EXECUTIVE SUMMARY

Over the years, there has been a shift in the design of microprocessors which have now become multicore, that is, two or more processing units (cores) on a single chip. This switch to parallel microprocessors has fostered the use of parallel computing which involves simultaneous use of multiple compute resources to solve a computational problem. Today, with the large number of parallel programming models available, their comparison is unavoidable. Choosing one of the models requires information of their pros and cons and cases in which the models performance their best/worst.

In this project, we performed a brief survey of several parallel programming models and chose OpenMP (shared memory model) and MPI (message passing model) for comparison. Five significantly diverse applications, each belonging to a different Berkeley dwarf [1] where a dwarf is simply a class of applications having similar communication and computation patterns, are chosen. Using the knowledge gained from past works, suitable metrics are indentified to measure the performance and ease-of-use of the models. Performance is measured using a metric called speedup which indicates how much faster a parallel code runs than a serial one. Quantifying ease-of-use or programmability is not as easy task but we make an effort to measure it accurately using metrics like Lines of Code and development time. The MPI and OpenMP implementations of the five applications are run on University's high performance machine, BlueCrystal with different number of cores. The experimental results are analyzed to determine the pros and cons of both the models. The experience of implementing the applications in OpenMP and MPI was used to measure the development time metric. The two models are compared and contrasted in terms of performance and programmability using the identified metrics. The issue of portability is also briefly visited. It is determined that there is no clear winner, that is, no model is best for all cases. Thus the choice of a model depends on the type of application which is to be parallelized. This should help a programmer decide when to choose which model over the other. This work can also serve as a reference to compare to when evaluating some another parallel programming model.

- Two applications; Monte Carlo (Section 4.2.1) and Lattice Boltzmann model (Section 4.2.5) out of the five chosen for comparison were self-implemented in OpenMP and MPI.
- Using knowledge gained from past works, suitable metrics are identified which are used for comparing the two models in terms of performance and programmability.
- Two models were compared and contrasted by running OpenMP and MPI implementations of five different applications on the cores of BlueCrystal.

TABLE OF CONTENTS

INTRODUCTION 8
1 BACKGROUND 9
1.1 WHAT IS PARALLEL COMPUTING? 9
1.2 WHY PARALLEL COMPUTING? 9
1.3 GENERAL PARALLEL COMPUTIG TERMS 10
1.4 PARALLEL COMPUTER ARCHITECTURES 10
1.4.1 Shared Memory Architecture 10
1.4.2 Distributed Memory Architecture 11
1.4.3 Hybrid Architecture 11
1.5 TYPES OF PARALLELISM 11
1.6 SOURCES OF PERFORMANCE LOSS IN PARALLEL COMPUTING 12
1.6.1 Load Imbalance 12
1.6.2 Parallel Overhead 13
1.6.3 Locality of Reference 14
1.7 REVIEW 14
2 SURVEY OF PARALLEL PROGRAMMING MODELS 16
2.1 CRITERIA FOR EVALUATING PARALLEL PROGRAMMING MODEL 16
2.1.1 Inspired by Psychological Research 16
2.1.2 Independent of Number of Processors 16
2.1.3 Data sizes and types supported 16
2.1.4 Styles of parallelism supported 17
2.1.5 System Architecture 17
2.1.6 Programming Methodologies 17
2.1.7 Worker Management 17
2.2 PARALLEL PROGRAMMING MODELS 17
2.2.1 Pthreads 17
2.2.2 OpenMP 18
2.2.3 MPI 19
2.2.4 UPC 20

2.2.5 FORTRESS 21
2.2.6 CUDA 21
2.2.7 OpenCL 22
REVIEW 23
3 LITERATURE SURVEY AND METRICS FOR COMPARISON 24
3.1 BERKELEY'S DWARF TAXONOMY 24
3.2 PREVIOUS WORK ON COMPARING PERFORMANCE 25
3.3 PREVIOUS WORK ON COMPARING PROGRAMMABILTY 27
3.4 METRICS FOR COMPARISON 28
3.4.1 Performance Metrics 28
3.4.2 Programmability Metrics 30
3.5 REVIEW 30
4 DESIGN AND IMPLEMENTATION 31
4.1 CHOOSING PARALLEL PROGRAMMING MODELS FOR COMPARISON 31
4.2 CHOOSING APPLICATIONS 31
4.2.1 Monte Carlo simulation – Map Reduce 31
4.2.2 K-Means clustering – Dense Linear Algebra 32
4.2.3 Dijkstra's shortest path algorithm – Graph Traversal 33
4.2.4 1D Fast Fourier Transform – Spectral Methods 33
4.2.5 Lattice Boltzmann – Structured Grid 34
4.3 RUNNING APPLICATIONS ON BLUECRYSTAL 36
4.3.1 Input Data 36
4.3.2 Testing 37
4.3.3 Measuring time taken 37
4.4 MEASURING PROGRAMMABILITY 38
4.5 REVIEW 38
5 RESULTS AND ANALYSIS 39
5.1 Monte Carlo (Map Reduce) 39
5.2 K-Means (Dense Linear Algebra) 40
5.3 Dijkstra's Algorithm (Graph Traversal) 42
5.4 1D FFT (Spectral Methods) 44
5.5 D2Q9 Lattice Boltzmann (Structured Grid) 45

5.6 PROGRAMMABILITY 47
5.7 REVIEW 48
6 DISCUSSION 49
6.1 REFLECTION OF THE CODING EXPERIENCE 49
6.2 PERFORMANCE COMPARISON SUMMARY 50
6.3 PROGRAMMABILITY COMPARISON SUMMARY 52
6.4 COMMENTS ON PORTABILITY 52
6.5 FEW PROBLEMS FACED DURING THE PROJECT 53
7 CONCLUSIONS ABD FUTURE WORK 54
7.1 CONCLUSIONS 54
7.2 FUTURE WORK 55
BIBLIOGRAPHY 56
APPENDIX 57

LIST OF FIGURES

Figure 1: Concept of Parallel Computing 9
Figure 2: Symmetric Multiprocessor 10
Figure 3: Non-Uniform Memory Access 10
Figure 4: Distributed Memory 11
Figure 5: Hybrid Architecture 11
Figure 6: Data Parallelism 12
Figure 7: Task Parallelism 12
Figure 8: Load Imbalance 13
Figure 9: Cache Hit and Miss 14
Figure 10:Shared Memory Thread Model 18
Figure 11: OMP Architecture 19
Figure 12: MPI Message Passing 20
Figure 13: CUDA-Memory Hierarchy 22
Figure 14: OpenCL-Memory Hierarchy 23
Figure 15: Speedup-Amdahl's Law 29
Figure 16: Speedup Gustafson's Law 29
Figure 17: Calculating Pi 31
Figure 18: Standard K-Means 33
Figure 19: Butterfly compuatations 34
Figure 20: Lattice Grid 34
Figure 21: Decomposition of Grid 35
Figure 22: Exchange of Cells 36
Figure 23: Timming Parallel Programs 37
Figure 24: Sub and Super Linear Speedup 39
Figure 25: Speedup vs Number of Cores – Monte Carlo 40
Figure 26: Time vs Number of Cores for MPI – Monte Carlo 40
Figure 27: Speedup vs Number of Cores(I) – KMeans 41
Figure 28: Speedup vs Number of Cores(II) – KMeans 41
Figure 29: Time vs Number of Cores for MPI – KMeans 42
Figure 30: Speedup vs Number of Cores(I) – Dijkstra's Algorithm 43
Figure 31: Speedup vs Number of Cores(II) – Dijkstra's Algorithm 43
Figure 32: Time vs Number of Cores for MPI –Dijkstra's Algorithms 43
Figure 33: Speedup vs Number of Cores – 1D FFT 44
Figure 34: Time vs Number of Cores for MPI – 1D FFT 44
Figure 35: Speedup vs Number of Cores – Lattice Boltzmann 45
Figure 37: Speedup vs Relative LOC 47
Figure 38: Productivity of MPI and OMP for 5 applications 47

LIST OF TABLES

Table 1: Summary of Dwarfs 24
Table 2: Metrics used in Previous Work 27
Table 3:Cache Performance- KMeans..... 37
Table 4:Cache Performance- Lattice Boltzmann 45
Table 5: Languages Supported.....52

INTRODUCTION

It is evident from the history of computing that developers have always aimed at solving a problem faster. Parallel computing was introduced as a means to achieve this aim. In 2005 Intel followed the lead of IBM's Power 4 and Sun Microsystems' Niagara processor and announced that its high performance microprocessors would henceforth rely on multiple processors or cores. This led to a shift in the development of microprocessors design to multi-core architectures. Multicore became a buzzword which was coined to describe a microprocessor with multiple processors or cores on a single chip. This gave a new dimension to parallel computing the aim of which is to increase the application's performance by executing the application on multiple processors. It is expected that future generations of applications would heavily exploit the parallelism offered by the multi-core architecture.

However, development of parallel programs is not a trivial task and requires an appropriate parallel decomposition of algorithms. The associated programmability problem has led to the introduction of a plethora of parallel programming models that aim at simplifying software development by raising the abstraction level. The industry has not settled for a single model, that is, there is no clear winner. Thus with advent of the multicore architecture and introduction of large number of parallel programming models, while developing parallel applications the programmer should correctly identify the parallel programming model to be used which will make the best use of the underlying hardware. So, there is a need to evaluate these models against each other which will introduce comparability and help developers choose a programming model that fits best with their requirements.

AIMS AND OBJECTIVES

The main aim of this project is to evaluate pros and cons of two parallel programming models; OpenMP (shared memory model) and MPI (Message Passing model) and compare them on the basis of their performance and ease-of-use. We need to identify suitable metrics to measure performance and programmability for comparing and contrasting the two models. We aim at selecting applications from different domains which will be implemented in MPI and OpenMP for comparison. All the programs are to be run on BlueCrystal, University's high performance machine.

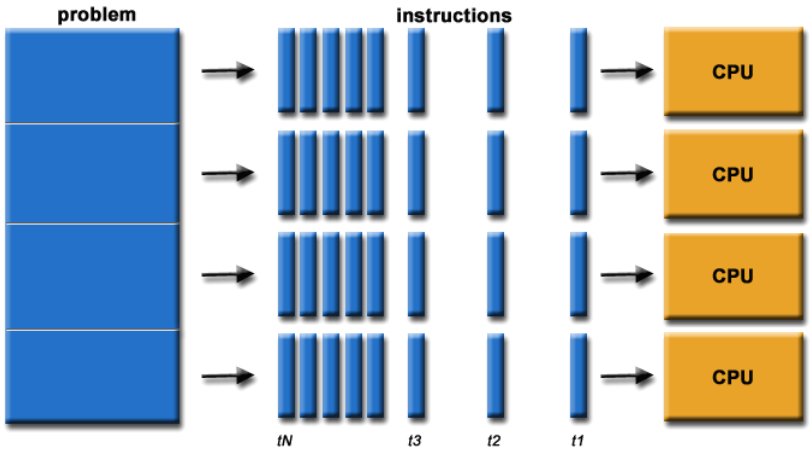
The objective of this project is to determine which model is suitable for which kind of applications and aid a programmer in making an informed choice of a parallel programming model suiting their requirements. We also have the objective of developing and extending our knowledge of parallel computing which is becoming very popular. Our ultimate goal is to produce a work which can serve as a reference to compare to when evaluating new or different programming models.

1 BACKGROUND

1.1 WHAT IS PARALLEL COMPUTING?

Simultaneously using multiple compute resources to solve a computational problem is termed as Parallel Computing [2]. As shown in Figure 1, taken from [2], the problem is broken down into discrete parts that can be solved simultaneously. Each part is divided into series of instructions and these instructions from each part are executed concurrently on different CPUs. The resources used to run the instructions can be a single computer with multiple processors or a number of computers connected by a network or a combination of both

Figure 1 Concept of Parallel Computing



The use of Parallel Computing started in the areas of science and engineering which had a number of computation intensive applications. Examples of such areas are geology, seismology, circuit design, molecular sciences, nuclear physics and so on. But today, parallel computing has become an important part of the commercial applications as well which provide an equal or greater driving force in development of faster computers. They require processing of large amounts of data, for example in data mining, financial and economic modeling, web search engines, oil exploration, medical imaging and diagnosis and so on.

1.2 WHY PARALLEL COMPUTING?

Use of parallel computing has become very popular and in many cases imperative. Following are a few reasons for its increasing prevalence -

- **Solve larger problems-** Many problems are so large that it is impractical to solve them on a single computer with limited memory. Using multiple processors is the only option.
- **Save time** - With parallel computing the time taken for a task to complete will be much less than in serial computing. Also, building parallel resources is not expensive so the cost incurred is not much.
- **Provide concurrency** - With multiple resources many different independent tasks can be done at a time. For example if one processor we are calculating the velocity of a particle, we can use the other processor for calculating the density which is independent from velocity.

- **Overcome the limits of serial computing -** There are physical and practical constraints on building faster serial computers. The only solution to this is using multiple computers.

1.3 GENERAL PARALLEL COMPUTIG TERMS

Several commonly used terms in parallel computing and their meanings are mentioned below. These terms will be used throughout the thesis.

- **Multicore** - A computer with multiple processors having shared memory address space.
- **Unit of Execution (UE)** - The processing elements working concurrently during the parallel execution of an application. They are usually threads or processes.
- **Barrier** - It is a technique used to synchronize the Units of Execution such that no UE can cross the barrier until all the UEs have finished their work and reached the barrier.
- **Tasks** – They are basically sequence of operations or part of the computational work executed by a processor and which together make up the entire program.
- **Node** – It is a computational element, having its own memory and comprises of single or multiple processors. Nodes are networked together to forms a distributed system.
- **Latency** - Time taken for a byte to travel across the network between two nodes
- **Bandwidth** – Number of bytes that can travel across the network between two nodes in unit of time.
- **Speedup –** Measure indicating how much faster does the parallel code run than a serial one.
- **Scalability** - Ability of a parallel system to have a proportionate increase the execution speed of the program as the number of processors are increased.

1.4 PARALLEL COMPUTER ARCHITECTURES

1.4.1 Shared Memory Architecture

All processors in the system have access to the same single global address space. Although the processors operate independently, they share the same memory resources. Generally, all the processors are placed equidistant from the shared memory, thus the time taken to access the memory is same for each of them [2]. This type of system is known as Symmetric Multiprocessor (SMP) machine shown in Figure 1. Usually, two or more SMPs are connected with each other to form Non-Uniform Memory Access (NUMA) system as shown in Figure 3. In this, processors do not have equal time access to the memory.

Figure 2- SMP

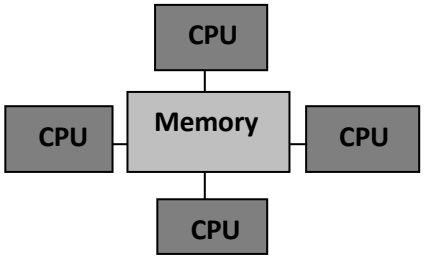
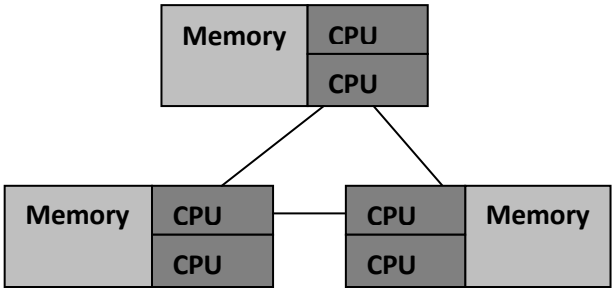


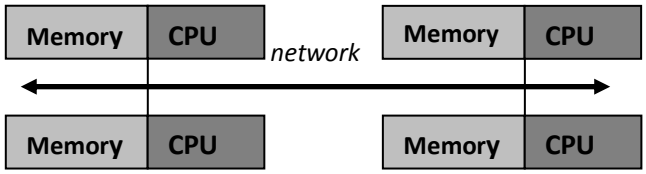
Figure 3 - NUMA



1.4.2 Distributed Memory Architecture

All the processors have their own private memory and there is no common shared memory. Processors are inter-connected via a network using simple Ethernet or any fast communication links as shown in Figure 4. Changes to the local memory of the processor have no effect on the memory of other processors [2]. The programmer defines how and when data is transferred among the processors. Processors communicate with each other by sending or receiving messages.

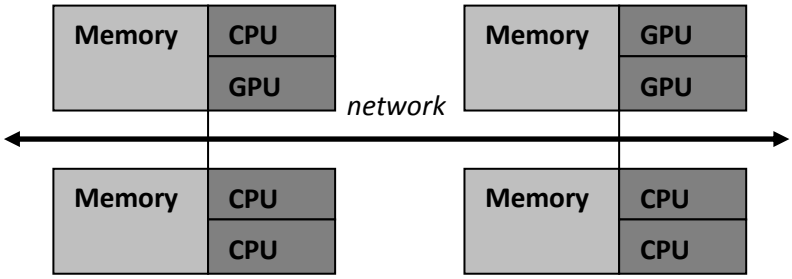
Figure 4 – Distributed memory



1.4.3 Hybrid Architecture

It is a combination of shared and distributed architectures [2]. As can be seen from Figure 5, they are distributed machines consisting of smaller multicore shared machines called as nodes which can be CPUs or Graphic Processing Units (GPUs). Communication between nodes takes place either by message passing across the network if the nodes are on different machines or locally on the same node via shared bus.

Figure 5 – Hybrid Architecture

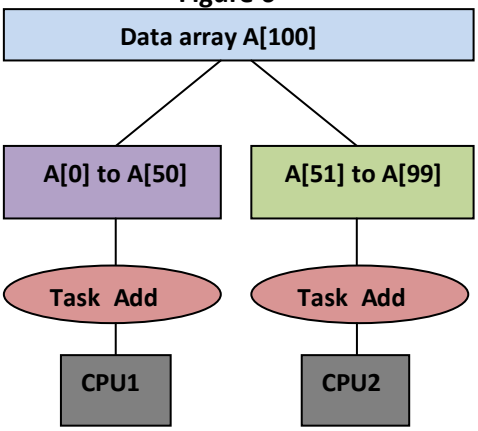


1.5 TYPES OF PARALLELISM

To convert an application from serial to parallel, we first need to identify the parts which can be parllelised.We need to decompose a problem into smaller sub-problems in order to find concurrency in the application which is exploited while parallelizing the serial problem. Fundamentally, there are two ways to decompose a problem and get parallelism –

1.3.1 Data Parallelism

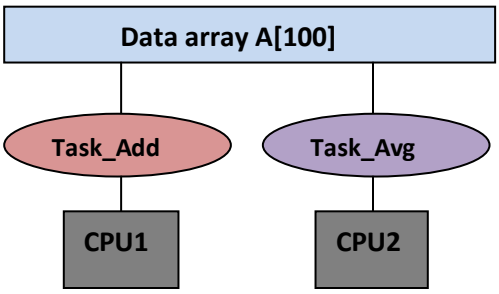
Figure 6



The data is partitioned into smaller subsets and tasks are performed on each subset i.e. identical tasks are performed in parallel [2]. As only one processor directs the activities of all other processing elements, there is a single control flow. Figure 6 explains the concept where the data A[100] is divided into two subsets and a task of addition is performed on both the sets in parallel by different processors. This kind of parallelism scales easily as it effectively utilizes the processing elements available.

1.3.2 Task Parallelism

Figure 7



The entire task of the program is partitioned into sub-tasks which can be run concurrently [2]. Thus unique tasks run in parallel on common data. Degree of parallelism is limited to number of sub-tasks that have been formulated. In the Figure 7 the data A[100] is used by two processors simultaneously to perform different tasks, add and average.

1.6 SOURCES OF PERFORMANCE LOSS IN PARALLEL COMPUTING

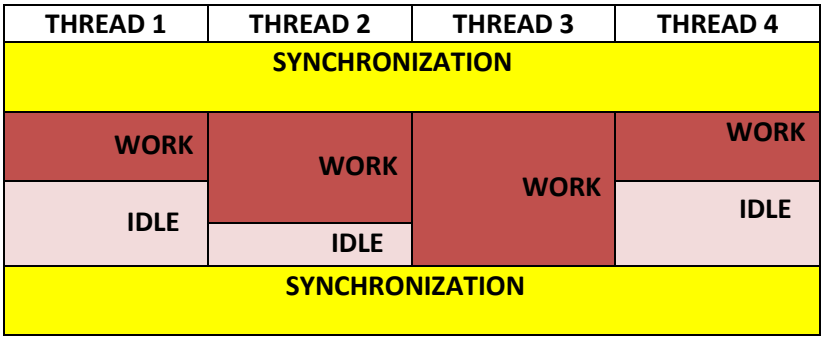
While running a parallel application on a parallel system, we expect the execution time to decrease linearly as we increase the number of processors. For example, if a serial program runs in time T and then we parallelize it and run it on 4 processors, we expects it to run in time T/4 that is we expect to achieve a linear speed up. However, this is an ideal case and in reality this is not very achievable due to certain issues that cause a performance loss. We mention several common reasons of performance of loss in parallel computing.

1.6.1 Load Imbalance

Load imbalance occurs when work is not evenly assigned to the parallel working elements like threads or processes [3]. When there is an imbalance in assignment of the load, some working elements finish early and have to wait for the working element having largest amount of work to

complete. This leads to idle times and low processor utilization which adversely affects the aim of achieving a linear speedup.

Figure 8 – Four threads having different workload size



1.6.2 Parallel Overhead

It is the cost incurred in parallel solution but not required by sequential solution. It can be measured in terms of the amount of time required to coordinate parallel tasks as opposed to doing useful work. Parallel overhead includes the following factors

1.6.2.1 Communication Overhead

In parallel computing, threads or processes have to communicate with each other for transferring data for example private sum has to be communicated to the master. As sequential solution does not require any communication, this entire communication causes as overhead in the parallel solution. Shared memory (Section 1.4.1) communication is cheaper than communication across a network (Section1.4.2) or non-uniform memory architecture.

1.6.2.2 Synchronization Overhead

This overhead is specific to shared memory architecture. This is incurred when a thread has to wait for a particular event on another thread or for a resource to get free which is being used by another thread [3]. As there is no shared data or resources in message passing model, there is synchronization overhead involved.

1.6.2.3 Software Overhead

Parallel programming models are available in form of a library or API or as an extension to current programming language which imposes software overhead of parallel compilers, libraries, tools and so on [3].

