CHAPTER 1

INTRODUCTION

The computing industry changed course in the year 2005 when Intel followed the trend of IBM's Power 4 and Sun Microsystems Niagara processor in announcing that its high performance microprocessors would henceforth rely on multiple processors or cores. The new industry buzzword "multicore" captures the plan of doubling the number of standard cores per die with every semi conductor process generation starting with a single processor. Multicore will obviously help multiprogrammed workloads, which contain a mix of independent sequential tasks, but how will individual tasks become faster? Switching from sequential to modestly parallel computing will make programming much more difficult without rewarding this greater effort with a dramatic improvement in power-performance (see Krste et al., (2006)). Hence, multicore is unlikely to be the ideal answer.

Rather than multicore, focus is shifted to manycore architectures. Successful manycore architectures and supporting software technologies could reset microprocessor hardware and software road maps for the next thirty years (Krste et al., (2006)).

Parallel computing environments based on distributed computing have become effective and economical platforms for high performance computing by providing controlled access to much larger and richer computational resources (Quinn (2001)). Distributed systems can increase application performance by a significant amount and the incremental enhancement of a network based on concurrent computing environment is usually straightforward because of the availability of high bandwidth networks (Chan & Saied (1987)). Computer Simulations have opened up a third dimension in scientific

investigation alongside theory and experimentation. Clock rates of processors have increased from about 40MHZ to over 2.0GHZ (e.g., a Pentium 4, circa 2002). The average number of cycles per instruction (CPI) of high end processors has improved by roughly an order of magnitude over the past ten years. All these translate to an increase in the peak floating point operation execution rate (floating point operation per second, or FLOPS) of several orders of magnitude (Annanth G. et. al, 2003). Many authors have discussed paradigms and languages for programming parallel computers, e.g. Akl (1997), Albert Zomaya (1996), Foster (1996), Quinn (2001) and Barry & Michael (2003), on various aspects of parallel algorithm design and analysis. Ted and El-Rewini (1992), Cluadia (2001), discussed various aspects of parallel computing using clusters. Jaja (1992) covers parallel algorithms for the Parallel Random access Machine (PRAM) model of computation. Anderson et al., (1998) discussed the case for network of workstations. Fox et al., (1988) provide an application-oriented view of algorithm design for problems in scientific computing.

Parallel computing refers to solving a task in a faster manner by employing multiple processors simultaneously. Driven by the continual desire for more computing power and the availability of appropriate technologies, parallel computing became popular in the late 1980s. Unfortunately, it soon became obvious that being able to build fast parallel hardware was not good enough. The real challenge turned out to be software. Claudia (2001) gave a survey of models and distinguishes between the parallel computing and the distributed computing. Parallel computing was only practised in areas with a high demand for computing power. Currently, networks of Personal Computers (PC) or workstations, so-called clusters, are gaining wide acceptance as affordable hardware platforms for parallelism.

A distributed system is a collection of autonomous computers that are interconnected with each other and are able to cooperate, thereby sharing resources such

as printers and databases. Distributed computing evolved during the 1980s and became ubiquitous during the 1990s. Distributed systems can be coupled as loosely as a network of autonomous PCs that access the same printers, or so tightly that the user has the view of a large homogeneous computing resource. Distributed systems are not sold as a whole, but grow naturally and incrementally. Individual computers are interconnected to local-area networks, and local-area networks are interconnected to wide-area networks, as the need arises. Claudia (2001) gave the following common characteristics between Parallel and Distributed Computing as: (i) multiple processors are used, (ii) the processors are interconnected by some network, and (iii) multiple computational activities (processes) are in progress at the same time and cooperate with each other. Some author considers parallel computing as a subfield of distributed computing (Albert (1996)). Many computations can be characterized as both parallel and distributed; we make a distinction summarized as follows: (i) parallel computing splits an application up into tasks that are executed at the same time, whereas distributed computing splits an application up into tasks that are executed at different locations, using different resources. (ii) Parallel computing places emphasis on the following features: (a) an application is split into subtasks that are solved simultaneously, often in a tightly coupled manner, (b) one application is considered at a time, and the goal is to speed up the processing of that single application, (c) the programs are usually run on homogeneous architectures, which may have a shared memory distributed computing.

