 sites. [Online]

Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V.

Lumsdaine. PFunc: Modern Task Parallelism For Modern High Performance Computing.
In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC), Portland,
Oregon, November 2009.

Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and


[13] Levent Kandiller, Princicples of mathematics in operations research, Volume 10
[Online]

[14] David A. Bader, David R. Helman Joseph JaJa "Practical Parallel Algorithms for
Personalized Communication and Integer Sorting," Institute for Advanced Computer
Studies and Department of Electrical Engineering, University of Maryland.


[16] Mark Fineup, “Teaching Problem-Solving Techniques Utilizing a Running-Example Problem” Computer Science Dept., University of Northern Iowa, Cedar Falls, IA. [Online]