1.6.2.4 Memory Overhead

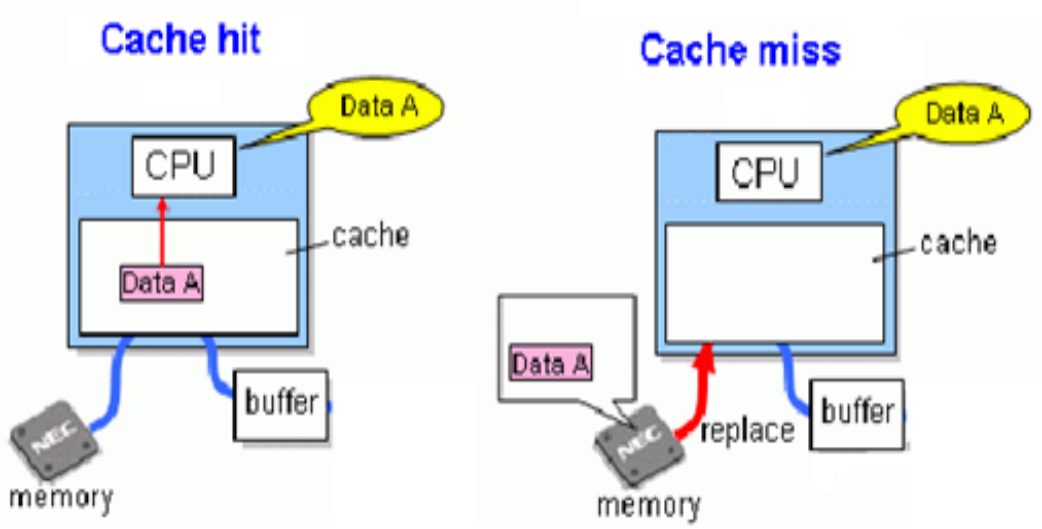
If a particular application requires N units of memory, it is not necessary that it will require N/4 units of memory when run in parallel using four processors. Parallel computations frequently

require more memory than the sequential computation. Applications with significant memory overhead will experience performance loss.

1.6.3 Locality of Reference

This issue is specific to the shared memory programming models. An application can show temporal and/or spatial locality. Temporal locality implies that if some data is referenced, then it is very likely that it will be referenced again in the near future. Also, if some data is referenced then there is a high probability that data nearby will be referenced in the near future which is termed as spatial locality. An application will benefit by storing the most often or most frequently used data in the cache as accessing data from a cache is much faster than from the main memory.

Figure 9 – Cache Hit and Miss



As shown in Figure 9, if requested data is present in the cache, it is termed as *cache hit* as for fetching this data no access to main memory is required. If the requested data is not found in the cache, it is called a *cache miss* and data is fetched from the main memory which consumes more time [3]. Applications having poor data locality show poor performance due to ineffective use of cache causing longer stall times waiting for main memory accesses

1.7 REVIEW

In this chapter, we have briefly introduced the concept of parallel computing and mentioned few reasons for using it. Commonly used terms in parallel computing and their meanings were listed. We discussed three different parallel computer memory architectures focusing on how the multiple processors are placed and how the memory is accessed. Decomposition of a problem to achieve parallelism is an important and challenging step in parallel programming. We studied two fundamental types of parallelism; data and task

which are logically contrast to each other. Finally, we briefly described the general reasons for performance loss in parallel computing due which achieving ideal linear speedup is difficult.

2 SURVEY OF PARALLEL PROGRAMMING MODELS

Parallel programming models enable the expression of parallel programs which can be compiled and executed. They present an abstraction above the hardware and memory architecture and allow expressing ideas in particular ways. They can be implemented in several ways such as language extensions, libraries invoked from sequential language or complete new execution models. There have been a myriad number of parallel programming models proposed. The development of microprocessors design has been shifting to multicore architectures. Parallelism is playing a significant role in future generations of applications. In choosing a parallel programming model, not only the performance aspect is important, but also qualitative the aspect of how well parallelism is abstracted to developers. A model with a well abstraction of parallelism leads to a higher application-development productivity.

2.1 CRITERIA FOR EVALUATING PARALLEL PROGRAMMING MODEL

Before studying the different programming models, it is essential to determine the various aspects on the basis of which the models should be evaluated. According to Asanovic et. al. [1] following four criteria must be used as the basis for assessing the parallel programming models. For analyzing the models it should be taken into consideration whether or not the models support the following features.

2.1.1 Inspired by Psychological Research

Humans are the ones which will use the programming models and thus they strongly affect the success of the model. Despite this fact, the research from psychology has had no impact. Programming model development in the past has been hardware-centric, application-centric, or formalism-centric. Hardware-centric ones are developed by manufacturers to increase the performance or efficiency of their hardware. Application-centric models aim to make the related application domain development easier, for example, Matlab. Formalism-centric models, such as Actors, have clean semantics and verify the correctness of portions of code. Although these characteristics are important, they do not support the broad process of parallel programming. Human cognition is not being considered while designing the parallel programming models. In future, the models must be human-centric focusing the human process of efficiently implementing, debugging and maintaining a complex parallel program. Thus whether or not a programming model has been inspired by psychological research can be one of the criteria for evaluating the parallel programming models.

2.1.2 Independent of Number of Processors

A few present models like MPI, explicitly deal with decomposing data, mapping tasks, and performing synchronization over thousands of processing elements. In future the models should not expose the number of processors, which will result in a high level of abstraction to the programmers.

2.1.3 Data sizes and types supported

The parallel research agenda inspires new languages and compilers. For investigating a parallel model, the data

sizes and types the model supports, should also be analyzed. Thus the models should support at least the primitive data types which include

- 1 bit (Boolean)
- 8 bits (Integer, ASCII)
- 16 bits (Integer, DSP fixed point, Unicode)
- 32 bits (Integer, Single-precision floating point, Unicode)
- 64 bits (Integer, Double-precision floating point)
- 128 bits (Integer, Quad-Precision floating point)
- Large integer (>128 bits) (Crypto)

2.1.4 Styles of parallelism supported

There are various styles of parallelism (Section 1.5) like independent task parallelism, word-level parallelism, bit-level parallelism. Most of the models present support only one style of parallelism. But with experiments it has been observed that some styles of parallelism have proven successful for some applications, and no style has proven best for all. The models should support different styles of parallelism so that they can cover larger number of applications. Thus the different styles of parallelism supported, should be considered while evaluating the parallel model

2.1.5 System Architecture

While examining a parallel model, it is important to study which memory architecture it supports. In Section 1.4 we have explained the different parallel system memory architecture. By determining the system architecture supported by the model, w can understand the limitations of the model or different issues involved with the specific model [5]. For example models supporting only shared memory architectural will have scalability issues as applications can run and utilize only processors within a single node and not on the nodes across the network.

2.1.6 Programming Methodologies

This evaluates how parallelism capabilities are exposed to programmers [5]. This may be in form of Application Programming Interface (API), special directives or new language specification.

2.1.7 Worker Management

This considers how unit of worker, threads or processes are created and managed. It is implicit if the programmer does not have to be bothered about the creation or destruction of the unit of worker and is explicit if its lifetime has to be managed [5].

2.2 PARALLEL PROGRAMMING MODELS

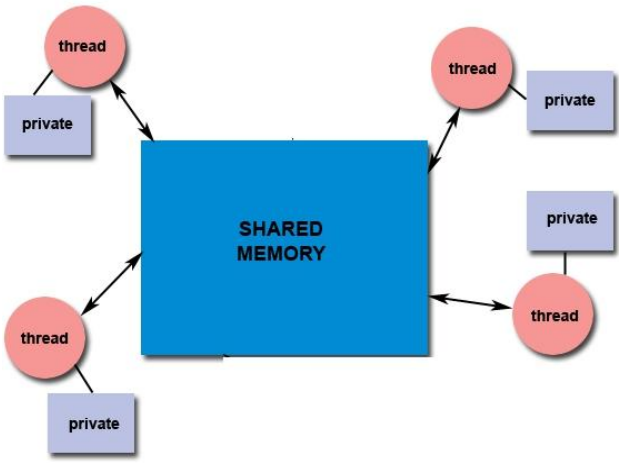
For selecting a few models for comparison, we need to do a research on various parallel programming models available and evaluate them on the basis of the criteria mentioned in Section 2.1. In this section we study several popular programming models.

2.2.1 Pthreads

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Pthreads or

Portable Operating System Interface (POSIX) threads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header [6]. Threads must be explicitly created and destroyed by the programmer using pthread_create and pthread_exit function. As shown in Figure 10, all threads have access to the same global, shared memory and each thread has its own private memory as well. Programmer must synchronize access to globally shared data. Programmer must be aware of data race and deadlocks and protect the critical section which is the portion of code that accesses shared data.

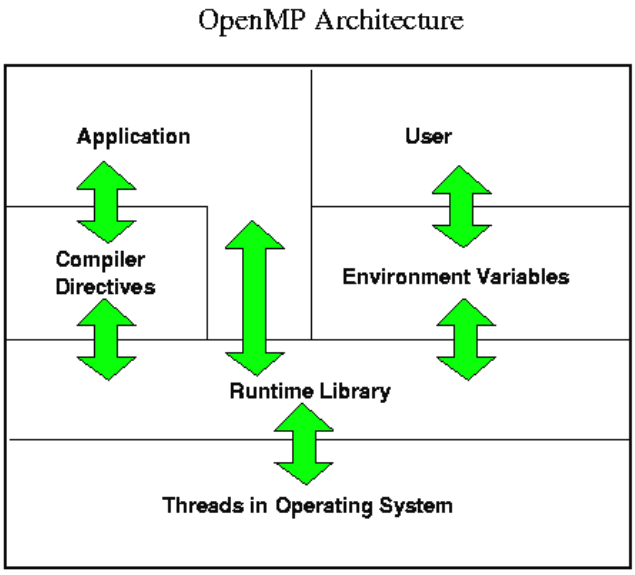
Figure 10 – Shared memory thread model



2.2.2 OpenMP

Open Multi-Processing (OpenMP) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that extend Fortran, C and C++ programs [7].It is portable across the shared memory architecture. Threads are the unit of worker. All threads have access to all the memory. It uses the fork-join model of parallel execution. It is intended to support programs that will execute correctly both as parallel programs and as sequential programs. Parallel region is denoted by specifying compiler directive (i) #pragma omp parallel {} for C/C++, and (ii) !$omp parallel and !$omp end parallel for Fortran. The OpenMP program begins execution as a single thread of execution called the master thread. The master thread runs in a serial region until the first parallel construct is encountered, when the master thread creates a team of threads, and becomes master of the team. There are work-sharing constructs like for, sections, single which distribute the execution of the associated statement among the members of the team that encounter it. Worker management, division of workload into task and assigning of tasks to thread is implicit. Thus less programming effort is required. The programmer is also relieved from the responsibility of synchronization.

Figure 11

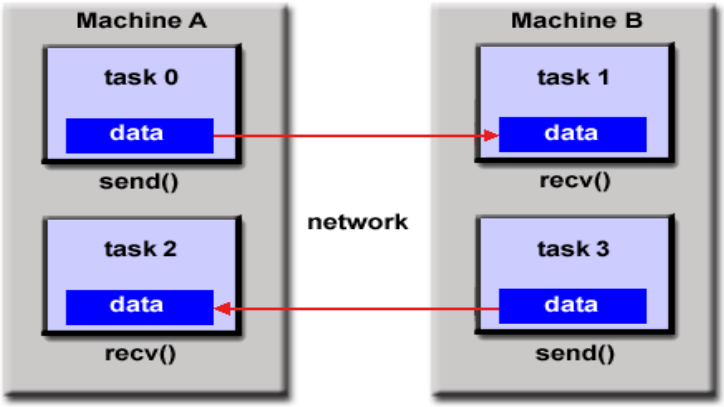


As illustrated in Figure 11, an OpenMP aware compiler will be capable of transforming the code-blocks marked by OpenMP directives into threaded code. At run time the user can then decide (by setting suitable environment variables) what resources should be made available to the parallel parts of his executable, and how they are organized or scheduled.

2.2.3 MPI

Message Passing Interface (MPI) is a specification for message passing operations. Processes are the unit of worker. It spawns the application over a number of processes. Each process runs conceptually on one physical processor [8]. These processes communicate with one another by passing messages.MPI is currently the de-facto standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++, and Fortran. OpenMPI, MVAPICH, MPICH, GridMPI, and LAM/MPI are some of the MPI implementations. Worker management is done implicitly thus the programmer does not have to code for creation, scheduling or destruction of processes. All parallelism is explicit, i.e the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. MPI_COMM_WORLD is the predefined communicator that includes all the MPI processes. Routine MPI_rank() returns the rank of the current process in the team of processes and MPI_size() routine gives the number of processes present. The message passing operations are either point to point such as MPI_Send/MPI_Recv which facilitates communications between two processes or collective which facilitate communications involving more than two processes. Figure 12 shows how communication between two processes takes place by passing messages using send() and recv(). When a synchronization point is needed, MPI_Barrier is used which blocks each process from continuing its execution until all processes have entered the barrier.

Figure 12



2.2.4 UPC

Unified Parallel C (UPC) is an extension of the C programming language designed for high-performance computing on large-scale parallel machines. It is a parallel programming language for shared memory architecture and distributed memory architecture. It adopts the concept of Partitioned Global Address Space (PGAS) in which the programmers view the system as one global address space which is logically partitioned into a number of per-thread address spaces. Each thread has two types of memory accesses: to its own private address space or to other threads' address space [9]. Accesses to both types of per-thread address space use the same syntax.UPC uses a concept called thread affinity to optimize memory-access performance between a thread and the per-thread address space. For worker management, programmers just need to specify number of threads required. Workload management is implicit and workload partitioning is explicit or implicit. upc_forall API is used for implicit work partitioning and task mapping. Thus to map the task to threads does not require additional programming effort. Whereas in the explicit approach, the programmers have to specify what will be run by each threads. upcrun command is used to run these programs and for worker management, the number of threads have to be specified with this call. Communication among threads, adopt the PGAS paradigm by making use of pointers. There are three types of pointer commonly used in UPC-

1. private pointer where the private pointers point to their own private address space.
2. private pointer-to-share where the private pointers point to the shared address space.
3. shared pointer-to- share where the shared pointers from one address space point to the other shared address space.

There are several synchronization mechanisms available in UPC. Following are the five different synchronization constructs in UPC [9]

1. Barrier - A blocking synchronization.
2. Split phase barriers - A non-blocking synchronization.
3. Fence - Ensures that all shared references is completed before its use.
4. Locks - Protect the shared data against multiple processors.
5. Memory consistency model - It can be relaxed in which shared data can be reordered during compile time or runtime or strict in which access to shared data is serialized.

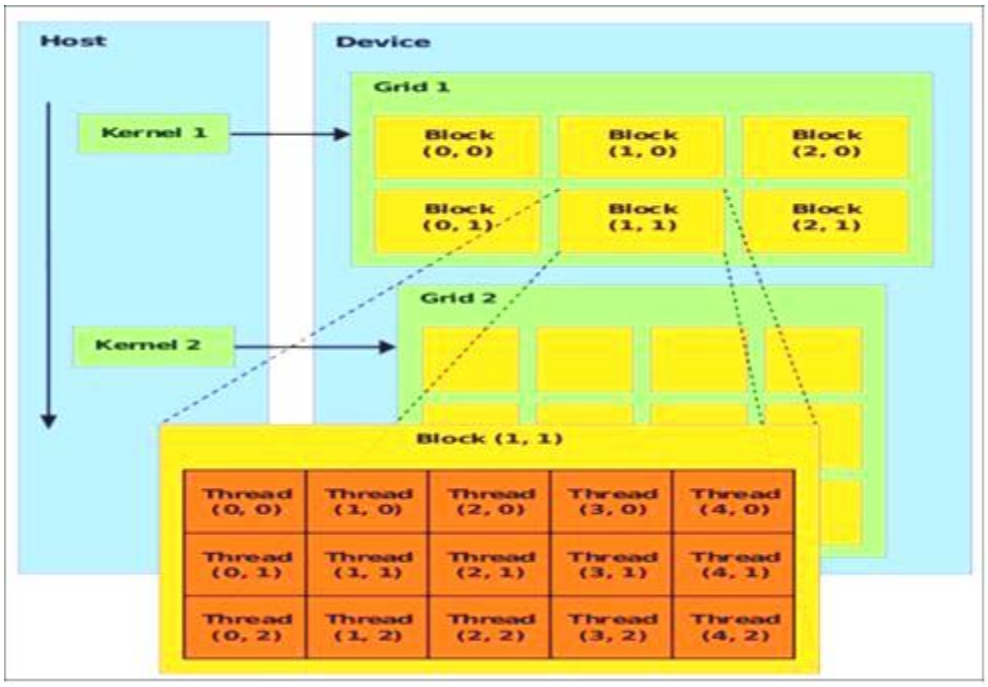
2.2.5 FORTRESS

The Fortress programming language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability []. Two basic concepts in Fortress are that of object and of trait. An object consists of fields and methods. The fields of an object are specified in its definition. An object definition may also include method definitions. Traits are named program constructs that declare sets of methods. In addition to objects and traits, Fortress allows the programmer to define top-level functions. Functions are first-class values: They can be passed to and returned from functions, and assigned as values to fields and variables. The programs are organized into components, which export and import APIs. APIs describe the "shape" of a component, specifying the types in traits, objects and functions provided by a component [10]. The unit of worker is threads. There are two kinds of threads in Fortress: implicit threads and spawned (or explicit) threads. Spawned threads are objects created by the spawn constructs. Worker management, workload partitioning and worker mapping in Fortress can be implicit or explicit. Fortress also supports a rich set of operations for defining parallel execution of large data structures. This support is built into the core of the language. For example, for loops in Fortress are parallel by default. There are two synchronization constructs, reductions and atomic expressions, for avoiding data races and unpredictable program behavior. In reduction computations are performed as local as possible to avoid the need of synchronization. Atomic expression consists of atomic keyword followed by body expression. It is used to control the data among the parallel executions.

2.2.6 CUDA

CUDA (Compute Unified Device Architecture) is the extension of C programming language designed to support of parallel processing on Nvidia GPU(Graphics Processing Unit) [12]. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. Using GPUs for performing computations in application which are traditionally performed by the CPUs is called general purpose computing on graphics processing unit (GP-GPU). CUDA programs are run in two parts, one on the host (i.e CPU) and the other on the device (i.e GPU). At GPU, a very large number of threads run in parallel and there is a two level hierarchy present, namely block and grid as shown in Figure 13 taken from [12]. Block is a set of tightly coupled threads where each thread is identified by a thread ID, while grid is a set of loosely coupled of blocks with similar size and dimension. A sample prototype of a CUDA kernel is __global__ void sampleKernel(float *a, float *b, float *c). The __global__ keyword simply indicates that this function may be called from either the host PC or the CUDA device. Every thread at the GPU executes the kernel. Each thread runs the same code, so the only way for them to differentiate themselves from the other threads is to use their threadId, and their blockId. Worker management in CUDA is done implicitly; programmers do not manage thread creations and destructions.

Figure 13 – Memory hierarchy in CUDA



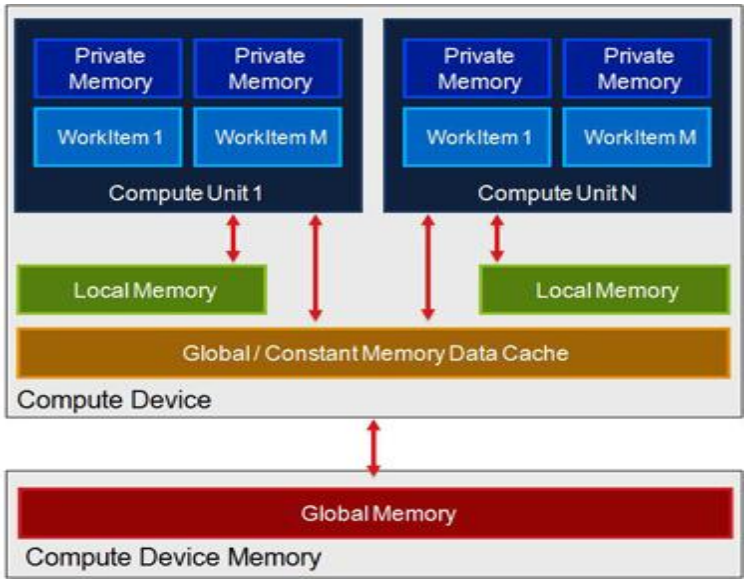
They just need to specify the dimension of the grid and block required to process a certain task. Using Global Function<<<dimGrid, dimBlock>>> (Arguments) construct, the programmer can define the workload to be run in parallel. Here, Global Function is the global function call to be run in threads, dimGrid is the dimension and size of the grid and dimBlock is the dimension and size of each block [20]. Function __syncthreads() is used to perform implicit synchronization of threads.

2.2.7 OpenCL

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs and other processors [11]. It supports general purpose parallel computations and is highly suitable for applications having to do large number of computations and which can be done in parallel. Like CUDA, OpenCL programming involves running code on two different platforms, host and device. One host has one or more compute devices each with one or more compute units each with one or more processing elements. Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The context of the kernel is defined by the host program which also manages the kernel's execution. Unlike CUDA, with OpenCL one can write accelerated portable code across different devices and architectures. The unit of worker is a work item. Each work item is identified with a unique global_id. Work-items are grouped into local work group and each work-item within a group has a unique local_id as well. Synchronization between work-items is possible only within workgroups by using barriers and memory fences. Outside the group, synchronization is not possible. All work is submitted to the device through queues. Each device must have a queue. When there are multiple devices, each device will have its own queue and synchronization between these queues must be done explicitly. Each queue can execute in order or out of order. There are four distinct memory regions in an OpenCL memory model as shown in Figure 14, taken from [11]

- Global Memory
- Constant Memory
- Local Memory
- Private Memory

Figure 14- OpenCL Memory Hierarchy



OpenCL uses a relaxed consistency memory model, i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. Workload partitioning can be implicit or explicit. Size of the workgroup can be explicitly defined by setting the global_size and local_size in the host program. While the OpenCL application itself can be written in either C or C++, the source for the application kernels is written in a variant of the ISO C99 C-language specification. These kernels are compiled via the built-in runtime compiler, or if desired, are saved to be loaded later.