Distributed computing places emphasis on the following features: (i) the computation use multiple resources that are situated in physically distant locations. Resources can be processors, memories, disks and databases; (ii) a distributed system runs multiple applications at a time. The application may belong to different users; (iii) distributed systems are often heterogeneous, open, and dynamic. Open means that the system comprises of device hardware and software from different vendors. Dynamic

means that the system structure can change over time, (iv) distributed systems do not have a shared memory, at least not at the hardware level. Non determinism and dynamic behavior are typical in parallel computing, Claudia (2001), aims at speeding up the execution of a single application. Distributed computing emphasizes the aspect of increasing the system throughput. As a general tendency, performance plays a larger role in parallel than in distributed computing. Despite these differences, the areas of parallel and distributed computing have a significant overlap, as noted. Since the mid 1990s, the areas have been converging and are described as being one field, which are: (1) the areas used the same architectures, on one hand, the invention of the fast network technologies enables the use of clusters in parallel computing. On the other hand, parallel machines are used as servers in distributed computing. (2) The issues of parallelism and distribution are researched together. Parallel and distributed computing is normally considered to be the union of the two fields.

Bertsekas and Tsitsiklis (1998) discuss parallel algorithms, with emphasis on numerical applications. Messina & Murli (1991) and Buyya (1999) present a collection of papers on various aspects of the application and potential of parallel computing. It is human nature to envision new applications that exceed the capabilities of computer systems and require more computational speed than presently available (Braunl (1993)). One way of increasing the computational speed is by using multiple processors operating together on a single problem (Blaise (1994)). The overall problem is split into parts, each of which is performed by a separate processor in parallel. Writing programs for the form of computations is known as Parallel Programming. Problems often cannot be divided perfectly into independent parts and interaction is necessary between parts, both for data transfer and synchronization of computations. What makes parallel computing timeless is that the continual improvements in execution speeds of single processors make parallel computers even faster, and there will always be grand

challenge problems that cannot be solved in a reasonable amount of time on current computers. The use of multiple processors often allows a larger or more precise solution of a problem to be solved in a reasonable amount of time. A parallel computer is not a new idea, for example, Gill writes about parallel programming in 1958 (Gill (1958)). Holland wrote about "computer capable of executing an arbitrary number of sub-programs simultaneously" in 1959 (Holland, 1959). Conway described the design of a parallel computer and its programming in 1963 (Conway (1963)). Papers with similar titles continue to be written 30 years later (Karp (1987)). Flynn (1972) and Flynn & Rudd (1996), wrote that "the continued drive for higher and higher performance system leads us to one simple conclusion: "the future is parallel". Parallel algorithm design for workstation clusters using the Master/Slave on the Single Program Multiple Data environment used to speed up the execution of a large class of existing sequential programs by Magee and Cheung (1991) was described.

Sequential numerical methods for solving time dependent problems have been explored extensively (Noye (1996) & Smith (1985)). Attempts have also been made towards parallel solutions on distributed memory Multiple Instruction Multiple Data (MIMD) machines. Large scale computational scientific and engineering problems, such as time dependent and 3D flows of viscous elastic fluids required large computational resources, with a performance approaching tens of giga (10^9) floating point calculations per seconds. An alternative and cost effective means of achieving a comparable performance is by way of distributed computing, using a system of processors loosely connected through a local area network (Chypher et al., (1993)). Relevant data need to be passed from processor to processor through a message passing mechanism (Fan et. al., (2003), Durst et al., (1993), Peizong (2002) & Jaris (2003)).

Traditionally, software has been written for serial computation to be run on a single computer having a single Central Processing Unit (CPU), where a problem is

broken into a discrete series of instructions and is executed one after the other with one instruction executed at any moment in time. Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. A problem is broken down into discrete parts to be solved concurrently, and broken into series of instructions executed on different CPUs (Tian & Yang (2007)). The computer resources include a single computer with multiple processors; number of computers connected by a network, combination of both. Parallel computing is used because: (1) it saves time (2) solve larger problems (3) provide concurrency i.e. multiple things done at the same time. (4) Cost saving i.e. use multiple resource instead of paying for a supercomputer (5) overcoming memory constraints. The uses of parallel computers also overcome limits to serial computing through transmission speed (Barry (2003), Sun & Gustafson (1991)) whose increasing speeds necessitate increasing proximity of processing elements. Hence, parallelism is the future of computing.

Many numerical methods have been suggested for the solution of Parabolic and Telegraphic Equations. Two types of finite-difference equations which have been studied are explicit difference equations and implicit difference equations (Evans (1988a) & Smith (1985), Burden & Douglass (2000)). In analyzing true dependable Parabolic and Telegraphic problems (Johnson (1982), Jain (1984), Sahimi (1993) & Evans (2003)) using numerical methods, the authors found that the machine time required was large and the cost prohibitive. This was true regardless of the method used. When explicit methods were used, the time step was restricted to value very much smaller than the maximum allowable for the solution of the differential equation, see (Yanenko (1971)). On the other hand, when the implicit methods were tried, the problem of solving large scale linear systems was encountered (Evans (1989) & Yanenko (1971)).

Succi et al., (1998) gave the under listed goals as motivation of Parallel and Distributed Computing. The three primary goals in parallel computing are: (i) cut turnaround time – this is a common goal in Industry, do more computations in less time without changing the size of the problem, (ii) job upside – common in academia where one would like to be call to look at the complex scenario generated by the interactions of more degrees of freedom, (iii) both. There are two factors that allow one to achieve these goals (Succi et al., (1998)): (i) S_1, the speed of a single processor and P, the number of processors or granularity. S_1 is a measure in megaflops (millions of floating point operations per seconds), (ii) the complexity, which is the size of the problem. The common aim of the three goals is to maximize the processing speed. It is generally believed that one cannot go beyond 10^9 flops/sec on a single chip. Alternatively, one can bundle together a large number of processors and have them working concurrently on distinct sub-portions of the global problem, the "divide and conquer" approach. One of the primary goals of theoretical parallel computing is to develop models which allow one to predict the type of efficiency one can obtain. This depends on many factors relating to both the details of the computational applications and the technological features of the computer architecture (Culler et al., (1999)).

El-Rewini & Lewis (1998) predict that the performance growth curve will peak around the year 2005. Significant progress has been made in network technologies so that fast, large and long distance networks can be built at a reasonable price. Another restriction of sequential architecture is the fact that the access time to memory limits the overall performance, the so-called Von-Neumann bottleneck. Access time is a problem of both main memory and disks, and it is likely to get worse in the future. Parallel and distributed computing helps insofar as that it increases the cache and main memory capacities as a by-product of adding processors. It allows data to be processed close to the location where they are generated, which reduces the amount of traffic to a system

wide memory; (iv) incremental growth – Parallel and Distributed computing are scalable, that is, in most cases we can add processors.

The key characteristics of parallel and distributed computing (Laurant 2001) cover the objectives, major problems, and the basic notions of parallel and distributed computing. The basic notion include: Task – which is the program or a part of the program in execution. The typical task has a single thread of control, but sometimes the term is also applied to computational activities that contain subtasks. The term "Process" is used synonymously with task. In contrast to a task, a "Job" is a whole, usually sequential, program in execution. A "Node or Process Elements" is an entity that is able to compute.

1.1 Key Concept and Trends

The key concept and trend recognize the main areas influencing the design and development of environments for parallel distributed computing, we wish to outline some general but key issues naturally emerging from advances and changes in distributed applications. Two fundamental concepts are software integration and interoperability, that refer to the growing need of assembling software modules or subsystem into an operational system and of doing such composition easily and reliably, possibly in a "plug-and-play" fashion (Ambrosiano et al., (2001)).

Strictly connected is the concept of reuse of existing software expressing the need of preserving years of research and development. To satisfy these needs many issues must be addressed: the definition of interoperability standards, including common software abstractions and interfaces and the languages for describing them, models and mechanisms for linking different software units and transferring data between them, consistent schemes for memory management and error handling, and so on.