REVIEW

The chapter started with listing several criteria which are useful to appraise the appropriateness of a parallel programming model. They not only help in assessing the performance of the models but also in evaluating their ease-of-use. It should be noted that these criteria are not exhaustive because other implementation issues like debugging can also be considered for evaluation. There are a number of parallel programming models available today. A survey of seven widely used models was done and each model was briefly discussed on the basis of the previously mentioned criteria. This survey has given us an in depth knowledge about the models and has set up a strong base for deciding which programming models to choose for comparison. Moreover, we also understood what factors to consider while studying a parallel programming model. So in future, if we need to investigate a new parallel model, we know the approach to be taken.

3 LITERATURE SURVEY AND METRICS FOR COMPARISON

This chapter focuses on investigating previous research work done comparing several parallel programming models in terms of performance and programmability. The knowledge gained from this research will be used to decide the metrics to be used for comparing in terms of performance and programmability. We first study the Berkeley's dwarf taxonomy [1] to understand how applications to be parallelized can be categorized in different classes on the basis of communication and computation patterns.

3.1 BERKELEY'S DWARF TAXONOMY

A multidisciplinary group of Berkeley researchers have defined a set of 13 Dwarfs, where each dwarf captures a pattern of computation and communication that is common to a class of important applications. Membership in a particular class of dwarf is defined by similarity in computation and data movement [1]. It is claimed that two applications belonging to the same class of dwarf is expected to show similar trends in performance. Table 1 summarizes the thirteen dwarfs, giving a brief description of each and a few example application that belong the particular category of dwarf.

Table 1 – Dwarf Summary

Dwarfs	Description	Example
Dense Linear Algebra	Data are dense matrices or vectors. They have high ratio of math-to-load operations and a high degree of data interdependency between parallel working elements	K-Means, ,Block triadiagonal Matrix
Sparse Linear Algebra	Data sets include few non-zero values. To reduce space and computation, data is usually stored in compressed form i.e. in a list of values and indices.	Conjugate Gradient, SuperLU
Spectral Methods	The data is transformed to frequency domain from time domain. The execution profile is typically characterized by multiple stages of processing, where dependencies within a stage form a "butterfly" pattern of computation. Typically, spectral methods use multiple butterfly stages, which involves all-to-all communication for some stages and strictly local for others.	Fast Fourier Transform
N-Body Methods	Involves large number of calculations within a timestep and all-to-all communication between timesteps. Interactions between many discrete points are calculated.	Barnes-Hut, Fast Multipole Method
Structured Grids	The data is organized in a regular grid format. Points on grid are conceptually updated together. Each point on grid is updated using values of neighborhood points.	Lattice Boltzmann, Game of Life

Unstructured Grids	Data is irregular grid and connectivity to a neighboring point is made explicit. Update to a point on grid requires determining a list of neighboring points, and then loading values from those neighboring points.	Back Propagation
Map Reduce	Comprises of redundant execution of a ”map” function and results are aggregated using a “reduce” function. Considered embarrassingly parallel.	Monte Carlo, similarity scores
Combinational Logic	Covers applications implemented with bit-level logic functions. Simple operations are performed on a very large amount of data.	Encryption/Decryption
Graph Traversal	Number of connected objects are visited and evaluated. Random memory look ups are involved. Latency tends to become a bottleneck.	Quick Sort, Dijkstra’s
Dynamic Programming	A complex problem is solved by solving a series of simpler subproblems.	Needleman-Wunsch
Backtrack	A very large space is searched to find an optimal solution. To limit the computations, the search space is pruned using an implicit method.	Neural Networks
Construct Graphical Models	Graphs are constructed in which variables are represented by nodes and conditional probability by edges.	Bayesian Networks.
Finite State Machines	Compasses a system whose behavior is defined by state and on input there is a transition in state.	String match, Parser.

The point of these dwarfs is to identify the core computation and communication for important future applications and help in developing programming models that will efficiently run the future applications. It has been made clear that the list of dwarfs is not complete and more dwarfs can be added to the list to ensure that a broader range of applications are covered.

3.2 PREVIOUS WORK ON COMPARING PERFORMANCE

Due to introduction of large number of parallel programming models, comparison of these models is unavoidable. Researchers have put in a lot efforts to fairly compare various models. Many experiments have been carried out in which same applications are run in different languages and results have been analyzed. Following section describes some of the previous works done in comparing parallel programming models.

[13] has discussed the implementation details the NAS Parallel benchmarks in OpenMP (Section 2.2.2). It examines the effectiveness of OpenMP in parallelizing the benchmark suite. It compares the performance of the OpenMP implementation with the MPI one on the basis of execution time. It concludes that the performance of OpenMP is quite good and very close to that of MPI but the setback is that the OpenMP implementation does not scale well as the number of cores is increased. Although, remarkable efforts have been made to introduce the OpenMP version of the NAS suite, the comparison with MPI is only on the basis of execution time and no other metric has been used.

[14] evaluates the performance of MPI (Section 2.2.3), OpenMP and UPC (Section 2.2.4) programming models on multicore architectures. It compares the performance of MPI and UPC on hybrid systems, that is, system with both shared and distributed memory, using NAS parallel benchmark suite. [14] also runs experiments to compare the performance of OpenMP, UPC and MPI models on shared memory. After the analysis of the experimental results it concludes that MPI performs the best on both hybrid and shared memory due to its efficient handling of data locality. Speedups obtained for both MPI and UPC are better in the case of shared memory. OpenMP cannot handle data locality well and has limited scalability. This paper has used only the applications from NAS Parallel benchmark suite** which focuses only on CFD applications. Thus these conclusions about the programming models cannot be generalized for all the application domains.

[15] compares the performance of CUDA(Section 2.2.6) and OpenCL (section 2.2.7) on the basis of data transfer times to and from the GPU, kernel execution times, and end-to-end application execution times using a specific real world application. It concludes that for applications similar to the one used for comparison, CUDA is better choice as it performed better than OpenCL especially when transferring data to and from GPU. Thus [15] aids the developers for choosing between CUDA and OpenCL for a particular class of applications. Extending this work for different application domains like computer vision, HPC, image processing and so on will be helpful for the programmers of these domains to make a correct choice of programming model.

[16] implements the Rodinia benchmark** suite in OpenMP and OpenCL and shows the differences between the two implementations. More importantly, it compares the two models on the basis of performance, portability and productivity. Performance is measured in terms of execution time. Portability indicates the reusability of solution built using the model. Productivity is the measure of development effort which is measured in terms of lines of code (LOC). It concludes that productivity of OpenMP is better, while OpenCL is portable for a larger class of devices and performance wise either can outperform the other. It would be interesting to extend this work done by finding what application features and parameters favor OpenMP or OpenCl which will help developers to make an informative choice between these two models.

[17] presents a new benchmark suite called evaluation suite to classify and compare shared memory parallel programming models. Using this suite, the performance and programmability of two models, POSIX threads (Section 2.2.1) and OpenMP is compared. Although no clear conclusion is drawn on the choice between these two models, valuable information on the scaling and speedup characteristics of the models is given and further extension of the evaluation suite is suggested.

3.3 PREVIOUS WORK ON COMPARING PROGRAMMABILTY

This section discusses few of the previously done work for comparing the ease-of-use of different parallel programming paradigms.

[18] carried out usability studies on two groups of Computer Science students, having no prior knowledge in parallel programming. Students were asked to solve the same problem on two different programming models. Table 2 taken from [18] lists some of the important issues considered in this work which affect the assessments of parallel programming systems.

Table 2 Different Metrics for Assessment

CATEGORY	ASSESSMENT METRIC
Performance	benchmark results speed of code generated memory usage turnaround time
Applicability	portability hardware dependence programming languages supported types of parallelism supported
Usability	learning curve probability of programming errors functionality integration with other systems deterministic performance compatibility with existing software suitability for large-scale software engineering power in the hands of an expert ability to do incremental tuning

In this work, some of the metrics used yielded no statistically conclusive results. There was no right selection in terms of the size and expertise in the focus group. The applications selected were not versatile and very limited in number. Thus after reading this paper, we decided to select applications from different domains and use only those metrics for comparing the programmability which we thought worked well.

[19] defines a new metric called Development time productivity for evaluation of parallel programming models in terms of programmability. It is basically a ratio of relative runtime performance to relative programmer effort.

$$Productivity = \frac{Speed}{Relative\ Effort} \qquad Relative\ effort = \frac{Parallel\ Effort}{Serial\ Effort}$$

The authors carried out several classroom assignments in which students were asked to develop parallel programs for some textbook applications. Experience of students in terms of hours spent on coding was also used to measure the parallel effort .We intend to use this metric in our project and measure the parallel effort in terms of Lines of Code and development time.

3.4 METRICS FOR COMPARISON

We not only intend to compare the performance of the models but also aim at taking a human-centric approach and compare on the basis of their ease-of-use, that is, how convenient or inconvenient is it for the programmer to implement the application using the particular model. Using the knowledge gained from past work, the metrics used to evaluate this project will be divided into 2 parts: performance and programmability.

3.4.1 Performance Metrics

3.4.1.1 Execution time

For measuring the performance of the models, execution time metric can be used. For fairness of comparison we should time only the parallel sections of the code. Also an accurate timing value, we will run the experiment for 5 times and then take the average. Thus the average time taken for the code to run for different number of cores will be recorded for every programming model

3.4.1.2 Speed Up

Speed up is another metric for measuring the performance. It is calculated by dividing the execution time on one processor by the execution time on n processors for the same program, thus normalizing the speedup factor for a single core to one. Ideally if we use *n* processors we would expect our problem to be solved *n* times as fast but this is rarely achieved.

3.4.1.2.1 Amdahl's Law

This law has been widely used to get an approximate estimation of improvements in performance of a parallel application when alternate implementations and designs are tried. It states that "*The performance improvement to be gained from some faster mode of execution is limited by the fraction of the time that the faster mode can be used.*" [20] The faster mode of execution implies program parallelization.

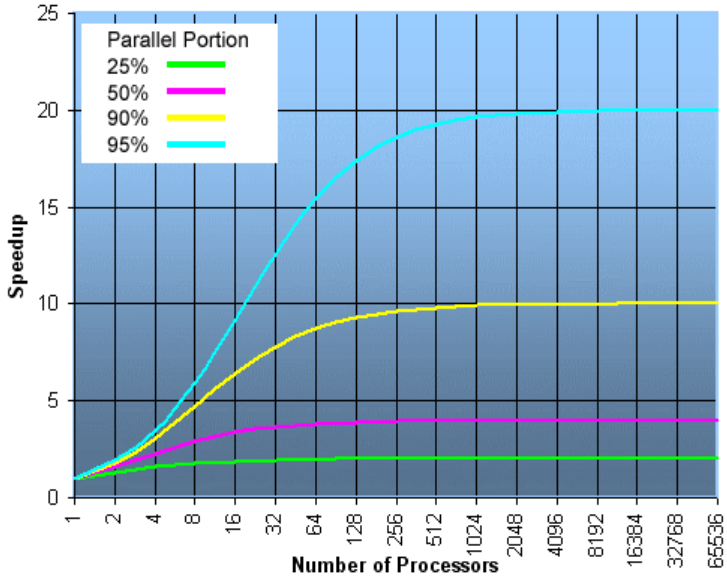
$$Speedup = \frac{1}{\left(\frac{P}{N}\right)+(1-P)}$$

P = Fraction of program that can be parallelized

N = Number of processors running in parallel.

Figure 15 taken from [2] shows how speedup in parallel computing is limited by fraction of code that cannot be parallelized. As it can be observed, according to this law, parallel programming s more beneficial if the program has a larger parallelizable part.

Figure 15



3.4.1.3 Gustafson's Law

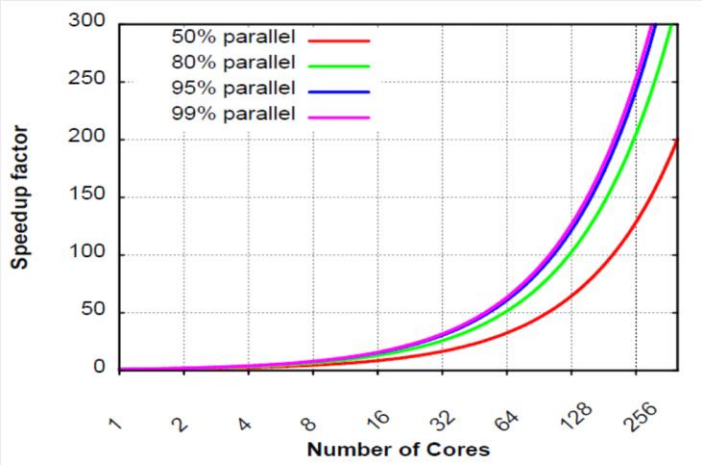
This law provides a counter point to Amdahl's Law. According to this law, applications having large data sets as input can be efficiently parallelized. Figure 16 taken from [3] shows how speedup is affected with increasing the number of cores according to this law. Following formula is used to calculate the speedup

Speedup = N(N-1).np

N = Number of processors running in parallel.

np = Fraction of the application that is sequential**.**

Figure 16



In this project we run the application on different number of cores and for different data set sizes to get an insight of the above two laws. We use the following formula to calculate the speedup S. Here Ts is the time taken to solve a problem on a single processing element and Tp is the time required to solve the problem on a parallel computer with p identical processing elements [15].

S= Ts/Tp.

3.4.2 Programmability Metrics

The concept of programmability is very difficult to quantify as we have to consider factors such as a programmer's performance. As discussed previously, efforts have been made to efficiently quantify this metric but there is no accurate way to do so. Various metrics have been introduced to measure the programmability. Although not extremely accurate, we can measure the *parallel effort* required to develop a parallel code using the following metrics.

3.4.2.1 LINES OF CODE (LOC)

LOC is a metric to measure the development effort, that is, efforts taken by a programmer to convert the serial code into parallel using a particular parallel model. We must make sure to include all aspects of the code in this metric, for example, for an OpenCL code LOC will be the total of LOC of host program and LOC of kernels.

3.4.2.2 DEVELOPMENT TIME

It is the time required by a programmer to actually code the entire program using a specific programming model. To ensure that the productivity is not affected by the learning curve, sufficient time should be spent learning each programming model prior to actually writing the code.

Using these two metrics, we will calculate the productivity which was introduced by [] as mentioned in Section3.3

3.5 REVIEW

In this chapter we discussed the 13 dwarfs defined in [1] which form different classes of parallel applications depending on communication and computation patterns. We investigated previous work done to compare the parallel programming models on the basis of performance and programmability. We observed that programmability is a very subjective issue and quantifying it accurately is difficult. Although many efforts have been made to do so, more work needs to be done in this area. Using the knowledge gained from past work we decided several metrics which will be used in this project to measure and compare performance and ease-of-use of a programming model.

4 DESIGN AND IMPLEMENTATION

In this chapter we outline the tasks done in the project. This includes selecting two parallel models out of the pool of models discussed in Section 2.2 for comparison, deciding the application to be used for comparison, running the codes on BlueCrystal and measuring the metrics discussed in Section 3.4.

4.1 CHOOSING PARALLEL PROGRAMMING MODELS FOR COMPARISON

After studying several popular programming models we decide to choose OpenMP (Section 2.2.2) and MPI (Section 2.2.3) for comparison. Our choice is dictated by the fact that both models are widely used and are different from each other. OpenMP has a shared memory model whereas MPI is the de facto standard for message passing. With the limited time of 3 months available for the project, choosing two models and not more seemed a feasible option as this gives sufficient time for covering good amount of applications and doing detailed analysis of the results. Another reason of choosing these models was that they were already introduced to us in the HPC unit. Thus relatively less time was required in getting familiar with the models and understanding them in detail. Moreover, the survey of the parallel models (Section 2) has developed a curiosity in knowing whether shared memory or message passing model performs better and for which kind of applications.

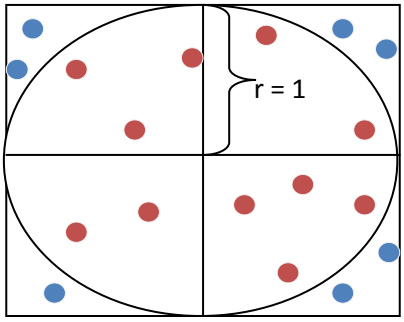
4.2 CHOOSING APPLICATIONS

Berkeley's 13 Dwarfs (Section 3.1) are used to choose the applications which will be implemented (or codes will be obtained from the web if available as open source) in OpenMP and MPI. We limit our focus to a subset of dwarves. We aim to choose applications from different domains such as data mining, fluid dynamics, graph algorithms and so on. Following section briefly describes the chosen applications and mentions the dwarf each belongs to.

4.2.1 Monte Carlo simulation – Map Reduce

In Monte Carlo simulations, we use large amount of random samples or random tests to find some aggregate property of the system. We use this simulation to calculate the value of Pi. We consider a circle of radius 1 with centre as the origin within a square that just contains the circle. A large number of random points are generated in the square, and the number of points that lie within the circle and those that lie outside are counted as shown in Figure 17.

Figure 17- Calculating Pi



$$\textit{Value of Pi} = \frac{\textit{Number of sample points inside the circle}}{\textit{Total number of sample points}}$$

We notice that this application is “embarrassingly parallel” as there is no interaction needed among the parallel working elements and all random points can be generated simultaneously and tested whether they lie inside the circle or not. Thus it falls under the Map Reduce dwarf. Using all this information and the knowledge of programming in OpenMP and MPI, the application was implemented in OpenMP and in MPI.

4.2.1.1 OpenMP Implementation

First we implemented a serial code which randomly generates numbers between 0 and 1, checks whether the generated number is inside the circle (using the equation of circle for unit radius) and increments the counter if it does. Lastly the ratio of the counter value and the total number of samples generated gives the area of the circle with unit radius which is also the estimation of the pi value. Now for the OpenMP implementation, we simply follow an incremental approach for parallelization. We add the compiler directive *# pragma omp parallel* construct outside the *for* loop. We also use the *reduction* clause so the value of the counter gets correctly updated and there are no race conditions. We note here that there is no synchronization of threads involved and there is data parallelization that is each thread takes its part of the samples, tests whether the numbers fall inside the circle or not and update the counter accordingly.

4.2.1.2 MPI implementation

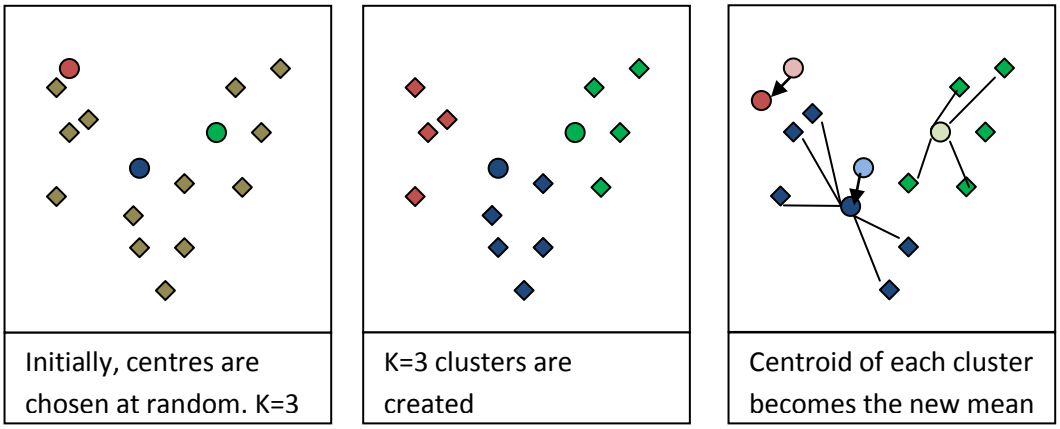
We again start with the serial code for the MPI implementation. As mentioned in Section **, in MPI there are processes. We generate the random numbers between 0 and 1 at each process. Thus every process has access only to the samples generated by it. Each process maintains a local counter and checks whether the sample is within the circle or not and increments the counter if it is. We then use *MPI_Reduce()* function to sum the value of the counter maintained at each process. We observe that there is no communication between the processes and this is why the application is called “embarrassingly parallel”.

Thus we have successfully implemented the Monte Carlo simulation for calculating the value of Pi using MPI and OpenMP.

4.2.2 K-Means clustering – Dense Linear Algebra

K-Means is an easy to implement and efficient clustering algorithm used extensively in data-mining. In k-means, a data object is comprised of several values, called features. By dividing a cluster of data objects into K sub-clusters, k-means represents all the data objects by the centroids of their respective sub-clusters. The first K elements of the input data are picked as the initial K cluster center coordinates. In each iteration the algorithm associates each data object with its nearest center, based on some chosen distance metric. The new centroids are calculated by taking the mean of all the data objects within each sub-cluster respectively. The algorithm iterates until no data objects move from one sub-cluster to another. Figure 18 demonstrates how a standard K-means algorithm works.

Figure 18 – Standard K-Means Algorithm



The source code for the parallel implementation of the K-Means algorithm in MPI and OpenMP is taken from [21]. It uses data parallelism (Section 1.3.1). Input data objects to be clustered are evenly partitioned across all processes or threads.

4.2.3 Dijkstra's shortest path algorithm – Graph Traversal

This algorithm is used to find the length of shortest path between two vertices (source and destination) of a graph. A set of nodes for which the shortest paths are known is maintained. This set is grown by adding nodes closest to the source using one of the nodes in the current shortest path set. We note that this application does not involve large amount of computation but there is a lot indirection involved while traversing the graph. The OpenMP and MPI implementation of this algorithm is taken from [3].The two codes implement the Dijkstra's algorithm for finding the shortest paths from vertex 0 (source) to the other vertices in an N-vertex undirected graph. In each case the vertices of the graph are distributed among the unit of execution (UE). After performing the local computations, the result is conveyed to the master UE.

4.2.4 1D Fast Fourier Transform – Spectral Methods

FFT is an efficient algorithm to compute Discrete Fourier Transform using the formula given below.

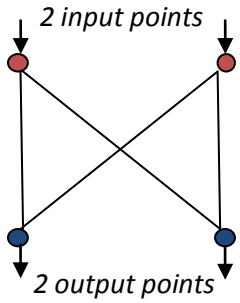
$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n} jk} \qquad j = 0, \ldots, n-1.$$

We use a 4-step Matrix Fourier algorithm implemented by [22] in OpenMP and MPI. It takes a NxN input matrix and includes the following steps.

- Apply a FFT on each column of the input matrix.
- Multiply each matrix element (index r, c) by the twiddle factor which is trigonometric constant coefficient.
- Apply a (length C) transform on each row.

- Transpose the Matrix.

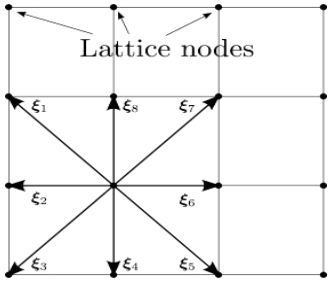
Figure 19



The basic computational element of FFT is converting two complex points into two other complex points and is known as "butterfly" due to its winged appearance as shown in Figure 19.

4.2.5 Lattice Boltzmann – Structured Grid

Figure 20



Lattice Boltzmann equation is used to simulate hydro dynamical. It is constructed on a lattice space having fluid particles. We use a D2Q9-BGK model having 9 discrete velocities in X-Y plane as shown in Figure 20. We observe that the data in this application is organized as a regular 2 dimensional grid. Also, updating an individual data element depends on a number of neighboring elements.

Thus this model falls under the *Structured Grid* dwarf. We use the serial code for this model was provided in the HPC unit simulating the flow of a fluid as it hits an obstacle. Firstly, we optimized this serial code and then parallelized it using OpenMP. After this, the optimized serial code was parallelized using MPI.

4.2.5.1 OpenMP Implementation

For parallelizing the optimized serial code using OpenMP, we started with parallelizing the *collision()* function. Combined parallel work sharing construct, *#pragma omp parallel for*, with the clauses private, schedule was added before the outer for loop in the *collision()* function. Similar construct was introduced in the functions *accelerate(),propagate(),rebound(), av_velocity().* In function *av_velocity()* an additional clause reduction*(op:variable-list)* was added. This clause performs a reduction on the scalar variables that appear in variable-list, with the operator op. I observed the changes in time taken after introducing parallelism in each function. Thus a first version of the parallel program was created which had parallelism in each function resulting in large amount of data parallelization (Section 1.3.1).

Now to introduce more parallelism we had to focus on task parallelism (Section 1.3.2). With OpenMP this can also be achieved to a certain level. For this, we made a change in the *timestep()* function which is removal of the call to function *propagate()* which was now called from the function *accelerate()* which resulted in forming a pipeline of the two tasks *accelerate()* and *propagate()*.

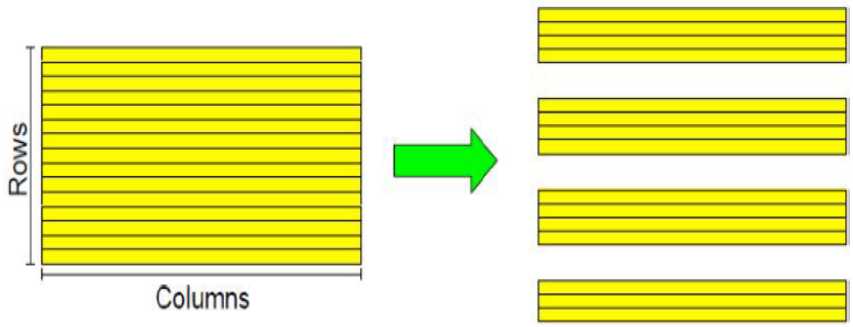
While converting a serial code to parallel, following points were considered to ensure correctness-

- Avoid race condition which is anomalous behavior caused by the unexpected dependence on the relative timing of events.
- Avoid two threads accessing the same location concurrently.
- Minimize the update of global variables.

4.2.5.2 MPI Implementation

While making the serial code parallel using MPI, the goal is to balance the CPU load of the serial application over several processors in order to reduce the overall execution time. I started with creating MPI processes and dividing the entire data space into the number of processes created. Each part was assigned to a particular process which performs its computation locally. Thus every process has its copy of tmp_cells and cells. I used MPI_rank() which returns the rank of the current process in the team of processes and MPI_size() routine for getting the number of processes present. The 300x200 input data grid was divided in the following manner along the X axis as shown in Figure 21.

Figure 21- Decomposition of Grid



It was observed that there is an interdependency of data belonging to different processes which must be taken care of. In function *propagate()* the values of *tmp_cells[]*, present in the other processes, are changed. Basically, it affects the value of *tmp_cells* present in the next process and in the previous process. Thus before the function *collision()* starts, these values must be copied correctly in the appropriate processes. The processes have to communicate with each other and exchange the *halo (*or *ghost cells)* correctly. MPI point-to-point operation was used which involves message passing between two different MPI processes. One process performs a send operation and the other performs a matching receive operation. ***MPI_Sendrecv()*** was used which combines a send and a receive into a single MPI call and make them happen simultaneously to avoid deadlock.

Figure 22 – Exchange of cells

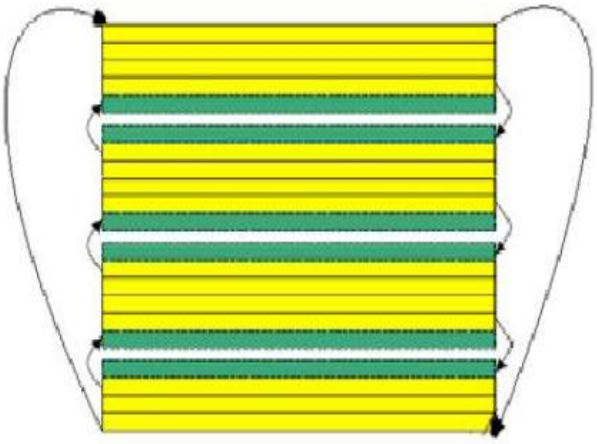


Figure 22 shows how exactly halo or ghost cells are swapped with a wrap around. After successful exchange of the halo, the *collision()* is called function which is the final step of the program. Each process runs this function on the local data ad sends this processed data to the master process. Thus all data is copied back to a common shared address space. We successfully implemented the lattice Boltzmann model in MPI. While collecting the source codes from the web, we made sure that the MPI and OpenMP codes for a particular application was taken from the same source so there is no difference in implementation in terms of algorithm followed or data structures used and the codes are equally optimized. This provides the basis for a fair comparison between the programming models.

4.3 RUNNING APPLICATIONS ON BLUECRYSTAL

Bluecrystal is University of Bristol's high performance machine. All the source codes were executed on the high memory nodes present on phase two of BlueCrystal having four 6 core processors**. Thus we have a maximum of 24 cores having shared memory (Section 1.3.1). The OpenMP and MPI codes are run on the 24 shared memory nodes and the results are compared. As MPI can run on distributed memory (Section 1.3.2), we also run the MPI codes on the distributed network on BueCrystal having 32 nodes each with 4 cores. Thus we could run the MPI codes and measure the performance on 32x4 = 128 cores. Although we cannot use these results to compare with OpenMP as OpenMP is strictly for shared memory architecture, we use them to analyze the performance of MPI and study the trends on larger number of cores. For all the experiments, the number of threads or processes are kept equal to the number of cores.

4.3.1 Input Data

An application requires data to be processed to generate an output. The input data for our programs was either taken from the same source from where the source code was obtained or in some cases the input data was randomly generated in the program itself. Table 3 mentions the source of the input data for every application used in the project.

Table 3 – Applications and the source of corresponding Input data sets

Application	*Source of Input Data*
Monte Carlo	The data is randomly generated in the program. User only needs to provide the number random samples to be generated.
K-Means clustering	We use two input data files taken from [21] having 204800 and 819200 data objects to be clustered.
Dijkstra's algorithm	The graph is randomly generated in the program and the user only needs to provide the number vertices in the graph.
1D FFT	The source code has a function to randomly generate the NxN input matrix
Lattice Boltzmann	Data file of size 3000X2000 was provided in the HPC unit

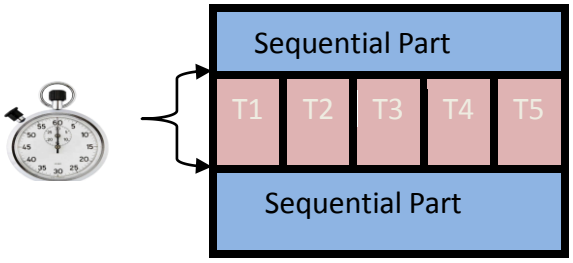
4.3.2 Testing

While executing the parallel programs, we must ensure that the program is actually running in parallel and gives correct output. We have serial code for each application. We thus tested the correctness of the parallel programs by comparing their results with that of the serial program. We assure that the serial programs are running correctly by testing them using smaller inputs for which output is already known. For example, in case of Dijkstra's shortest path algorithm (Section 4.2.3), we gave several graphs with small number of nodes as inputs and checked the respective outputs which we could easily compute beforehand. Not all applications can be tested in such a manner but as the serial codes are taken from trusted third parties, we assume they run correctly. For example the Lattice Boltzmann (Section 4.2.5) serial code was given in the HPC unit and so we could safely assume that it gives correct results. For checking whether the program is running concurrently, in case of OpenMP, we made use of the *omp_get_max_threads()* function which gives the total number of threads created and use *omp_get_thread_num()* to print the computations done by each thread. In MPI, we use the *MPI_Comm_rank()* function to output the work done by each process. Our main concern with MPI was to make sure that correct data is written to correct memory location.

4.3.3 Measuring time taken

We run all the OpenMP and MPI codes on different number of cores, up to 24 and note down the time taken in each case. We run each code five times and take the average of the five values and we re-ran the program for the outputs that differed significantly. A program contains a serial part and a parallelizable part, as shown in Figure 23 and we time only the parallelizable part of the program using the *gettimeofday()* function in C.

Figure 23 – Timing the Parallel Program



4.4 MEASURING PROGRAMMABILITY

As mentioned in Section 3.4.2, Productivity is the metric used for measuring the Programmability. To calculate productivity we need data like *Lines of Code (LOC)* and Development time (Section 3.4.2.2). For every code in OpenMP or MPI, we measure LOC ignoring the white spaces and comments. *Development time* is measured only for the applications which were implemented by us i.e. Monte Carlo simulation for calculating value of Pi (Section 4.2.1) and Lattice Boltzmann method (Section 4.2.5). While we were implementing the application in OpenMP and MPI, we kept track of the approximate amount of time spent to code each application in MPI and OpenMP in terms of hours and use this as a development time metric.

4.5 REVIEW

We chose OpenMP and MPI for comparison. A subset of the Berkeley's dwarf was selected and the applications to be used for comparison were chosen such that each application comes under a different dwarf thus covering more application domains and different patterns of parallelism. Each chosen application was briefly described in this chapter and we mentioned which part of the each application could be parallelized. The source codes were either self-implemented or collected from the web and ran on BlueCrystal. The MPI codes were run on shared memory (to fairly compare with OpenMP) and on distributed memory (to analyze the performance on larger number of cores). We then briefly mentioned how testing was carried out and how execution time, lines of codes and development time were calculated. Thus, this chapter summarizes the various tasks performed in this project.

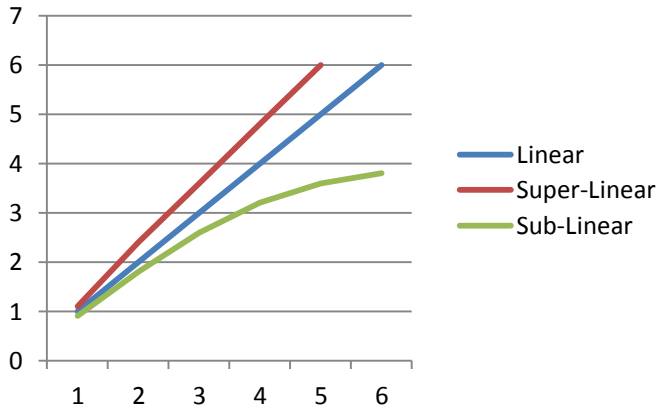
5 RESULTS AND ANALYSIS

This chapter analyzes the performance of the five applications (Section 4.4) implemented with MPI and OpenMP, using the performance metrics previously defined. The models are compared in terms of performance and programmability using the metrics previously defined. Performance of MPI is separately analyzed on distributed memory wit larger number of cores.

We start with understanding the following terms which will help in analyzing the performance.

- Linear Speedup – This speedup increases linearly with increase in number of processors. It is the ideal expected speedup.
- Super Linear Speedup – The speedup obtained is more than the linear speedup. That is speedup increases faster than the number of processors.
- Sub Linear Speedup – This is the real speedup which flattens after certain number of processors are added

Figure 24



Using this knowledge, we now discuss the performance of OpenMP and MPI for each application

5.1 Monte Carlo (Map Reduce)

The MPI and OMP codes for calculating the value of Pi were run and timings were measured as mentioned in Section 4.3.3. Speedup was calculated as mentioned in Section 3.4.1.2. Figure 24 shows the speedup obtained by running the OpenMP and MPI codes on different number of cores using 100000000 random samples. These speedups are compared to the ideal linear speedup. We observe that the performance of both OpenMP and MPI is close to the ideal speedup. Up to 4 cores the performance of both are similar. As we increase the number of cores to eight and more the performance of both OpenMP and MPI starts deviating from the ideal speedup. Also, we observe that OpenMP performs little slower than MPI as we increase the number of cores. As the only important step in this application is the reduction step that is summing the total number of samples inside the circle, the slower performance of OpenMP suggests that the reduction in MPI works faster than that in OpenMP. Thus if we have to perform sum of a very large array or find maximum or minimum element of an array, we should use MPI as it has a better performance than OpenMP.

Figure 25

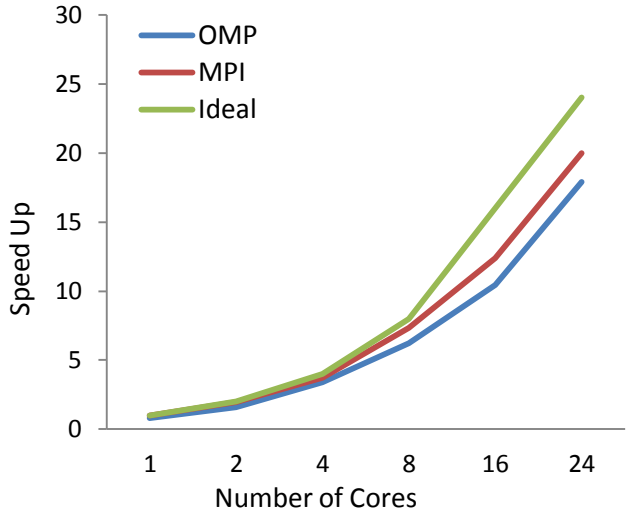
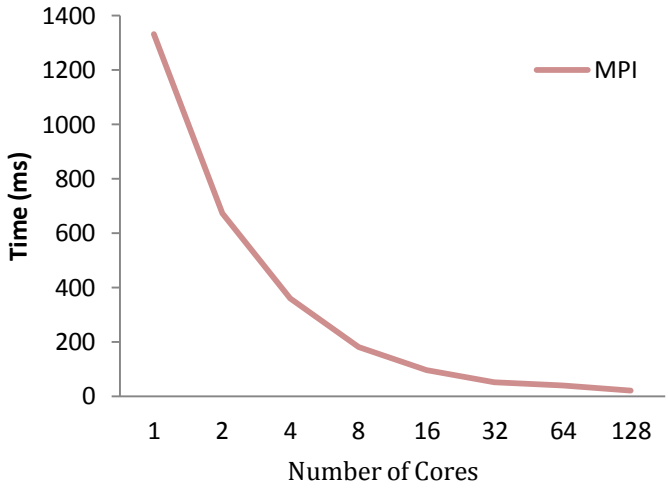


Figure 26



In this project, we not only compare the performance of the two models on shared memory but also analyze the performance of MPI on distributed memory with larger number of cores. Figure 26 shows the time taken to run the MPI codes on different number of processors on distributed memory architecture. We observe that the MPI code scales really well that is the execution time decreases proportionately as we increase the number of processors (at least up to 32 cores). This is expected because the application is "embarrassingly parallel" and has negligible inter process communication. We observe that curve tends to flatten after 32 cores. This is because the computation time at each process is negligible and there is a parallel overhead in terms of time taken to start up the processes is involved due to which the execution time remains almost constant and is approximately equal to the time taken to startup the parallelization.

5.2 K-Means (Dense Linear Algebra)

We first executed the parallel codes using an input dataset having 204800 objects and setting the value of *K* to 10 which means that 10 clusters will be formed of the input data set. Figure 27 shows the speedup obtained for MPI and OpenMP on different number of cores. We can see that up to 4 cores the performance of both the models is comparable but as the cores increase, the performance of OpenMP starts to deviate from the linearity. From 16 cores, the speedup gained flattens indicating parallel saturation.

Figure 27-Data size = 204800

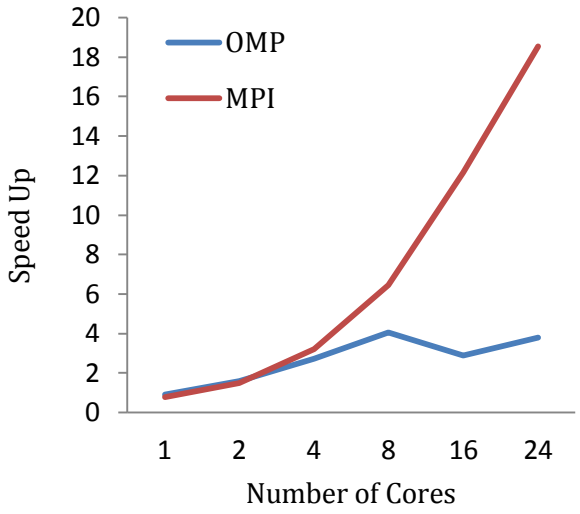
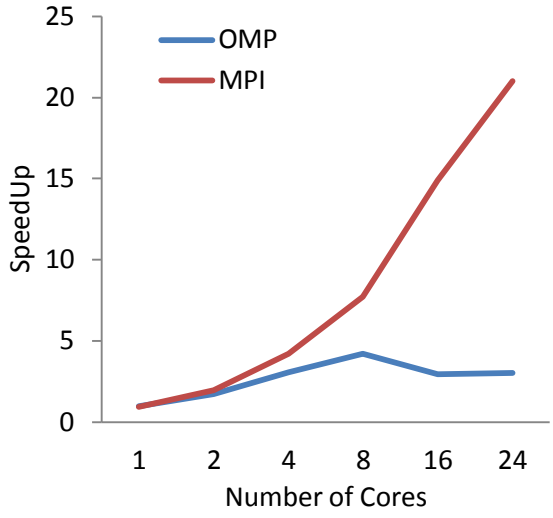


Figure 28 – Data size = 819200



The reason behind this is the data locality issue discussed in Section 1.6.3. The number of cache read misses increases due to which more time is taken as data is to be read from the main memory. To confirm this we used a tool called *Cachegrind* belonging to the *Valgrind toolkit* [24] which is a cache and branch prediction profiler. It gives the number of cache referrals, the cache read misses and cache write misses. Basically it summarizes the cache accesses for instructions executed and for data. Table 4 shows the result of using cachgrind for 2, 8 and 16 threads. *D1 misses* indicates the number of times data not found in the D1 cache. LLd misses indicated the number of time data not found in the LL cache.

Table 4 – Cache Performance with different number of Cores

	D1 misses	*LLd misses*
2 cores	15,090,140	14,022,930
8 cores	14,006,103	12,676,394
16 cores	14,006,826	12,671,351

We observe that for 2 threads the misses are high and for 8 threads, the cache misses reduce greatly. Thus for 8 threads we get a speedup as expected and is not affected by cache misses. But for 16 threads, the cache misses roughly stay the same (slightly increase for D1 and slightly decrease for LLd) due to which even though the parallelism has been increased expected speedup is not achieved as accesses to main memory remain the same.

Whereas the performance of MPI is really good almost achieving linearity**.** As in MPI every process has its own private memory, there is no issue of data locality.

Figure 28 shows the performance for a larger input dataset with 819200 objects. We observe that the two graphs are identical to each other. OpenMP still has performance saturation after 8 cores which is expected because if for smaller data size there is data locality issue then for larger data the issue will definitely persist as the cache size remains the same so more data is in the main memory. We see that the performance of MPI still scales very well and the curve has a similar, close to linearity shape as before.

Figure 29

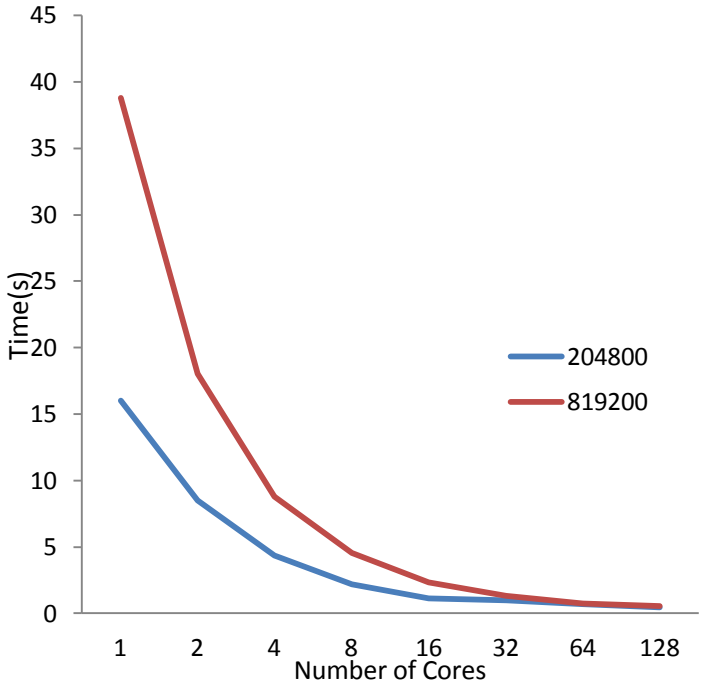


Figure 29 shows the execution time taken for running the MPI codes in distributed memory system with a maximum of 128 cores for two input datasets of size 208400 and 819200. We notice that the timing curve begins to flatten for smaller data sets from 16 cores whereas for larger data sets it flattens from 32 cores. This implies that there is more performance gain for larger data set than for smaller and the reason for this is that in case of the larger data there is more computation and so the computation to communication ratio is higher than that in case of smaller data set. Thus processor spend comparatively less time on communication than on computation due to which the curve does not start to flatten as early as in the case of the smaller data set.