Component standards and implementations, e.g. Horsemann & Kirtland, (1997), Englander (2001), were initially developed by the business world, that recognized their importance. However, they do not support basic needs high-performance computing such as the abstraction needed by parallel programming and the performance. A large effort is currently devoted to defining a standard component architecture for high-performance computing in the context of the Common Component Architecture (CCA) Forum (see Armstrong et al., (1999)).

In this scenario, a significant role should be played by repositories of software interfaces, implementations and documentation searchable by both human and machine clients. Mobile agents are recognized as a possible solution for resource discovery and monitoring, and expert assistants and knowledge discovery in data base techniques are considered useful for the selection for the selection of the most appropriate software or software machine par (Joshi et al., (2000)). Already performance is still a fundamental feature and achieving it is made much more difficult by the variety and complexity of current hardware and software platforms. A challenging goal is to obtain software with portable performance. In this context, compiler technology and parallelism exploitation tools play a central role, as well as techniques for the development of self-adaptive, latency tolerant, parameterized or performance-annotated code (Petitet et al., (2001)). Hence, an important design principle of parallel and distributed environments are ease of use, where the term ‘ease” is obviously related to the expertise of the target users. Among the other things, this requires languages and or Graphical User Interfaces (GUIs) close to the users knowledge domain, for design, implementation, composition, execution and analysis of applications, and other tools such as debuggers and profilers. All the above concepts can be considered suitable common goals in the design of modern environments for parallel

and distributed computing. However, existing implementations or proposals of such environments usually address different topics with different emphasis.

1.2 Motivation for the Thesis

The promise of parallelism has fascinated researchers for three decades. In the past, parallel computing efforts have shown promise and gathered investment, but in the end, uniprocessor computing always prevail. Nevertheless, we argue general-purpose computing is taking an irreversible step toward parallel architectures. What's different this time? This shift towards increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.

Here, our goal is to delineate application requirements in a manner that is not overly specific to individual applications or the optimizations used for certain hardware platforms. Our approach, described in this thesis, is to define a number of numerical algorithms on our built hardware platform, which each capture a pattern of computation and communication common to a class of important applications (we place emphasis on overlapping communication and computation).

1.3 Contribution of this Thesis

The primary contribution of this thesis is a fast, low-overhead data and thread migration mechanism. In this work, we created Armadillo Generation Distributed System (AGDS) motivated by the need to be at par with present trends of employing parallel computing possibilities across heterogeneous and homogeneous computing environments to solve

large problems a single sequential computer cannot solve due to memory constraint and speed. The amount of time required to coordinate parallel tasks with the use of AGDS involves task start-up time, synchronizations, data communications, software overheads and task termination. The AGDS was setup for PVM-based parallel processing, which involves confining the hosts on a network with Linux operating system in Institute of Mathematical Sciences, University of Malaya.

Here, we present the parallel implementation of the explicit and implicit alternating schemes on MPI/PVM clusters with the DD method and using the SPMD model to obtain results with significant accuracies. We solve the 1-D, 2-D Parabolic Equations exemplified on Bio-Heat Equation, 1-D, 2-D and 3-D Telegraph Equations by using the double sweep methods of Peaceman-Rachford (P-R) (Peaceman & Rachford (1955)) and Mitchell-Fairweather (MF-DS) (Mitchell & Fairweather (1964)). Each method involves the solution of sets of tridiagonal equations along the lines parallel to the x and y axes at the first and second time steps, respectively. We use two approaches of numerical schemes to approximate the differential equations of the implicit alternating method (Rohalla & Paiviz (2007)). The first is by employing the double sweep of P-R and the tridiagonal system of equations that arises from the difference method is then solved by using the two-stage IADE of D'Yakonov and other higher order iterative methods. We computed some examples to test the parallel algorithms and the effects of various parameters on the performance of the algorithms are discussed.

The prime objective of our platform is not to be specific to one problem; it should be able to solve a wide variety of time-dependent PDE for various applications (Chi-Chung et al., (1994), Sun (1991), Tian (2007) & McDonough (1994)). To maintain scalability, efficiency and speedup of the various algorithms considered in this thesis, we use routines that provide non-blocking functionality to overlap communication with

computation to reduce communication delays. We compared parallel performance of the algorithms. However, this process tends to break the sequential constraint in the view of simulating complex grid sizes on the differential equations. As a result data overhead is similar to L2 cache fills on a conventional uniprocessor system. The key architecture features that enable the data is data representation using capabilities communication and memory access through architectural explicit queues.