5.3 Dijkstra's Algorithm (Graph Traversal)

The Dijkstra's shortest path programs for MPI and OpenMP were run initially for 100 graphical nodes. From Figure 30 we observe that for smaller number of graph nodes (100), the more the parallelism we had the slower the code ran in case of both MPI and OpenMP. Thus we see the speed up curve go down as the number of cores increase. The reason is that the synchronization overhead (Section1.6.2.2) for both MPI and OpenMP is large and could not be compensated by parallel computations which are not very large. So instead of running faster, the program actually runs slower with increase in number of cores. We then run the application for larger number of nodes that is 20000 nodes.

Figure 30

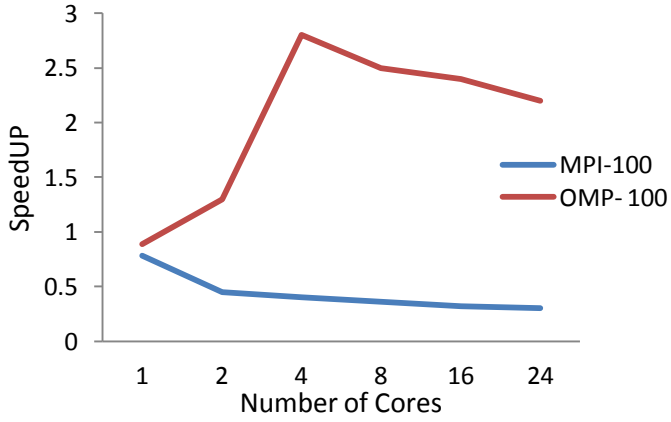
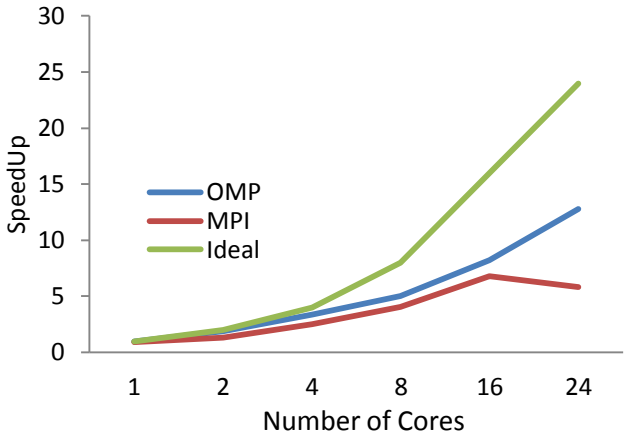
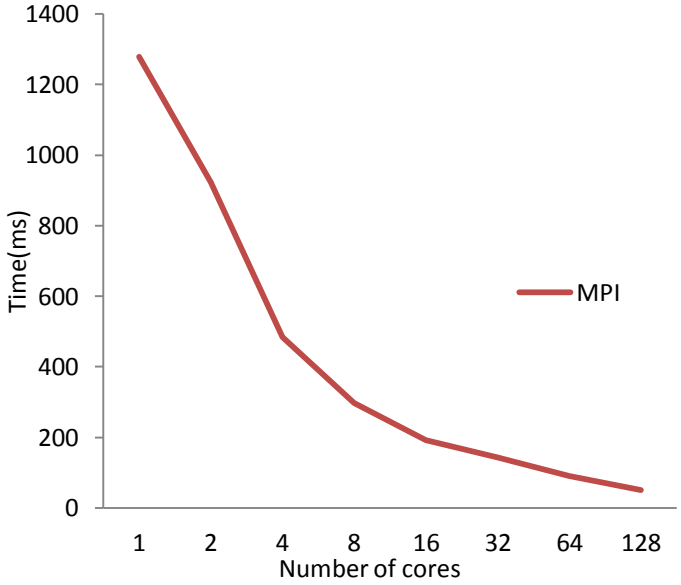


Figure 31



We can see from Figure 31 that now, the application takes advantage of parallelism and the speedup curve goes up and not down as in the previous case. This because in this case there is larger amount of computations involved and even though the synchronization overhead exists, it is overshadowed by the parallel computations. Dijkstra's algorithm has inherent problem of limited explicit parallelism and excessive synchronization. We confirm this as the OpenMP code has several *#pragma omp barrier* which are used for synchronization and which result in a lot of thread waits. Due to this, there is a significant gap between the ideal and parallel speedup which becomes wider as number of cores increase. The performance of MPI and OpenMP is comparable up to 16 cores, with OpenMP performing slightly better than MPI. But after 16 cores, the performance of MPI degrades because at this stage the communication between the processes becomes a bottleneck. Figure 32 shows the execution time taken for the MPI codes on distributed memory with large number of cores.

Figure 32



5.4 1D FFT (Spectral Methods)

As mentioned in Section 4.2.4, FFT is a communication intensive application. So its performance will depend on the communication operations of the two models. We observe from the graph in Figure 33 that MPI and OpenMP both perform well and also scale well as the speedup curve does not flatten anywhere. The performance of OpenMP is little better than that of MPI implying that the communication of threads in OpenMP is faster than the communication of processes. This was expected because threads communicate by sharing variable and safely accessing the variables when needed. Whereas in MPI, the processes communicate by passing messages to each other which involves the issue of latency (Section 1.3)

Figure 33

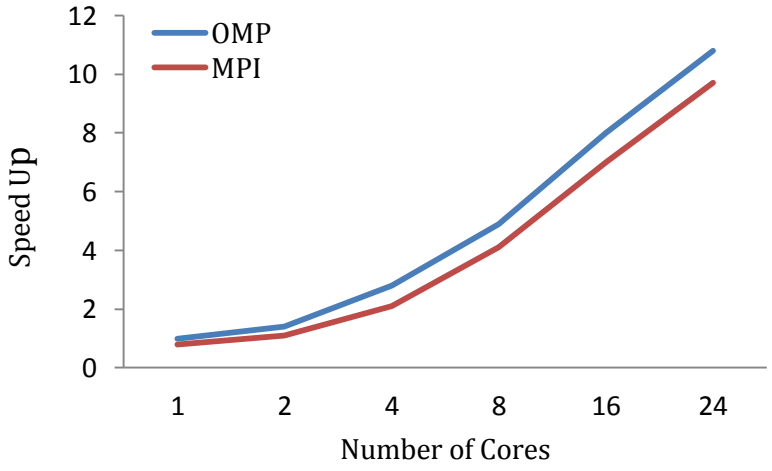
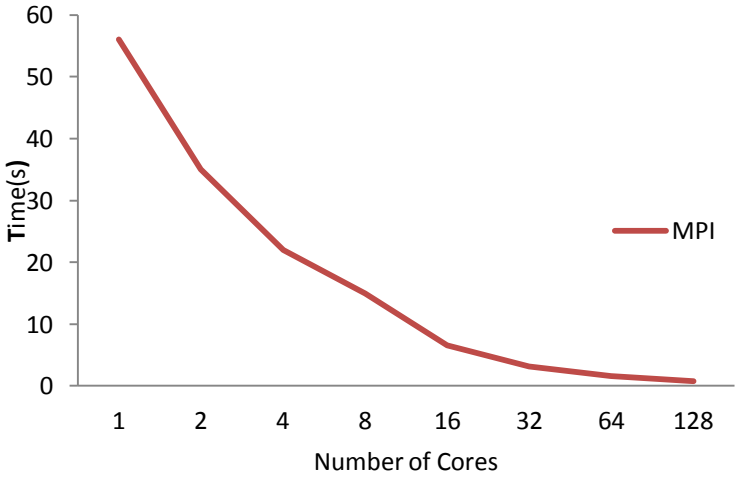


Figure 34 shows the speedup obtained when the MPI implementation is run on distributed memory. We observe that MPI scales well in this case.

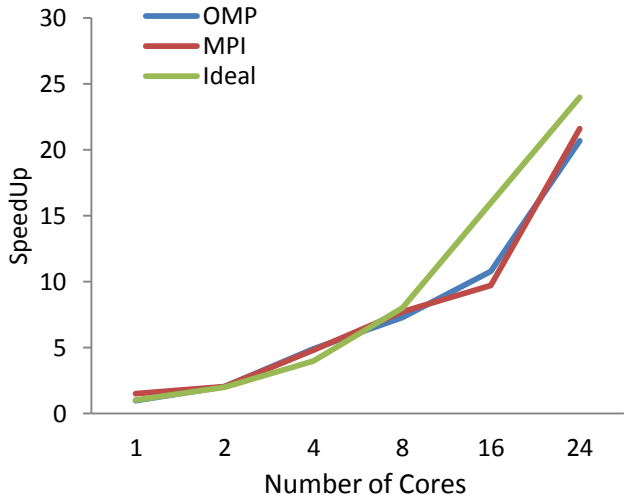
Figure 34



5.5 D2Q9 Lattice Boltzmann (Structured Grid)

The input dataset is of size 300x200. We observe from Figure35 that when number of cores are 4, the speedup obtained for OpenMP is more than the ideal expected speedup, that is, the speedup is super linear which is explained in Figure 22. This can be attributed to better cache utilization using 4 cores. To support our argument we determine the cache performance using *cachegrind.*

Figure 35



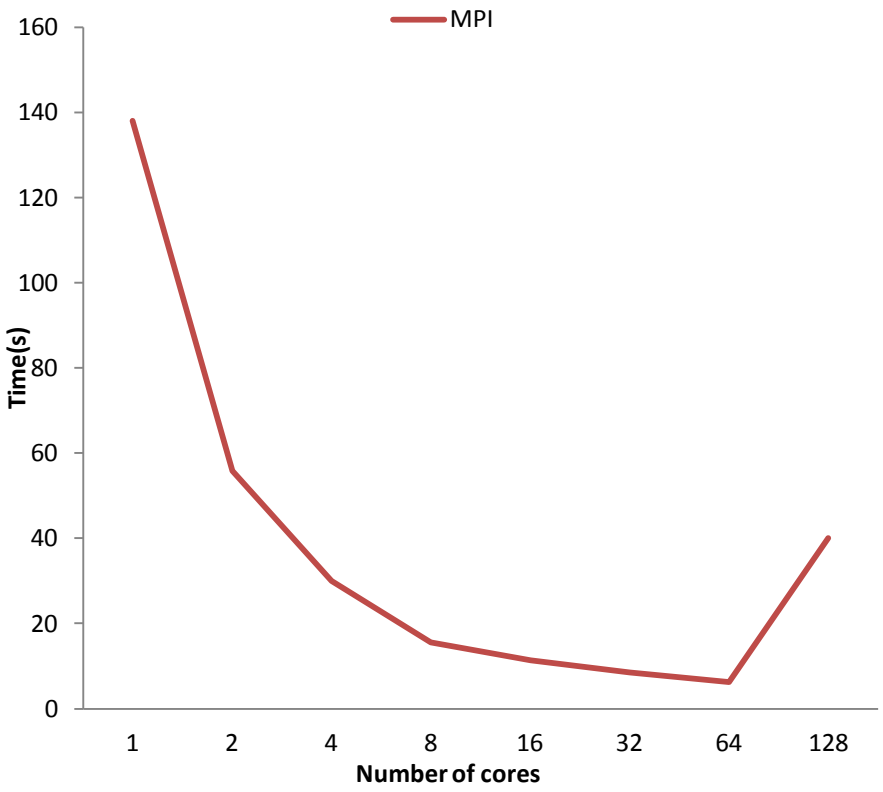
As it can be noted from Table 5, the miss rates of LLd and D1 caches are lower when 4 cores are used. So when 4 cores are used, not only has the parallelism increased but the cache misses are also reduced which eventually results in a super linear speedup.

Table 5 – Cache Performance

	1 Core	**4 Cores**
LLd miss rate	0.98%	0.55%
D1 miss rate	7.9%	7.5%

The performance of MPI is similar to OpenMP. The speedup for the both the models decrease around 16 cores where the curve deviates from the ideal speedup. The result of running the MPI version on the distributed memory architecture is shown in Figure 36.

Figure 36



We observe that up to 8 cores, the time taken to execute decreases proportionately with increase in number of cores. This is because the there is limited communication and sufficient computations per process involved. Thus the processes spend more time computation than on communication. As cores are further increased, the time curve tends to flatten and thus the speedup obtained due to parallelism decreases. This implies that time spent of communication and computation is similar. But at 128 cores, the time taken suddenly increases and becomes more than its predecessors. As the work load is distributed among a very large number of processes, the computations per process is relatively less whereas the inter process communication increases as the number of processes have increased. This leads to an increase in the time taken to execute the application with 128 processes.

Here, if we increase the size of our input data set, there would be no increase in time at 128 cores because then the computations per process will still be significant as due to larger data set, each process will have more data to process. There will definitely be communication bottleneck but at a later stage that 128 cores in case of larger data sets.

5.6 PROGRAMMABILITY

The aim of the project is not only to compare the models on the basis of their performance but also on the basis of their ease-of-use. As mentioned in Section 3.4.2, we measured Programmability in terms of productivity. Productivity is calculated in terms of Lines of Code (LOC) and Development time. Section 4.4 mentions how we measured these two metrics. After calculating the LOC of each program, we calculated the relative LOC that is the LOC with respect to the serial code. We then plot a graph (Figure 37) of the relative LOC of every MPI and OpenMP programs and the corresponding speedup obtained. For each program consider the speedup obtained using 8 cores and so the ideal speedup is 8. We notice that all the plots lie between 1 and 2 values of relative LOC. This was expected because we knew the LOCs for the parallel codes are always more than the LOCs of serial code. Also we note that the plots of MPI are farther from 1 than that of OpenMP, indicating that the MPI programs are comparatively longer.

Figure 37

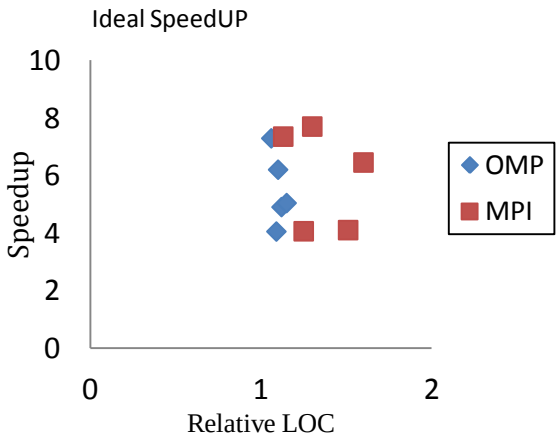


Figure 38

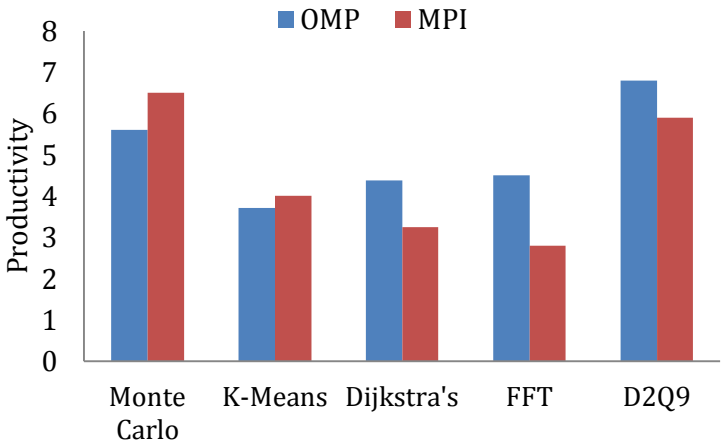


Figure 38 compares the productivity of each OMP and MPI applications. We note that for Monte Carlo simulations and for K-means clustering the productivity of MPI is higher. This implies that even thought the MPI code for these applications are longer than OpenMP code, one should still choose MPI because MPI has far better performance and also the codes are not very long as compared to OpenMP.

We also decided to use our experience of implementing the two applications (Monte Carlo and Lattice Boltzmann) in MPI and OpenMP. We kept track of the approximate number of hours we spent on coding (Table 6) and used this metric to compare the two models.

Table 6 – Approximate Development time effort

Application	**Time (hr) to parallelize**	
	OpenMP	*MPI*
Calculating Pi	2	2
D2Q9 Lattice Boltzmann	8	14

Coding the Monte Carlo application was not very difficult and took the same amount of time for MPI and OpenMP. This is because the application did not require interaction between the units of execution. Thus in case of such applications it is better to use MPI, performance and ease-of-use wise. Parallelizing the lattice Boltzmann application using MPI took longer than time spent on parallelizing using OpenMP. This is because we had to take care of the inter process communications, splitting of data and distributing to each process and so on (Section 4.2.5.2). We have seen in previous section that for Lattice Boltzmann performance of both the models is similar so OpenMP is a more favorable choice for this application. We can discuss only these two applications because the rest were not self-implemented but collected from the web and the development time information for them was not available.

5.7 REVIEW

We discussed and compared the performance of both the models for the five chosen applications. For every application we analyzed the graph showing speedup obtained using OpenMP and MPI. Reasons for performance scale up, drop or saturation were discussed for every case. We showed the comparison of programmability of OpenMP and MPI. Productivity was calculated using LOC metric. We used our coding experience to measure programmability by keeping track of the approximate development time which was used to compare the models for their ease-of-use.

6 DISCUSSION

This chapter gives an overview of all the tasks performed in the project, the problems faced during the Project and how they were handled, the challenging part of the project, experience gained from programming using MPI and OpenMP. The results of comparing the two models are summarized and few critical points observed during the projects which are worth mentioning are visited.

We started the project with investigating several parallel programming paradigms available. We realized that there is a myriad of parallel programming models present today and for a novice user, selecting one of them for parallelizing an application will be a very difficult task. After briefly studying seven programming models in terms of their memory architecture, communication model, we decided to select OpenMP and MPI for the Project.

6.1 REFLECTION OF THE CODING EXPERIENCE

The experience gained from implementing the two applications in both the models has made us realize the issues of designing a parallel program and general points to be considered while implementing a parallel program in any programming model. Following section lists these points.

- **Problem Decomposition**
  After understanding the serial program that is to be parallelized, we need to think on how the problem can be divided into chunks or smaller sub problems which can run concurrently. This is called problem decomposition which can be done in two ways –
  - Data decomposition – The input dataset on which the problem works is divided in to smaller chunks and assigned to the units of execution.
  - Function Decomposition – The entire program is considered as one task and smaller sub tasks are formed which can run in parallel. Each unit of execution runs a different task.

- **Communication**
  While coding the Monte Carlo application we did not need to perform any communications between the units of execution. This was expected because this application was an embarrassingly parallel one. When we started implementing the D2Q9 lattice Boltzmann program, we realized that all applications are not as simple as the previous one and require the units of execution to communicate with each other to share data. Also, after studying the codes of other applications, taken from the web we noticed that the communication needed depends on the application and is different for different kinds of application. These communications are a point of concern because they come with a cost as resources which could be used for computations are now to be used for sending or receiving the data. We noted that communication was more of an issue for MPI than for OpenMP. As MPI is a message passing model, the communications are explicit and visible to the used. During the MPI implementation we came across the point to point communication and collective communication options. One must consider the factors of performance and accuracy before choosing a type of communication.

- **Data Dependencies**
  The most important point to consider in parallel programming to ensure correctness of the result is handling the data dependencies. Suppose we a program that runs few sub tasks and if the order in which the tasks are run is important to get correct output then we say that the there is data dependency. To handle these dependencies, first we need to understand where they exist which can be determined by carefully studying the serial program. For example while studying the serial code for D2Q9 LBM, we noticed that it has dependency in one of the *for* loops and also there is data dependency outside the *for* loop, between two functions
- **Synchronization**
  The data dependencies can be handled using synchronization mechanism which implies that a unit of execution must not proceed until all other units have reached to the same point in the program. For both the models synchronization is achieved using *barriers.* We observed that synchronization is more of an issue in case of OpenMP than in MPI. This is because OpenMP is shared memory model due to which access to the critical region (globally shred address space) should also be synchronized.
- **Load Balancing**
  We discussed the issue of load balancing in Section 1.6.1. During the practical implementation, we learnt how to distribute the tasks equally among the units of execution and its importance, performance wise. For example, when there was a *for* loop, we divided the number of iterations and assigned equal number of iterations to each work unit.

Thus from the coding experience we learnt various important factors to be considered in parallel programming and how to correctly handle them. This knowledge is not specific to a parallel paradigm but is general enough and can be used for other paradigms as well.

6.2 PERFORMANCE COMPARISON SUMMARY

In chapter 5 we analyzed the results of running each of the five applications in MPI and OpenMP and compared the speedup obtained in both the cases. Following section comments on the results and informs which model is better for which application in terms of performance.

- *Monte Carlo* – The performance of both the models was comparable but MPI was little faster than OpenMP when number of cores increased to 8 and more. The speedups in both the cases were close to the ideal speedup.

- *K-Means* – For this application, MPI was a clear winner. The main reason of OpenMP's poor performance was data locality. This application requires a lot of memory accesses and there were a lot of cache misses in case of OpenMP. To support the argument, we measure the cache misses using cachegrind. MPI performed well because the application is not communication intensive and the communications were overlapped with computations which saved time.

- *Dijkstra's Algorithm* – This is communication intensive application and also involves random memory lookups. The performance of OpenMP is better than MPI because communication becomes a bottleneck and it cannot be efficiently overlapped with computations as there are not many computations involved in these applications. Although OpenMP outperforms MPI, its performance is far from the linearity.

- *FFT* – Performance of OpenMP is little better than MPI and both models scale well in case of this application.

- *Lattice Boltzmann* – This is a structured grid application which has long strided memory access. Performance of MPI and OpenMP is very similar up to 8 cores after which the speedup for OpenMP slows down comparatively due to increased synchronization overhead.

FEW CRITICAL COMMENTS ON PERFORMANCE

- Speedup obtained in case of all applications for both the models is less than that obtained for the Monte Carlo application. This was expected because it is an embarrassingly parallel program as we mentioned earlier.

- It was noticed that performance does not simply increase with increase in number of cores as there is cost of start up, communication and synchronization involved. There can be times when the parallel code runs slower than the serial one. For example the Dijkstra's algorithm for 100 graph nodes ran slower than the serial code (for both the models).

- When we executed the application for larger input datasets, we achieved higher scalability that is parallelism was more beneficial for larger datasets than for smaller ones. This ascertains Gustafson's Law (Section 3.4.1.1.2)

- To observe the difference in inter process communication in MPI on shared memory and distributed memory, we ran a small test. We measured the communication overhead that is time spent to send a 0 byte message between two MPI processes on the same node and on different nodes. We observed that time required for the processes on the same node was around 4 micro seconds and that for processes on different nodes was around 7 micro seconds. This was expected because for processes on different nodes, the message has to be sent over the network which is takes relatively longer.

- Performance can be optimized by improving the utilization of cache but this was not done in this project because that makes the comparison hardware dependent and specific to the particular memory or cache architecture. OpenMP usually suffers from scalability whereas MPI comes with the cost of communication.

6.3 PROGRAMMABILITY COMPARISON SUMMARY

As mentioned earlier, quantifying and measuring programmability is more difficult than measuring performance. After going through several previous works, we decided upon the metrics to be used to measure programmability. We measured productivity using LOC as the metric for measuring the parallel effort. Also, the time taken to implement the two applications in MPI and OpenMP was recorded (approximately). In terms of productivity, MPI is proved better for Monte Carlo and K-Means clustering. Whereas OpenMP was better for FFT and graph traversal. For structured grid application (Lattice Boltzmann model) the development time for MPI was significantly more than that for OpenMP. Few more experiments can be performed to measure the productivity. For example classroom experiments can be carried out where students can be given a serial application and asked to parallelize using different models and log of the time taken by each student can be maintained. During this project we observed that it is very important to consider the human effort required to parallelize an application using a particular model. Parallel programming is not very trivial as there additional issues to be considered and so it is imperative to measure the programmer's effort involved. We noticed that there is no hard and fast approach for comparing the parallel models in terms of programmability. This is still a very open issue and requires more consideration. Few general result of comparing the programmability were

- As we measure the LOC for every code we determined that OpenMP and MPI codes are always longer than the serial. Moreover, MPI codes are always longer than OpenMP.
- From Figure 35 and Table 6 we observed that a parallel code (OpenMP or MPI) will require more development effort than the corresponding serial code.
- Even though the LOC or development time for OpenMP is always lower than MPI, it does not imply that its productivity will always be higher as speedup is also considered in measuring the productivity.

6.4 COMMENTS ON PORTABILITY

Portability is the ability of using the same software on different environments. As we are comparing the programming models, which are simply two different software, we must examine their portability. Portability implies removing hardware dependency [4]. We can make the code run faster if we take the system architecture into consideration and so there is always a tradeoff between portability and efficiency. We wanted to make our comparison generic and so we did not focus on the hardware architecture, that is, we did not optimize our codes to give better performance on the hardware we were running on. Although all our codes were in C language, we investigated the programming languages supported by the two models. Table7 shows the languages each model supports indicating their portability across different languages. We note that both the models support most of the popular programming languages. Thus in terms of portability one model is not preferred over the other as both are equally portable.

Table 7

Model	**Languages Supported**
OpenMP	Fortran, C, C++.
MPI	Fortran, C, C++, Python

6.5 FEW PROBLEMS FACED DURING THE PROJECT

- It should be mentioned that while choosing the parallel paradigms to be compared, we also intended to choose OpenCL (Section 2.2.7) which is used to write program across heterogeneous architectures (CPUs + GPUs). We thought it would be very interesting to compare MPI, OpenMP and OpenCL (all three meant for different memory architectures). So, while implementing the first application, Monte Carlo, we implemented it in OpenCL as well. But eventually, when we were collecting codes form the web, we could not find the same application coded in OpenCL, MPI and OpenMP. OpenCL codes used completely different approach and were not comparable to OpenMP and MPI. This was not suitable for a fair comparison. Also, with the limited time for the project it was not possible to convert the chosen MPI or OpenMP codes to OpenCL. Thus we decided to drop out OpenCL but we definitely intent to include this as one of our future works.

- The major problem faced was gaining access the resources used to be used. The BlueCrystal clusters, on which the programs were run, were down most of the time. The sudden “crashing” of the machine caused a lot interruption in the project. Also, even when the machine was working, there were problems with the infini-bands connecting the clusters or some configuration faults.

- In this project we have compared the performance for five applications. Although five applications were sufficient, we had also decided the sixth application to be parallelized. Implementation of parallel AES block encryption/decryption algorithm belonging to the Combinational Logic dwarf (Section 3.1) was already begun. Unfortunately we did not have sufficient time left and we had to drop the plan.

7 CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

In this project we briefly surveyed seven parallel programming models and selected OpenMP – shared memory model and MPI – message passing model for comparison. The Berkeley's dwarf taxonomy was used to select applications where each dwarf captures a unique computational and communication pattern which is common to a class of applications. Project focuses on a subset of dwarfs including *Dense Linear Algebra, Map Reduce, Spectral Methods, Graph Traversal* and *Structured Grids*. Thus we ensured that the selected applications are significantly diverse, aiming to cover larger domain of applications to be able to draw broader conclusions. We identified some suitable metrics to evaluate the parallel models in terms of performance and programmability.

We presented a comparison of MPI and OpenMP on shared memory architecture. We observed that both MPI and OpenMP show comparable performance and there is no clear winner. Thus we cannot say that one model is better than the other in all the situations. Up to four cores the performance of both MPI and OpenMP is very similar and relatively closer to the linear speedup. As parallelism is increased, depending on the application, either OpenMP suffers due to excessive thread management and synchronization overhead or MPI has the problem of excessive communication becoming a bottleneck. We also compared the two models for their ease-of-use in terms of productivity. From our coding experience we conclude that the main advantage of OpenMP is its simplicity and incremental approach towards parallelism. By running the MPI implementation on distributed architecture we conclude that it is relatively more scalable across large number of processors. For some cases, the speedup at 128 cores was not as high as expected which was due to the network contention which increases with number of cores and is a challenging issue which must be handled. We also experienced that increasing the size of the data set increases the scalability of the application, that is, parallelism becomes more beneficial thus verifying Gustafson's Law. Thus we have successfully analyzed the pros and cons of both the models in this project. The portability of the two models was determined and we concluded that both the models are equally portable.

One of the objectives of this project was to help a developer to make an informed choice of a parallel programming model. Before actually choosing a model, a user should first study the application to be parallelized and identify its type by determining the dwarf it comes under. After examining the communication and computation patterns present in the particular dwarf and the amount of effort the user wishes to put, a choice between MPI and OpenMP can be made using this work. For example if the user wishes to put relatively less efforts in parallelizing a structured grid application, then OpenMP should be selected because even though MPI performs slightly better than OpenMP in this case, the productivity of OpenMP is significantly higher than MPI.

A suggestion for future programming models is that efforts must be made to integrate the best of both the worlds (shared memory and message passing models). The future models should be more human-centric, that is, they must be developed in keeping in mind not only the performance of the model but also its ease-of-use.

7.2 FUTURE WORK

In this project, we have only chosen a subset of dwarfs so an obvious extension to the project is to complete the list of thirteen dwarfs by selecting a representative application from the remaining dwarfs, implementing it in the two models and comparing them using the metrics identified in this project.
We mentioned in Section 6.5 that we had also chosen OpenCL for comparison but the plan had to be dropped due unavailable resources. So the future work will also involve porting all the applications in OpenCL and comparing the results with that of MPI and OpenMP.

Due to the limitations of hardware available, we could compare the model for 24 cores. Including even larger datasets and running the applications on even larger number of cores will be an interesting task to perform.

In this project, we used personal coding experience to evaluate the productivity if the models using the development time effort. We can also carry several classroom experiments in which students are given textbook problems to parallelize using different models and their efforts are recorded. In our case we recorded the development time manually and simply in a log book but this can be automated by installing some time logging software on our machines.

As determined throughout this project, there is no clear winner among the two models. As discussed earlier, the reason for this is different model is preferred for different situations. Also a person's view or preference guides the results obtained for the programmability metrics. We hope this work will provide an insight to anyone learning about parallel programming or anyone contemplating the choice between shard memory models and message passing models. The larger aim is to bring parallel programming to masses which can be done by continuing to work on projects like this.

BIBLIOGRAPHY

1. Asanovic, K., Catanzaro, B. C., Patterson, D. A., & Yelick, K. A. (2006). The Landscape of Parallel Computing Research : A View from Berkeley.
2. Barney, Blaise. "Introduction to Parallel Computing". Lawrence Livermore National Laboratory.
3. Burgstaller, B., & Scholz, B. (2011). Performance of Parallel Programs, 1-63.
4. Mattson,Sanders, Massingill. (2004) Patterns for Parallel Programming
5. Kasim, H., March, V., Zhang, R., & See, S. (2008). Survey on Parallel Programming Model.
6. Barney, Blaise. "POSIX Threads Programming". Lawrence Livermore National Laboratory.
7. Architecture, O., Board, R., Architecture, O., Board, R., Architecture, O., & Board, R. (2002). OpenMP C and C ++ Application Program Interface
8. Barney, Blaise. "Message Passing Interface (MPI)". Lawrence Livermore National Laboratory
9. Yelick, K. (2006). UPC Outline CS 267 Unified Parallel C (UPC) UPC Execution Model, 1-11
10. Steele, G., & Maessen, J.-willem. (2006). Fortress Programming Language Tutorial.
11. Rosenberg, O. (2011). OpenCL Overview, (November).
12. Zeller, C. (n.d.). Tutorial CUDA
13. Jin, H., Frumkin, M., Yan, J., & Field, M. (1999). The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, (October).
14. Fraguela, B. B., & Mouri, J. C. (2009). Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures, 174-184.
15. Dickson, N. G., & Hamze, F. (n.d.). A Performance Comparison of CUDA and OpenCL, (1).
16. Shen, J., Fang, J., Varbanescu, A. L., & Sips, H. (n.d.). OpenCL vs . OpenMP : A Programmability Debate, 1-17.
17. Andersch, M., & Juurlink, B. (n.d.). A Benchmark Suite for Evaluating Parallel Programming Models Introduction and Preliminary Results
18. Szafron, D. (1994). University of Alberta An Experiment to Measure the Usability of Parallel, (May).
19. Funk, A., Basili, V., Hochstein, L., & Kepner, J. (2005). Application of a development time productivity metric to parallel software development. *Proceedings of the second international workshop on Software engineering for high performance computing system applications - SE-HPCS '05*, 8. New York, New York, USA: ACM Press. doi:10.1145/1145319.1145323
20. Amdahl. G. (1967). Validity of single processor approach to achieveing large scale computing capabilities.
21. Liao,W.(2005). *Parallel K-Means data clustering.* Available: http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html. Last accessed 15th Aug 2012.
22. Hassan Jafri. (2006). *1D FFT.* Available: http://charm.cs.uiuc.edu/cs498lvk/projects/. Last accessed 20th Aug 2012.
23. Engler, S. T. (n.d.). Benchmarking the 2D Lattice Boltzmann BGK Model, 1-10.
24. Nicholas, N. (2004). Dynamic Binary Analysis and Instrumentation or Building Tools is Easy,50-60.

APPENDIX

RAW READINGS

1. Monte Carlo application (Time in s)

Cores	2	4	8	16	24
OMP	2.77	1.48	1.10	0.56	0.325
	2.51	1.56	1.04	0.43	0.36
	2.87	1.32	1.07	0.6	0.28
	2.64	1.36	1.20	0.52	0.32
	2.83	1.52	1.12	0.48	0.36
MPI	2.453	1.52	0.945	0.56	0.278
	2.66	1.48	0.90	0.54	0.30
	2.54	1.56	0.98	0.58	0.25
	2.36	1.58	0.96	0.60	0.28
	2.31	1.48	0.88	0.21	0.30

2. K-Means Clustering (Time in s)

Cores	2	4	8	16	24
OMP	19.99	11.49	6.59	4.44	6.22
	18.23	9.9	6.21	4.89	5.14
	16.8	12.4	5.89	3.65	5.03
	22.97	15.6	4.23	4.99	5.10
	24.4	8.99	7.21	5.21	4.83
MPI	23.05	11.455	5.58	2.79	1.48
	21.79	10.44	5.70	2.55	1.63
	25.66	13.21	5.23	2.31	1.88
	24.52	11.72	4.91	2.67	1.23
	22.59	12.90	6.2	2.92	1.56

3. Dijkstra's Algorithm (Time in ms)

Cores	2	4	8	16	24
OMP	713	502	174	160	79
	688	494	185	176	97
	702	482	180	163.5	87.7
	710	490	196	178	90
	696	506	199	159	82
MPI	941	932	519	360	180
	966	938	513	368	169
	953	943	524	354	175
	948	928	507	363	195
	959	930	524	370	188

4. 1D FFT (Time in s)

Cores	2	4	8	16	24
OMP	8.9	5.7	3.122	2.6	1.8
	8.6	5.5	3.05	2.5	1.9
	8.4	5.1	3.71	2.8	1.4
	9.2	6.0	2.93	2.53	1.56
	8.56	5.32	3.1	2.48	1.89
MPI	23.05	11.455	5.58	2.79	1.48
	22.09	12.31	5.03	2.56	1.59
	24.58	10.98	6.9	3.01	1.12
	23.11	11.40	6.25	2.15	1.90
	21.90	12.01	4.81	2.31	1.57

5.Lattice Boltzmann

Cores	2	4	8	16	24
OMP	126	58	30	25	13
	129	52	36	29	14
	135	69	29	21	11
	115	43	40	20	13
	124	55	37	17	12
MPI	110	56	28	20	13
	119	61	28	24	11
	105	60	27	21	12
	113	47	25	19	14
	121	50	29	20	15

6. Lines of Code

	Monte Carlo	**K-Means**	**Dijkstra's**	**1D FFT**	**D2Q9 LBM**
Serial	60	296		146	400
OpenMP	66	397		166	427
MPI	68	469		221	525

MPI CODE FOR LATTICE BOLTZMANN

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<mpi.h>
#include<omp.h>

#define NSPEEDS        9
#define PARAMFILE      "input.params"
#define OBSTACLEFILE   "obstacles_300x200.dat"
#define FINALSTATEFILE  "final_state.dat"
#define AVVELSFILE     "av_vels.dat"

/* struct to hold the parameter values */
typedef struct {
  int    nx;          /* no. of cells in y-deirection */
  int    ny;          /* no. of cells in x-direction */
  int    maxIters;     /* no. of iterations */
  int    reynolds_dim;  /* dimension for Reynolds number */
  double density;      /* density per link */
  double accel;        /* density redistribution */
  double omega;        /* relaxation parameter */

} t_param;

/* struct to hold the 'speed' values */
typedef struct {
  double speeds[NSPEEDS];
} t_speed;

enum boolean { FALSE, TRUE };

/*
** function prototypes
*/

/* load params, allocate memory, load obstacles & initialise fluid particle densities */
int initialise(t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
               int** obstacles_ptr, double** av_vels_ptr,t_speed** tmp_cells_ptr1,t_speed** tmp_cells_ptr2);

/*
** The main calculation methods.
** timestep calls, in order, the functions:
** accelerate_flow(), propagate(), rebound() & collision()
*/
int timestep(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles, int rank);
int accelerate_flow(const t_param params, t_speed* cells, int* obstacles,int rank);
int propagate(const t_param params, t_speed* cells, t_speed* tmp_cells,int *obstacles, int rank);
int rebound(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles);
double collision(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles, int rank, t_speed* local_tmp1,
t_speed* local_tmp2);
```

```
int write_values(const t_param params, t_speed* cells, int* obstacles, double* av_vels);

/* finalise, including freeing up allocated memory */
int finalise(const t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
                    int** obstacles_ptr, double** av_vels_ptr);

/* Sum all the densities in the grid.
** The total should remain constant from one timestep to the next. */
double total_density(const t_param params, t_speed* cells);
//float total_density(const t_param params, t_speed* cells);
/* compute average velocity */
double av_velocity(const t_param params, t_speed* cells, int* obstacles);
//float av_velocity(const t_param params, t_speed* cells, int* obstacles);
/* calculate Reynolds number */
double calc_reynolds(const t_param params, t_speed* cells, int* obstacles);
//float calc_reynolds(const t_param params, t_speed* cells, int* obstacles);

/* utility functions */
void die(const char* message, const int line, const char *file);

/*
** main program:
** initialise, timestep loop, finalise
*/
int main(int argc, char* argv[])
{

  t_param  params;          /* struct to hold parameter values */
  t_speed* cells     = NULL;  /* grid containing fluid densities */
  t_speed* tmp_cells = NULL;  /* scratch space */
  t_speed* local_tmp1 = NULL;  /* scratch space */
 t_speed* local_tmp2 = NULL;  /* scratch space */
  int*     obstacles = NULL;  /* grid indicating which cells are blocked */
int dest1,dest2;
  double*  av_vels   = NULL;  /* a record of the av. velocity computed for each timestep */
 // float*  av_vels   = NULL;
  int      ii,jj,zz;             /* generic counter */
double t11,t22,t33,t44;       /* check points for reporting processor time used */
  int rank;              /* rank of process */
  int size;              /* number of processes started */
int tot_cells = 0;
int recv_cells=0;
double tot_u_x,recv_u_x=0.0;
zz =0;
 MPI_Status status;
int start, stop;
  int count;
  int cnt =0;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

/* initialise our data structures and load values from file */
```

```
initialise(&params, &cells, &tmp_cells, &obstacles, &av_vels,&local_tmp1,&local_tmp2);



  count = params.ny/size;
  start = rank*count;
if(rank == size-1)
stop = params.ny;
else
  stop = start + count;

dest1 = rank-1;
dest2 = (rank +1)%size;
if(rank == 0)
dest1 = size-1;

if(rank == 0)
start = params.ny;
if(rank == size-1)
stop = 0;
t11 = MPI_Wtime();
for (ii=0;ii<params.maxIters;ii++) {

tot_cells =  propagate(params,cells,tmp_cells,obstacles,rank);

if(rank%2==0)
{


MPI_Sendrecv(&tmp_cells[stop*params.nx], sizeof(t_speed)*params.nx, MPI_BYTE,dest2, rank, &local_tmp1[0],
sizeof(t_speed)*params.nx,MPI_BYTE, dest2, dest2, MPI_COMM_WORLD,&status);

MPI_Sendrecv(&tmp_cells[(start)*params.nx- params.nx], sizeof(t_speed)*params.nx, MPI_BYTE,dest1, rank,
&local_tmp2[0], sizeof(t_speed)*params.nx,MPI_BYTE, dest1, dest1,MPI_COMM_WORLD,&status);

}
else
{

MPI_Sendrecv(&tmp_cells[(start)*params.nx- params.nx], sizeof(t_speed)*params.nx, MPI_BYTE,dest1, rank,
&local_tmp2[0], sizeof(t_speed)*params.nx,MPI_BYTE,dest1, dest1,MPI_COMM_WORLD,&status);

MPI_Sendrecv(&tmp_cells[stop*params.nx], sizeof(t_speed)*params.nx, MPI_BYTE,dest2, rank, &local_tmp1[0],
sizeof(t_speed)*params.nx,MPI_BYTE, dest2, dest2, MPI_COMM_WORLD,&status);

}

if(rank!=0)
zz = ((params.nx*params.ny)/size)*rank ;

if(rank==0)
zz = (params.nx*params.ny);
```

```
tot_u_x = collision(params,cells,tmp_cells,obstacles,rank,local_tmp1,local_tmp2);



if(rank!=0)
{
        MPI_Send(&tot_u_x, sizeof(double), MPI_BYTE,0, rank, MPI_COMM_WORLD);
        MPI_Send(&tot_cells, sizeof(int), MPI_BYTE,0, rank, MPI_COMM_WORLD);
}

if(rank == 0)
{

 recv_cells = tot_cells;
 recv_u_x = tot_u_x;

 for(jj=1;jj<size;jj++)
 {
            MPI_Recv(&tot_u_x, sizeof(double), MPI_BYTE,jj,jj, MPI_COMM_WORLD,&status);
     MPI_Recv(&tot_cells, sizeof(int), MPI_BYTE,jj,jj, MPI_COMM_WORLD,&status);

    recv_u_x += tot_u_x;
    recv_cells+=tot_cells;
}
//if(ii==params.maxIters-1)
//printf("\ncells %d\n",recv_cells);
 av_vels[ii] = recv_u_x/(double)recv_cells;

 }
}

MPI_Barrier(MPI_COMM_WORLD);
t22 = MPI_Wtime();

if(rank==0)
start = 0;
if(rank==size-1)
stop = params.ny;

//printf("\nproc %d send val %d \n",rank,(params.nx*(params.ny/size))*rank);
if(rank!=0 )
MPI_Send(&cells[(params.nx*(params.ny/size))*rank], sizeof(t_speed) * (stop*params.nx-start*params.nx), MPI_BYTE, 0,
rank, MPI_COMM_WORLD);


if(rank==0)
{  /* write final values and free memory */

  for(jj=1;jj<size;jj++)
 {
```

```
if(jj == size-1)
MPI_Recv(&cells[(params.nx*(params.ny/size))*jj], sizeof(t_speed) * params.nx*params.ny -
((params.nx*(params.ny/size))*jj) , MPI_BYTE, jj, jj, MPI_COMM_WORLD,&status);
else
MPI_Recv(&cells[(params.nx*(params.ny/size))*jj],sizeof(t_speed) *
(params.ny/size)*params.nx,MPI_BYTE,jj,jj,MPI_COMM_WORLD,&status);

}

  printf("\n==done==\n");
  printf("Reynolds number:\t%.12E\n",calc_reynolds(params,cells,obstacles));

  printf("Elapsed time:\t\t %lf (s)\n",t22-t11);
  write_values(params,cells,obstacles,av_vels);
  finalise(&params, &cells, &tmp_cells, &obstacles, &av_vels);

}

MPI_Finalize();

  return EXIT_SUCCESS;
}


int propagate(const t_param params, t_speed* cells, t_speed* tmp_cells, int * obstacles, int rank)
{



int ii,jj,zz;          /* generic counters */
  int x_e,x_w,y_n,y_s;  /* indices of neighbouring cells */
 double w1,w2;  /* weighting factors */

int size;
int start, stop;
  int count;
  int cnt =0;

MPI_Comm_size( MPI_COMM_WORLD, &size );
count = params.ny/size;
start = rank*count;
if(rank == size-1)
stop = params.ny;
else
  stop = start + count;

w1 = params.density * params.accel / 9.0;
w2 = params.density * params.accel / 36.0;

  /* loop over _all_ cells */

for(ii=start;ii<stop;ii++) {
```

```
zz = params.nx*ii;