We do believe that a generic framework is possible, a framework that could be deployed for distributed applications, as well as support several services necessary for end-user applications. We begin with the buildup of our parallel platform and the services it supports. The idea behind this is to discover the infrastructure requirements of such services, which are considered in the design of our platform. Afterwards, we describe various algorithms implemented on two different differential equations and their implementation on two different parallel platforms using MPI and PVM, each of which deals with a significant of the proposed framework. The first chapter of this thesis addresses core functionalities of the framework, and the rest of the chapters depict use cases where some service is added to the framework.

Before starting on the design of our framework, we make a concise description on the platforms, in particular on the fields of the services we aim at providing with our framework. Data adaption is the core component of the platform application, since it deals with the data domain adaption as well as proposes algorithms that can be used for high-level services. We provide a mapping function which adapt overlapping of communication and computation to avoid unnecessary synchronization. We use the Single Program Multiple Data (SPMD) as a technique to achieve parallelism under the Domain Decomposition (DD) strategy. SPMD is the most common style of parallel programming (Eduardo et al., (2007)).

The master/slave concurrent programming model is typical of the SPMD model. To help with the program development under a distributed computing environment, a number of software tools have been developed. Parallel Virtual Machine (PVM) is chosen here since it has a large user groups (Geist et. al., (1992)) and Message Passing Interface (MPI) (Groop et. al., (1999), Snir et. al., (1998) and McMahon & Skjellum (1996)) which simplified numerous issues for both application developers.

Historically, users have written scientific applications for large distribution memory computers using explicit communications as the programming model (Callahan & Kennedy (1988)). This trend crystallized with the creation of automatic data and computation decomposition on distributed memory computers (Peizong (2002), Gupta (1992a)) and Wide-Area implementation of the MPI (Foster et al., (1998)) since the overall performance of the distributed system often depends on the effectiveness of its communication. Parallel algorithms have been implemented for the finite-difference method (Guang-Wei et al., (2001)), the discrete eigenfunctions method (Aloy et al., (2007)) and Evans (2003) used the AGE method on 1-D problem. Johnson et al., (1987) have implemented the Alternating Direction methods on multiprocessors, the boundary element method and the finite volume method using the DD have also been implemented. In addition, time and functional decompositions have also been used in parallel implementation.

This thesis is organized as follows: Chapter 2 introduces the basic notions of parallel computing, the parallel machine models and interconnects as well as methodologies for designing and building parallel programs. Focus was also on the motivation and differences of the considered software tools. Chapter 3 briefs on the various finite-difference methods on 1-D and 2-D Parabolic Equations and their related numerical methods. Chapter 4 presents the various numerical schemes and their improvement on 1-D and 2-D Bio-Heat Equations. Chapter 5 presents the related

numerical schemes on 1-D, 2-D and 3-D Telegraph Equations. Here, we treat some properties of accuracy and stability on 2-D and 3-D case with parallel implementation the algorithms as well as parallel performance analysis. Chapter 6 collects the results and benchmark problems for 1-D, 2-D and 3-D case for Parabolic, Bio-Heat and Telegraph Equations. Chapter 7 gives the discussions of the results on both MPI/PVM platforms. Finally, a summary of the work and future works are included in chapter 8.

1.4 Related Work

In parallel computing, the design and implementation of message-passing applications have been recognized as complex tasks. PVM and MPI have improved the situation, as they permit the implementation of applications independently on underlying architecture. PVM and MPI allow for the general form of parallel computation, as programs may exhibit arbitrary communication dependencies. In general, programs forming tree inter-process communication dependencies, where each process communicates only with its parent and its child processes, are well suited to PVM, while regular ring or grid process communication dependencies are well suited to MPI. However, as we know, PVM and MPI are low-level tools and somewhat difficult to use for building applications. The step from application design to implementation remains a demanding task.