//if(rank == size-1)
//zz2 =((params.nx*params.ny)/size)*rank ;
 if( !obstacles[zz] &&
                (cells[zz].speeds[3] - w1) > 0.0 &&
                (cells[zz].speeds[6] - w2) > 0.0 &&
                (cells[zz].speeds[7] - w2) > 0.0 ) {
    /* increase 'east-side' densities */
    cells[zz].speeds[1] += w1;
    cells[zz].speeds[5] += w2;
    cells[zz].speeds[8] += w2;
    /* decrease 'west-side' densities */
    cells[zz].speeds[3] -= w1;
    cells[zz].speeds[6] -= w2;
    cells[zz].speeds[7] -= w2;
   }


 for(jj=0;jj<params.nx;jj++) {
                zz = ii*params.nx + jj;
    /* determine indices of axis-direction neighbours
    ** respecting periodic boundary conditions (wrap around) */
    y_n = (ii + 1) % params.ny;
    x_e = (jj + 1) % params.nx;
    y_s = (ii == 0) ? (ii + params.ny - 1) : (ii - 1);
    x_w = (jj == 0) ? (jj + params.nx - 1) : (jj - 1);
    /* propagate densities to neighbouring cells, following
    ** appropriate directions of travel and writing into
    ** scratch space grid */
    if(!obstacles[zz])
      cnt++;

    tmp_cells[zz].speeds[0]  = cells[zz].speeds[0]; /* central cell,zz.speeds[4]recv */
                                                    /* no movement   */
    tmp_cells[ii *params.nx + x_e].speeds[1] = cells[zz].speeds[1]; /* east .speeds[8]recv*/
    tmp_cells[y_n*params.nx + jj].speeds[2]  = cells[zz].speeds[2]; /* north..out-high */
    tmp_cells[ii *params.nx + x_w].speeds[3] = cells[zz].speeds[3]; /* west ..speed[7]recv*/
    tmp_cells[y_s*params.nx + jj].speeds[4]  = cells[zz].speeds[4]; /* south...out-low */
    tmp_cells[y_n*params.nx + x_e].speeds[5] = cells[zz].speeds[5]; /* north-east..out-high */
    tmp_cells[y_n*params.nx + x_w].speeds[6] = cells[zz].speeds[6]; /* north-west..out-high */
    tmp_cells[y_s*params.nx + x_w].speeds[7] = cells[zz].speeds[7]; /* south-west...out-low */
    tmp_cells[y_s*params.nx + x_e].speeds[8] = cells[zz].speeds[8]; /* south-east ...out-low*/

 }
}
  return cnt;
}



double collision(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles, int rank, t_speed*
local_tmp1,t_speed* local_tmp2)
```

```
{
  int size;
  int ii,jj,kk,zz;              /* generic counters */
  const double c_sq = 1.0/3.0;  /* square of speed of sound */
  const double w0 = 4.0/9.0;    /* weighting factor */
  const double w1 = 1.0/9.0;    /* weighting factor */
  const double w2 = 1.0/36.0;   /* weighting factor */
  double u_x,u_y;            /* av. velocities in x and y directions */
  double tot_u_x = 0.0;
  double u[NSPEEDS];          /* directional velocities */
  double d_equ[NSPEEDS];       /* equilibrium densities */
  double u_sq;                /* squared velocity */
  double local_density,local_density1;        /* sum of densities in a particular cell */
int y_n,y_s,x_w,x_e;
int start, stop;
  int count;
    MPI_Comm_size( MPI_COMM_WORLD, &size );
  count = params.ny/size;

  start = rank*count;
 if(rank == size-1)
stop = params.ny;
else
  stop = start + count;
  /* loop over the cells in the grid
  ** NB the collision step is called after
  ** theii*params.nx + jj
 propagate step and so values of interest
  ** are in the scratch-space grid */

//  for(ii=start;ii<stop;ii++) {



     for(jj=0;jj<params.nx;jj++) {

 ii = start;
      zz = ii*params.nx + jj;

                /* don't consider occupied cells */
    y_n = (ii + 1) % params.ny;
     x_e = (jj + 1) % params.nx;
     y_s = (ii == 0) ? (ii + params.ny - 1) : (ii - 1);
     x_w = (jj == 0) ? (jj + params.nx - 1) : (jj - 1);

tmp_cells[zz].speeds[2] = local_tmp2[jj].speeds[2];
tmp_cells[ii *params.nx + x_w].speeds[6] = local_tmp2[ x_w].speeds[6];
tmp_cells[ii *params.nx + x_e].speeds[5] = local_tmp2[ x_e].speeds[5];


ii = stop-1;
zz = ii*params.nx + jj;
```

```
                /* don't consider occupied cells */
    y_n = (ii + 1) % params.ny;
     x_e = (jj + 1) % params.nx;
     y_s = (ii == 0) ? (ii + params.ny - 1) : (ii - 1);
     x_w = (jj == 0) ? (jj + params.nx - 1) : (jj - 1);
tmp_cells[zz].speeds[4] = local_tmp1[jj].speeds[4];
tmp_cells[ii *params.nx + x_w].speeds[7] = local_tmp1[ x_w].speeds[7];
tmp_cells[ii *params.nx + x_e].speeds[8] = local_tmp1[ x_e].speeds[8];

}



  for(ii=start;ii<stop;ii++) {


     for(jj=0;jj<params.nx;jj++) {

      zz = ii*params.nx + jj;
if(obstacles[zz]) {
                /* called after propagate, so taking values from scratch space
                ** mirroring, and writing into main grid */
                cells[zz].speeds[1] = tmp_cells[zz].speeds[3];
                cells[zz].speeds[3] = tmp_cells[zz].speeds[1];
       cells[zz].speeds[4] = tmp_cells[zz].speeds[2];
                cells[zz].speeds[7] = tmp_cells[zz].speeds[5];
                cells[zz].speeds[8] = tmp_cells[zz].speeds[6];
                cells[zz].speeds[5] = tmp_cells[zz].speeds[7];
                cells[zz].speeds[6] = tmp_cells[zz].speeds[8];
       cells[zz].speeds[2] = tmp_cells[zz].speeds[4];

 }



     if(!obstacles[zz]) {
                /* compute local density total */
                local_density1 = 0.0;
        local_density = 0.0;
                for(kk=0;kk<NSPEEDS;kk++) {
                  local_density += tmp_cells[zz].speeds[kk];
                }

                /* compute x velocity component */

                u_x = (tmp_cells[zz].speeds[1] +
                      tmp_cells[zz].speeds[5] +
                      tmp_cells[zz].speeds[8]
                      - (tmp_cells[zz].speeds[3] +
                                tmp_cells[zz].speeds[6] +
                                tmp_cells[zz].speeds[7]))
                  / local_density;
```

```
//      tot_u_x = tot_u_x + u_x;
            /* compute y velocity component */
            u_y = (tmp_cells[zz].speeds[2] +
                  tmp_cells[zz].speeds[5] +
                  tmp_cells[zz].speeds[6]
                  - (tmp_cells[zz].speeds[4] +
                              tmp_cells[zz].speeds[7] +
                              tmp_cells[zz].speeds[8]))
                   / local_density;
            /* velocity squared */
            u_sq = u_x * u_x + u_y * u_y;

            /* directional velocity components */
            u[1] =   u_x;       /* east */
            u[2] =       u_y;  /* north */
            u[3] = - u_x;       /* west */
            u[4] =     - u_y;  /* south */
            u[5] =   u_x + u_y;  /* north-east */
            u[6] = - u_x + u_y;  /* north-west */
            u[7] = - u_x - u_y;  /* south-west */
            u[8] =   u_x - u_y;  /* south-east */
            /* equilibrium densities */
            /* zero velocity density: weight w0 */
            d_equ[0] = w0 * local_density * (1.0 - u_sq / (2.0 * c_sq));
            /* axis speeds: weight w1 */


            d_equ[1] = w1 * local_density * (1.0 + u[1] / c_sq
                                                + (u[1] * u[1]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[2] = w1 * local_density * (1.0 + u[2] / c_sq
                                                + (u[2] * u[2]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[3] = w1 * local_density * (1.0 + u[3] / c_sq
                                                + (u[3] * u[3]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[4] = w1 * local_density * (1.0 + u[4] / c_sq
                                                + (u[4] * u[4]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            // diagonal speeds: weight w2
            d_equ[5] = w2 * local_density * (1.0 + u[5] / c_sq
                                                + (u[5] * u[5]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[6] = w2 * local_density * (1.0 + u[6] / c_sq
                                                + (u[6] * u[6]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[7] = w2 * local_density * (1.0 + u[7] / c_sq
                                                + (u[7] * u[7]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
            d_equ[8] = w2 * local_density * (1.0 + u[8] / c_sq
                                                + (u[8] * u[8]) / (2.0 * c_sq * c_sq)
                                                - u_sq / (2.0 * c_sq));
```

```
            /* relaxation step */
            for(kk=0;kk<NSPEEDS;kk++) {
              cells[zz].speeds[kk] = (tmp_cells[zz].speeds[kk]
                                                                    + params.omega *
                                                                    (d_equ[kk] - tmp_cells[zz].speeds[kk]));
      local_density1 += cells[zz].speeds[kk];
            }

   tot_u_x += (cells[zz].speeds[1] +
                                cells[zz].speeds[5] +
                                cells[zz].speeds[8]
                                - (cells[zz].speeds[3] +
                                  cells[zz].speeds[6] +
                                  cells[zz].speeds[7])) /
              local_density1;

    }
   }


 }

 return tot_u_x;
}

int initialise(t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
                    int** obstacles_ptr, double** av_vels_ptr,t_speed** tmp_cells_ptr1,t_speed** tmp_cells_ptr2)
{
 FILE   *fp;      /* file pointer */
 int    ii,jj,zz;    /* generic counters */
 int    xx,yy;    /* generic array indices */
 int    blocked;   /* indicates whether a cell is blocked by an obstacle */
 int    retval;
 double w0,w1,w2;  /* weighting factors */

 /* open the parameter file */
 fp = fopen(PARAMFILE,"r");
 if (fp == NULL) {
  die("could not open file input.params",__LINE__,__FILE__);
 }

 /* read in the parameter values */
 retval = fscanf(fp,"%d\n",&(params->nx));
 if(retval != 1) die ("could not read param file: nx",__LINE__,__FILE__);
 retval = fscanf(fp,"%d\n",&(params->ny));
 if(retval != 1) die ("could not read param file: ny",__LINE__,__FILE__);
 retval = fscanf(fp,"%d\n",&(params->maxIters));
 if(retval != 1) die ("could not read param file: maxIters",__LINE__,__FILE__);
 retval = fscanf(fp,"%d\n",&(params->reynolds_dim));
 if(retval != 1) die ("could not read param file: reynolds_dim",__LINE__,__FILE__);
 retval = fscanf(fp,"%lf\n",&(params->density));
 if(retval != 1) die ("could not read param file: density",__LINE__,__FILE__);
```

```
  retval = fscanf(fp,"%lf\n",&(params->accel));
  if(retval != 1) die ("could not read param file: accel",__LINE__,__FILE__);
  retval = fscanf(fp,"%lf\n",&(params->omega));
  if(retval != 1) die ("could not read param file: omega",__LINE__,__FILE__);

  /* and close up the file */
  fclose(fp);

  /*
  ** Allocate memory.
  **
  ** Remember C is pass-by-value, so we need to
  ** pass pointers into the initialise function.
  **
  ** NB we are allocating a 1D array, so that the
  ** memory will be contiguous.  We still want to
  ** index this memory as if it were a (row major
  ** ordered) 2D array, however.  We will perform
  ** some arithmetic using the row and column
  ** coordinates, inside the square brackets, when
  ** we want to access elements of this array.
  **
  ** Note also that we are using a structure to
  ** hold an array of 'speeds'.  We will allocate
  ** a 1D array of these structs.
  */

  /* main grid */
  *cells_ptr = (t_speed*)malloc(sizeof(t_speed)*(params->ny*params->nx));
  if (*cells_ptr == NULL)
    die("cannot allocate memory for cells",__LINE__,__FILE__);

  /* 'helper' grid, used as scratch space */
  *tmp_cells_ptr = (t_speed*)malloc(sizeof(t_speed)*(params->ny*params->nx));
  if (*tmp_cells_ptr == NULL)

    die("cannot allocate memory for tmp_cells",__LINE__,__FILE__);

*tmp_cells_ptr1 = (t_speed*)malloc(sizeof(t_speed)*(params->nx));
  if (*tmp_cells_ptr1 == NULL)
die("cannot allocate memory for tmp_cells",__LINE__,__FILE__);
*tmp_cells_ptr2 = (t_speed*)malloc(sizeof(t_speed)*(params->nx));
  if (*tmp_cells_ptr2 == NULL)
die("cannot allocate memory for tmp_cells",__LINE__,__FILE__);


  /* the map of obstacles */
  *obstacles_ptr = malloc(sizeof(int*)*(params->ny*params->nx));
  if (*obstacles_ptr == NULL)
    die("cannot allocate column memory for obstacles",__LINE__,__FILE__);

  /* initialise densities */
  w0 = params->density * 4.0/9.0;
```

```
w1 = params->density      /9.0;
w2 = params->density      /36.0;

for(ii=0;ii<params->ny;ii++) {
   //zz1 = ii*params->nx;
   for(jj=0;jj<params->nx;jj++) {

     zz = ii*params->nx + jj;

    (*cells_ptr)[zz].speeds[0] = w0;

   // centre t

   /* axis directions */

  (*cells_ptr)[zz].speeds[1] = w1;
   (*cells_ptr)[zz].speeds[2] = w1;
   (*cells_ptr)[zz].speeds[3] = w1;
   (*cells_ptr)[zz].speeds[4] = w1;
   /* diagonals */
   (*cells_ptr)[zz].speeds[5] = w2;
   (*cells_ptr)[zz].speeds[6] = w2;
   (*cells_ptr)[zz].speeds[7] = w2;
   (*cells_ptr)[zz].speeds[8] = w2;

  }
}

/* first set all cells in obstacle array to zero */
for(ii=0;ii<params->ny;ii++) {
  zz = ii*params->nx;
  for(jj=0;jj<params->nx;jj++) {
    (*obstacles_ptr)[zz + jj] = 0;
  }
}

/* open the obstacle data file */
fp = fopen(OBSTACLEFILE,"r");
if (fp == NULL) {
  die("could not open file obstacles",__LINE__,__FILE__);
}

/* read-in the blocked cells list */
while( (retval = fscanf(fp,"%d %d %d\n", &xx, &yy, &blocked)) != EOF) {
  /* some checks */
  if ( retval != 3)
    die("expected 3 values per line in obstacle file",__LINE__,__FILE__);
  if ( xx<0 || xx>params->nx-1 )
    die("obstacle x-coord out of range",__LINE__,__FILE__);
  if ( yy<0 || yy>params->ny-1 )
    die("obstacle y-coord out of range",__LINE__,__FILE__);
  if ( blocked != 1 )
    die("obstacle blocked value should be 1",__LINE__,__FILE__);
```

```
    /* assign to array */
    (*obstacles_ptr)[yy*params->nx + xx] = blocked;
  }

  /* and close the file */
  fclose(fp);

  /*
  ** allocate space to hold a record of the avarage velocities computed
  ** at each timestep
  */
  *av_vels_ptr = (double*)malloc(sizeof(double)*params->maxIters);

  return EXIT_SUCCESS;
}

int finalise(const t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
             int** obstacles_ptr, double** av_vels_ptr)
{
  /*
  ** free up allocated memory
  */
  free(*cells_ptr);
  *cells_ptr = NULL;

  free(*tmp_cells_ptr);
  *tmp_cells_ptr = NULL;

  free(*obstacles_ptr);
  *obstacles_ptr = NULL;

  free(*av_vels_ptr);
  *av_vels_ptr = NULL;

  return EXIT_SUCCESS;
}

double av_velocity(const t_param params, t_speed* cells, int* obstacles)
{
  int    ii,jj,kk,zz;      /* generic counters */
  int    tot_cells = 0;  /* no. of cells used in calculation */
  double local_density;  /* total density in cell */
  double tot_u_x;       /* accumulated x-components of velocity */

  /* initialise */
  tot_u_x = 0.0;

  /* loop over all non-blocked cells */
  for(ii=0;ii<params.ny;ii++) {
//       zz1 = ii*params.nx;
      for(jj=0;jj<params.nx;jj++) {
        zz = ii*params.nx + jj;
      /* ignore occupied cells */
```

```
    if(!obstacles[zz]) {
            /* local density total */
            local_density = 0.0;
            for(kk=0;kk<NSPEEDS;kk++){
              local_density += cells[zz].speeds[kk];
            }
            /* x-component of velocity */
            tot_u_x += (cells[zz].speeds[1] +
                        cells[zz].speeds[5] +
                        cells[zz].speeds[8]
                        - (cells[zz].speeds[3] +
                          cells[zz].speeds[6] +
                          cells[zz].speeds[7])) /
              local_density;
            /* increase counter of inspected cells */
            ++tot_cells;
    }
   }
  }
//printf("\ntot_ux is %f\n",tot_u_x);
//printf("\ntot_Cells %d\n",tot_cells);
  return tot_u_x / (double)tot_cells;
}

double calc_reynolds(const t_param params, t_speed* cells, int* obstacles)
{
  const double viscosity = 1.0 / 6.0 * (2.0 / params.omega - 1.0);

  return av_velocity(params,cells,obstacles) * params.reynolds_dim / viscosity;
  int ii,jj,kk,zz;       /* generic counters */
  double total = 0.0;  /* accumulator */

  for(ii=0;ii<params.ny;ii++) {
      //zz1 = ii*params.nx;
     for(jj=0;jj<params.nx;jj++) {
        zz = ii*params.nx + jj;
     for(kk=0;kk<NSPEEDS;kk++) {
            total += cells[zz].speeds[kk];
     }
    }
  }

  return total;
}

int write_values(const t_param params, t_speed* cells, int* obstacles, double* av_vels)
{
  FILE* fp;                   /* file pointer */
  int ii,jj,kk,zz;              /* generic counters */
  const double c_sq = 1.0/3.0;  /* sq. of speed of sound */
  double local_density;        /* per grid cell sum of densities */
  double pressure;            /* fluid pressure in grid cell */
  double u_x;                 /* x-component of velocity in grid cell */
```

```
double u_y;             /* y-component of velocity in grid cell */
//int tot_cells =0;
fp = fopen(FINALSTATEFILE,"w");
if (fp == NULL) {
  die("could not open file output file",__LINE__,__FILE__);
}

for(ii=0;ii<params.ny;ii++) {
  for(jj=0;jj<params.nx;jj++) {
     zz = ii*params.nx + jj;
    /* an occupied cell */
    if(obstacles[ii*params.nx + jj]) {
            u_x = u_y = 0.0;
            pressure = params.density * c_sq;
    }
    /* no obstacle */
    else {
            local_density = 0.0;
            for(kk=0;kk<NSPEEDS;kk++) {
              local_density += cells[zz].speeds[kk];
            //tot_cells++;
     }
            /* compute x velocity component */
            u_x = (cells[zz].speeds[1] +
                 cells[zz].speeds[5] +
                 cells[zz].speeds[8]
                 - (cells[zz].speeds[3] +
                         cells[zz].speeds[6] +
                         cells[zz].speeds[7]))
              / local_density;
            /* compute y velocity component */
            u_y = (cells[zz].speeds[2] +
                 cells[zz].speeds[5] +
                 cells[zz].speeds[6]
                 - (cells[zz].speeds[4] +
                         cells[zz].speeds[7] +
                         cells[zz].speeds[8]))
              / local_density;
            /* compute pressure */
            pressure = local_density * c_sq;
    }
    /* write to file */
    fprintf(fp,"%d %d %.12E %.12E %.12E %d\n",ii,jj,u_x,u_y,pressure,obstacles[ii*params.nx + jj]);
  }
}

fclose(fp);

fp = fopen(AVVELSFILE,"w");
if (fp == NULL) {
  die("could not open file output file",__LINE__,__FILE__);
}
for (ii=0;ii<params.maxIters;ii++) {
```

```
    fprintf(fp,"%d:\t%.12E\n", ii, av_vels[ii]);
  }

  fclose(fp);
  return EXIT_SUCCESS;

}

void die(const char* message, const int line, const char *file)
{
  fprintf(stderr, "Error at line %d of file %s:\n", line, file);
  fprintf(stderr, "%s\n",message);
  fflush(stderr);
  exit(EXIT_FAILURE);
}
```

OPENMP CODE FOR LATTICE BOLTZMANN

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

#define NSPEEDS         9
#define PARAMFILE       "input.params"
#define OBSTACLEFILE    "obstacles_300x200.dat"
#define FINALSTATEFILE  "final_state.dat"
#define AVVELSFILE      "av_vels.dat"

/* struct to hold the parameter values */
typedef struct {
  int    nx;           /* no. of cells in y-deirection */
  int    ny;           /* no. of cells in x-direction */
  int    maxIters;     /* no. of iterations */
  int    reynolds_dim;  /* dimension for Reynolds number */
  double density;      /* density per link */
  double accel;        /* density redistribution */
  double omega;        /* relaxation parameter */
} t_param;

/* struct to hold the 'speed' values */
typedef struct {
  double speeds[NSPEEDS];
} t_speed;

enum boolean { FALSE, TRUE };

/*
** function prototypes
*/

/* load params, allocate memory, load obstacles & initialise fluid particle densities */
int initialise(t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
               int** obstacles_ptr, double** av_vels_ptr);

/*
** The main calculation methods.
** timestep calls, in order, the functions:
** accelerate_flow(), propagate(), rebound() & collision()
*/
int timestep(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles);
int accelerate_flow(const t_param params, t_speed* cells, int* obstacles,t_speed* tmp_cells);
int propagate(const t_param params, t_speed* cells, t_speed* tmp_cells, int ii);
int rebound(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles);
int collision(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles);
int write_values(const t_param params, t_speed* cells, int* obstacles, double* av_vels);
```

```
/* finalise, including freeing up allocated memory */
int finalise(const t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
             int** obstacles_ptr, double** av_vels_ptr);

/* Sum all the densities in the grid.
** The total should remain constant from one timestep to the next. */
double total_density(const t_param params, t_speed* cells);

/* compute average velocity */
double av_velocity(const t_param params, t_speed* cells, int* obstacles);

/* calculate Reynolds number */
double calc_reynolds(const t_param params, t_speed* cells, int* obstacles);

/* utility functions */
void die(const char* message, const int line, const char *file);

/*
** main program:
** initialise, timestep loop, finalise
*/
int main(int argc, char* argv[])
{
  t_param  params;          /* struct to hold parameter values */
  t_speed* cells     = NULL;  /* grid containing fluid densities */
  t_speed* tmp_cells = NULL;  /* scratch space */
  int*     obstacles = NULL;  /* grid indicating which cells are blocked */
  double*  av_vels   = NULL;  /* a record of the av. velocity computed for each timestep */
  int      ii;             /* generic counter */
  //clock_t   tic, toc;        /* check points for reporting processor time used */
  struct timeval t1,t2;


  /* initialise our data structures and load values from file */
  initialise(&params, &cells, &tmp_cells, &obstacles, &av_vels);

  /* iterate for maxIters timesteps */
//  tic = clock();
  if (gettimeofday(&t1, NULL) != 0)
    perror("gettimeofday");
  //printf("Number of seconds since Jan 1 1970: %ld\n", (long)t1.tv_sec);

  for (ii=0;ii<params.maxIters;ii++) {


    timestep(params,cells,tmp_cells,obstacles);

    av_vels[ii] = av_velocity(params,cells,obstacles);

#ifdef DEBUG
    printf("==timestep: %d==\n",ii);
    printf("av velocity: %.12E\n", av_vels[ii]);
```

```
    printf("tot density: %.12E\n",total_density(params,cells));
#endif
  }

 if (gettimeofday(&t2, NULL) != 0)
    perror("gettimeofday");


  /* write final values and free memory */
  printf("==done==\n");
  printf("Reynolds number:\t%.12E\n",calc_reynolds(params,cells,obstacles));
  printf("Elapsed time:\t\t%ld (s)\n",((long)t2.tv_sec - (long)t1.tv_sec));
  write_values(params,cells,obstacles,av_vels);
  finalise(&params, &cells, &tmp_cells, &obstacles, &av_vels);

  return EXIT_SUCCESS;
}

int timestep(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles)
{


  accelerate_flow(params,cells,obstacles,tmp_cells);



  rebound(params,cells,tmp_cells,obstacles);

  collision(params,cells,tmp_cells,obstacles);

  return EXIT_SUCCESS;
}

int accelerate_flow(const t_param params, t_speed* cells, int* obstacles, t_speed* tmp_cells)
{
  int ii,jj,zz,prop;     /* generic counters */
  double w1,w2;  /* weighting factors */

  /* compute weighting factors */
  w1 = params.density * params.accel / 9.0;
  w2 = params.density * params.accel / 36.0;

  /* modify the first column of the grid */
 jj=0;
 # pragma omp parallel for private(zz,jj) schedule (static)
  for(ii=0;ii<params.ny;ii++) {
     zz = ii*params.nx+jj ;
    /* if the cell is not occupied and
    ** we don't send a density negative */

    if( !obstacles[zz] &&
                (cells[zz].speeds[3] - w1) > 0.0 &&
                (cells[zz].speeds[6] - w2) > 0.0 &&
```

```
                (cells[zz].speeds[7] - w2) > 0.0 ) {
    /* increase 'east-side' densities */
    cells[zz].speeds[1] += w1;
    cells[zz].speeds[5] += w2;
    cells[zz].speeds[8] += w2;
    /* decrease 'west-side' densities */
    cells[zz].speeds[3] -= w1;
    cells[zz].speeds[6] -= w2;
    cells[zz].speeds[7] -= w2;


  }

   prop = propagate(params, cells, tmp_cells,ii);

   }
  return EXIT_SUCCESS;
}

int propagate(const t_param params, t_speed* cells, t_speed* tmp_cells,int ii)
{
  //int ii,jj,zz;          /* generic counters */
 int zz,jj;
   int x_e,x_w,y_n,y_s;  /* indices of neighbouring cells */
//int reb;
  /* loop over _all_ cells */

# pragma omp parallel for schedule(static) private(zz,x_e,x_w,y_n,y_s)


    for(jj=0;jj<params.nx;jj++) {
        zz = ii*params.nx + jj;


    /* determine indices of axis-direction neighbours
    ** respecting periodic boundary conditions (wrap around) */
    y_n = (ii + 1) % params.ny;
    x_e = (jj + 1) % params.nx;
    y_s = (ii == 0) ? (ii + params.ny - 1) : (ii - 1);
    x_w = (jj == 0) ? (jj + params.nx - 1) : (jj - 1);
    /* propagate densities to neighbouring cells, following
    ** appropriate directions of travel and writing into
    ** scratch space grid */



    tmp_cells[ii *params.nx + jj].speeds[0]  = cells[zz].speeds[0]; /* central cell, */
                                                /* no movement   */
    tmp_cells[ii *params.nx + x_e].speeds[1] = cells[zz].speeds[1]; /* east */

     tmp_cells[y_n*params.nx + jj].speeds[2]  = cells[zz].speeds[2]; /* north */
```

```
      tmp_cells[ii *params.nx + x_w].speeds[3] = cells[zz].speeds[3]; /* west */

       tmp_cells[y_s*params.nx + jj].speeds[4]  = cells[zz].speeds[4]; /* south */


     tmp_cells[y_n*params.nx + x_e].speeds[5] = cells[zz].speeds[5]; /* north-east */

     tmp_cells[y_n*params.nx + x_w].speeds[6] = cells[zz].speeds[6]; /* north-west */


      tmp_cells[y_s*params.nx + x_w].speeds[7] = cells[zz].speeds[7]; /* south-west */

      tmp_cells[y_s*params.nx + x_e].speeds[8] = cells[zz].speeds[8]; /* south-east */


  }
 return EXIT_SUCCESS;
}

int rebound(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles)
{
  int ii,jj,zz;  /* generic counters */

  /* loop over the cells in the grid */

# pragma omp parallel for schedule(static) private(jj,zz)
 for(ii=0;ii<params.ny;ii++) {



    for(jj=0;jj<params.nx;jj++) {

       zz = ii*params.nx + jj;
     /* if the cell contains an obstacle */

     if(obstacles[zz]) {
                /* called after propagate, so taking values from scratch space
                ** mirroring, and writing into main grid */

                cells[zz].speeds[1] = tmp_cells[zz].speeds[3];

                cells[zz].speeds[2] = tmp_cells[zz].speeds[4];

                cells[zz].speeds[3] = tmp_cells[zz].speeds[1];

                cells[zz].speeds[4] = tmp_cells[zz].speeds[2];

       cells[zz].speeds[5] = tmp_cells[zz].speeds[7];

       cells[zz].speeds[6] = tmp_cells[zz].speeds[8];
```

```
      cells[zz].speeds[7] = tmp_cells[zz].speeds[5];

      cells[zz].speeds[8] = tmp_cells[zz].speeds[6];



   }


  }
 }


 return EXIT_SUCCESS;
}

int collision(const t_param params, t_speed* cells, t_speed* tmp_cells, int* obstacles)
{
 int kk;              /* generic counters */
 const double c_sq = 1.0/3.0;  /* square of speed of sound */
 const double w0 = 4.0/9.0;    /* weighting factor */
 const double w1 = 1.0/9.0;    /* weighting factor */
 const double w2 = 1.0/36.0;   /* weighting factor */
 double u_x,u_y;             /* av. velocities in x and y directions */
 double u[NSPEEDS];          /* directional velocities */
 double d_equ[NSPEEDS];      /* equilibrium densities */
 double u_sq;                /* squared velocity */
 double local_density;       /* sum of densities in a particular cell */
 int ii,jj,zz;
 /* loop over the cells in the grid
 ** NB the collision step is called after
 ** theii*params.nx + jj
propagate step and so values of interest
 ** are in the scratch-space grid */



  # pragma omp parallel for   private (jj,u,d_equ,zz,local_density,kk,u_x,u_y,u_sq)schedule (static)

  for(ii=0;ii<params.ny;ii++) {

        for(jj=0;jj<params.nx;jj++) {

      zz = ii*params.nx + jj;

       /* don't consider occupied cells */
       if(!obstacles[zz]) {
              /* compute local density total */
              local_density = 0.0;
              for(kk = NSPEEDS-1;kk!=-1;kk--) {
                local_density += tmp_cells[zz].speeds[kk];
              }
              /* compute x velocity component */
```

```
    u_x = (tmp_cells[zz].speeds[1] +
                    tmp_cells[zz].speeds[5] +
                    tmp_cells[zz].speeds[8]
                    - (tmp_cells[zz].speeds[3] +
                            tmp_cells[zz].speeds[6] +
                            tmp_cells[zz].speeds[7]))
                / local_density;
               /* compute y velocity component */

    u_y = (tmp_cells[zz].speeds[2] +
                    tmp_cells[zz].speeds[5] +
                    tmp_cells[zz].speeds[6]
                    - (tmp_cells[zz].speeds[4] +
                            tmp_cells[zz].speeds[7] +
                            tmp_cells[zz].speeds[8]))
                / local_density;


/* velocity squared */
               u_sq = u_x * u_x + u_y * u_y;
               /* directional velocity components */
               u[1] =   u_x;       /* east */
               u[2] =        u_y;  /* north */
               u[3] = - u_x;       /* west */
               u[4] =      - u_y;  /* south */
               u[5] =   u_x + u_y;  /* north-east */
               u[6] = - u_x + u_y;  /* north-west */
               u[7] = - u_x - u_y;  /* south-west */
               u[8] =   u_x - u_y;  /* south-east */
               /* equilibrium densities */
               /* zero velocity density: weight w0 */
               d_equ[0] = w0 * local_density * (1.0 - u_sq / (2.0 * c_sq));
               /* axis speeds: weight w1 */

               d_equ[1] = w1 * local_density * (1.0 + u[1] / c_sq
                                                       + (u[1] * u[1]) / (2.0 * c_sq * c_sq)
                                                       - u_sq / (2.0 * c_sq));
               d_equ[2] = w1 * local_density * (1.0 + u[2] / c_sq
                                                       + (u[2] * u[2]) / (2.0 * c_sq * c_sq)
                                                       - u_sq / (2.0 * c_sq));
               d_equ[3] = w1 * local_density * (1.0 + u[3] / c_sq
                                                       + (u[3] * u[3]) / (2.0 * c_sq * c_sq)
                                                       - u_sq / (2.0 * c_sq));
               d_equ[4] = w1 * local_density * (1.0 + u[4] / c_sq
                                                       + (u[4] * u[4]) / (2.0 * c_sq * c_sq)
                                                       - u_sq / (2.0 * c_sq));
               // diagonal speeds: weight w2
               d_equ[5] = w2 * local_density * (1.0 + u[5] / c_sq
                                                       + (u[5] * u[5]) / (2.0 * c_sq * c_sq)
                                                       - u_sq / (2.0 * c_sq));
```

```
            d_equ[6] = w2 * local_density * (1.0 + u[6] / c_sq
                                                    + (u[6] * u[6]) / (2.0 * c_sq * c_sq)
                                                    - u_sq / (2.0 * c_sq));
            d_equ[7] = w2 * local_density * (1.0 + u[7] / c_sq
                                                    + (u[7] * u[7]) / (2.0 * c_sq * c_sq)
                                                    - u_sq / (2.0 * c_sq));
            d_equ[8] = w2 * local_density * (1.0 + u[8] / c_sq
                                                    + (u[8] * u[8]) / (2.0 * c_sq * c_sq)
                                                    - u_sq / (2.0 * c_sq));


            /* relaxation step */

            for(kk=0;kk<NSPEEDS;kk++) {
              cells[zz].speeds[kk] = (tmp_cells[zz].speeds[kk]
                                                                + params.omega *
                                                                (d_equ[kk] - tmp_cells[zz].speeds[kk]));
            }

     }
    }
   }


 return EXIT_SUCCESS;
}

int initialise(t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
                int** obstacles_ptr, double** av_vels_ptr)
{
  FILE   *fp;      /* file pointer */
  int    ii,jj,zz;    /* generic counters */
  int    xx,yy;    /* generic array indices */
  int    blocked;  /* indicates whether a cell is blocked by an obstacle */
  int    retval;
  double w0,w1,w2;  /* weighting factors */

  /* open the parameter file */
  fp = fopen(PARAMFILE,"r");
  if (fp == NULL) {
    die("could not open file input.params",__LINE__,__FILE__);
  }

  /* read in the parameter values */
  retval = fscanf(fp,"%d\n",&(params->nx));
  if(retval != 1) die ("could not read param file: nx",__LINE__,__FILE__);
  retval = fscanf(fp,"%d\n",&(params->ny));
  if(retval != 1) die ("could not read param file: ny",__LINE__,__FILE__);
  retval = fscanf(fp,"%d\n",&(params->maxIters));
  if(retval != 1) die ("could not read param file: maxIters",__LINE__,__FILE__);
  retval = fscanf(fp,"%d\n",&(params->reynolds_dim));
  if(retval != 1) die ("could not read param file: reynolds_dim",__LINE__,__FILE__);
  retval = fscanf(fp,"%lf\n",&(params->density));
```

```
  if(retval != 1) die ("could not read param file: density",__LINE__,__FILE__);
  retval = fscanf(fp,"%lf\n",&(params->accel));
  if(retval != 1) die ("could not read param file: accel",__LINE__,__FILE__);
  retval = fscanf(fp,"%lf\n",&(params->omega));
  if(retval != 1) die ("could not read param file: omega",__LINE__,__FILE__);

  /* and close up the file */
  fclose(fp);

  /*
  ** Allocate memory.
  **
  ** Remember C is pass-by-value, so we need to
  ** pass pointers into the initialise function.
  **
  ** NB we are allocating a 1D array, so that the
  ** memory will be contiguous.  We still want to
  ** index this memory as if it were a (row major
  ** ordered) 2D array, however.  We will perform
  ** some arithmetic using the row and column
  ** coordinates, inside the square brackets, when
  ** we want to access elements of this array.
  **
  ** Note also that we are using a structure to
  ** hold an array of 'speeds'.  We will allocate
  ** a 1D array of these structs.
  */

  /* main grid */
  *cells_ptr = (t_speed*)malloc(sizeof(t_speed)*(params->ny*params->nx));
  if (*cells_ptr == NULL)
    die("cannot allocate memory for cells",__LINE__,__FILE__);

  /* 'helper' grid, used as scratch space */
  *tmp_cells_ptr = (t_speed*)malloc(sizeof(t_speed)*(params->ny*params->nx));
  if (*tmp_cells_ptr == NULL)
    die("cannot allocate memory for tmp_cells",__LINE__,__FILE__);

  /* the map of obstacles */
  *obstacles_ptr = malloc(sizeof(int*)*(params->ny*params->nx));
  if (*obstacles_ptr == NULL)
    die("cannot allocate column memory for obstacles",__LINE__,__FILE__);

  /* initialise densities */
  w0 = params->density * 4.0/9.0;
  w1 = params->density      /9.0;
  w2 = params->density      /36.0;

# pragma omp parallel for schedule(static) private(jj,zz)
for(ii=0;ii<params->ny;ii++) {

     for(jj=0;jj<params->nx;jj++) {
```

```
      zz = ii*params->nx + jj;

     (*cells_ptr)[zz].speeds[0] = w0;

    // centre

    /* axis directions */

   (*cells_ptr)[zz].speeds[1] = w1;
    (*cells_ptr)[zz].speeds[2] = w1;
    (*cells_ptr)[zz].speeds[3] = w1;
    (*cells_ptr)[zz].speeds[4] = w1;
    /* diagonals */
    (*cells_ptr)[zz].speeds[5] = w2;
    (*cells_ptr)[zz].speeds[6] = w2;
    (*cells_ptr)[zz].speeds[7] = w2;
    (*cells_ptr)[zz].speeds[8] = w2;

  }
 }

 /* first set all cells in obstacle array to zero */
# pragma omp parallel for schedule(static) private(jj,zz)
 for(ii=0;ii<params->ny;ii++) {
   zz = ii*params->nx;
   for(jj=0;jj<params->nx;jj++) {
     (*obstacles_ptr)[zz + jj] = 0;
   }
 }

 /* open the obstacle data file */
 fp = fopen(OBSTACLEFILE,"r");
 if (fp == NULL) {
   die("could not open file obstacles",__LINE__,__FILE__);
 }

 /* read-in the blocked cells list */
 while( (retval = fscanf(fp,"%d %d %d\n", &xx, &yy, &blocked)) != EOF) {
   /* some checks */
   if ( retval != 3)
     die("expected 3 values per line in obstacle file",__LINE__,__FILE__);
   if ( xx<0 || xx>params->nx-1 )
     die("obstacle x-coord out of range",__LINE__,__FILE__);
   if ( yy<0 || yy>params->ny-1 )
     die("obstacle y-coord out of range",__LINE__,__FILE__);
   if ( blocked != 1 )
     die("obstacle blocked value should be 1",__LINE__,__FILE__);
   /* assign to array */
   (*obstacles_ptr)[yy*params->nx + xx] = blocked;
 }

 /* and close the file */
 fclose(fp);
```

```
  /*
  ** allocate space to hold a record of the avarage velocities computed
  ** at each timestep
  */
  *av_vels_ptr = (double*)malloc(sizeof(double)*params->maxIters);

  return EXIT_SUCCESS;
}

int finalise(const t_param* params, t_speed** cells_ptr, t_speed** tmp_cells_ptr,
                    int** obstacles_ptr, double** av_vels_ptr)
{
  /*
  ** free up allocated memory
  */


  free(*cells_ptr);
  *cells_ptr = NULL;

  free(*tmp_cells_ptr);
  *tmp_cells_ptr = NULL;

  free(*obstacles_ptr);
  *obstacles_ptr = NULL;

  free(*av_vels_ptr);
  *av_vels_ptr = NULL;

  return EXIT_SUCCESS;
}

double av_velocity(const t_param params, t_speed* cells, int* obstacles)
{
  int    ii,jj,kk,zz;      /* generic counters */
  int    tot_cells = 0;  /* no. of cells used in calculation */
  double local_density;  /* total density in cell */
 double tot_u_x;       /* accumulated x-components of velocity */

  /* initialise */
  tot_u_x = 0.0;

 /* loop over all non-blocked cells */
#pragma omp parallel for schedule(static) private(jj,zz,kk,local_density) reduction(+:tot_u_x ,tot_cells)
 for(ii=0;ii<params.ny;ii++) {

//  # pragma omp parallel for schedule(dynamic) private(kk,zz,local_density)shared(tot_u_x,tot_cells)
     for(jj=0;jj<params.nx;jj++) {
       zz = ii*params.nx + jj;
     /* ignore occupied cells */
     if(!obstacles[zz]) {
                /* local density total */
```

```
            local_density = 0.0;
            for(kk=0;kk<NSPEEDS;kk++){
              local_density += cells[zz].speeds[kk];
            }

    /* x-component of velocity */

    tot_u_x += (cells[zz].speeds[1] +
                              cells[zz].speeds[5] +
                              cells[zz].speeds[8]
                              - (cells[zz].speeds[3] +
                                cells[zz].speeds[6] +
                                cells[zz].speeds[7])) /
              local_density;
            /* increase counter of inspected cells */

            ++tot_cells;
    }
   }
  }

  return tot_u_x / (double)tot_cells;
}

double calc_reynolds(const t_param params, t_speed* cells, int* obstacles)
{
  const double viscosity = 1.0 / 6.0 * (2.0 / params.omega - 1.0);

  return av_velocity(params,cells,obstacles) * params.reynolds_dim / viscosity;
}

double total_density(const t_param params, t_speed* cells)
{
  int ii,jj,kk,zz;       /* generic counters */
  double total = 0.0;  /* accumulator */
  #pragma omp parallel for schedule(static) private(jj,kk)
  for(ii=0;ii<params.ny;ii++) {

    for(jj=0;jj<params.nx;jj++) {
       zz = ii*params.nx + jj;
    for(kk=0;kk<NSPEEDS;kk++) {
            total += cells[zz].speeds[kk];
    }
   }
  }

  return total;
}

int write_values(const t_param params, t_speed* cells, int* obstacles, double* av_vels)
{
```

```
FILE* fp;                   /* file pointer */
int ii,jj,kk,zz;              /* generic counters */
const double c_sq = 1.0/3.0;  /* sq. of speed of sound */
double local_density;        /* per grid cell sum of densities */
double pressure;            /* fluid pressure in grid cell */
double u_x;                 /* x-component of velocity in grid cell */
double u_y;                 /* y-component of velocity in grid cell */

fp = fopen(FINALSTATEFILE,"w");
if (fp == NULL) {
  die("could not open file output file",__LINE__,__FILE__);
}

for(ii=0;ii<params.ny;ii++) {
  for(jj=0;jj<params.nx;jj++) {
    zz = ii*params.nx + jj;
    if(obstacles[zz]) {
              u_x = u_y = 0.0;
              pressure = params.density * c_sq;
    }
    /* no obstacle */
    else {
              local_density = 0.0;
              for(kk=0;kk<NSPEEDS;kk++) {
                local_density += cells[zz].speeds[kk];
              }
              /* compute x velocity component */
              u_x = (cells[zz].speeds[1] +
                   cells[zz].speeds[5] +
                   cells[zz].speeds[8]
                   - (cells[zz].speeds[3] +
                             cells[zz].speeds[6] +
                             cells[zz].speeds[7]))
                / local_density;
              /* compute y velocity component */
              u_y = (cells[zz].speeds[2] +
                   cells[zz].speeds[5] +
                   cells[zz].speeds[6]
                   - (cells[zz].speeds[4] +
                             cells[zz].speeds[7] +
                             cells[zz].speeds[8]))
                / local_density;
              /* compute pressure */
              pressure = local_density * c_sq;
    }
    /* write to file */
    fprintf(fp,"%d %d %.12E %.12E %.12E %d\n",ii,jj,u_x,u_y,pressure,obstacles[zz]);
  }
}

fclose(fp);

fp = fopen(AVVELSFILE,"w");
```

```
  if (fp == NULL) {
    die("could not open file output file",__LINE__,__FILE__);
  }
  for (ii=0;ii<params.maxIters;ii++) {
    fprintf(fp,"%d:\t%.12E\n", ii, av_vels[ii]);
  }

  fclose(fp);

  return EXIT_SUCCESS;
}

void die(const char* message, const int line, const char *file)
{
  fprintf(stderr, "Error at line %d of file %s:\n", line, file);
  fprintf(stderr, "%s\n",message);
  fflush(stderr);
  exit(EXIT_FAILURE);
}
```