Ensemble (Contronis (1997), Contronis (1998), Contronis (2001)), supports the design and implementation of message passing architectures, particularly MPMD and those demanding irregular or partially regular process topologies. Ensemble is mainly for abstract programming; design of application, instead of compiling the programs directly. Ensemble uses a tool to generate the abstract parallel programs into pure MPI code and then compiles the code into modular MPI components. In our work, we use

the SPMD technique with domain decomposition for parallel implementation, which help the update and synchronization, so communication cannot be changed easily. The SPMD technique employing domain decomposition implementation has flexible mapping strategy which provides automatic and manual mapping, but in Ensemble mapping can be done only manually.

The Nanothreads Programming Model (NPM) (Hadji et al., (2002)), is a programming model for shared memory multiprocessors. The NPM can integrate with MPI, used on distributed memory systems. The runtime system is based on a multilevel design that supports both the model (NPM and MPI) individually, but offers the capability to combine their advantages . Existing MPI codes can be executed without any changes and codes for shared memory machines be used directly, while the concurrent use of both models is easy. The major feature of the NPM runtime system is portable, as it is based exclusively on calls to MPI and Nthlib, a user-level threads library that has been ported to several operating systems. The runtime system supports the hybrid-programming model (MPI + OpenMP) as in Hu et al., (2000). NPM decomposes application into fine-grain tasks and executed in a dynamic multiprogrammed environment. The parallelizing compiler analyzes the source program in order to produce an intermediate representation, called the hierarchical task graph. The graph is used for the mapping of user tasks to the physical processors on runtime.

A different approaches can be employed for scheduling parallel applications on a cluster. Traditional static space-sharing by Zhu (1992), is a simple approach that involves finding enough idle computers in a cluster and mapping processes of a parallel application onto these computers. For instance, a parallel application needs to wait until enough idle computers are available before it can be started; otherwise, it has to sacrifice the level of its parallelism. A study of concurrent execution of parallel and sequential applications on a non-dedicated cluster was done by Andrzej et al., (2008).

Time-sharing is intrinsically supported in a cluster via local scheduling. In this case, the local scheduler is responsible for time-sharing of the Central Processing Unit (CPU) among all the processes which have been allocated to that computer. Processes from a parallel application can be placed into some or all of the computers in the cluster depending on the required parallelism. However, processes belonging to the same parallel application would not be guaranteed to execute at the same time across the computers in the cluster. Previous studies (Anglano, (2000), Wong et al., (1999)) have found that if the parallel application is communicative intensive, this uncoordinated scheduling of processes would lead to a great loss of performance in its execution since a process stalls when it communicates with a non-scheduler in the process. Furthermore, both Wong et al., (1999) and Strazdins & Uhlmann, (2004), have presented the results of concurrent execution of multiple parallel applications on a cluster using local scheduling, but the results are quite difference. Among the related work, Sahimi et al., (2002), resemble closely to our work in the sense that the concurrent execution of parallel and sequential applications on a heterogeneous and homogeneous cluster is proposed in order to utilize any wasted resources. However the approach taken by Sahimi et al., (2002) are quite different from ours. Then, Dou & Phan-Thien, (1997), use a parallel implementation of domain decomposition technique on MIMD parallel architectures using PVM platform only, testing lamina flow in cavity. Laurant (2001), uses a method for automatic placement of communications in SPMD parallelization adapted for static placement.

Performance models for explicit group methods and detailed study of their hypothetical implementation on two distributed memory multicomputer with different computational speed and communication bandwidth were investigated (Kok et al., (2008)). The bidirectional one-point model was used by Bhat et al., (2003), for fixed-size messages. Even if non-blocking multi-threaded communication libraries allow for

initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network". Experimental evidence of this fact has recently been reported by Saif & Parashar, (2004), who report that asynchronous MPI sends get serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2. Recently Schulz (2005), developed a toolset that can identify the critical part of MPI application, extract it and then produce a graphical representation for visualization.

On the parallel computing front Rathis et al., (2001), have proposed a parallel ADI solver for linear array of processors, Chan & Saied (1987), have implemented ADI scheme on hypercube, later Lixing et al., (1998) parallelized the ADI solver on multiprocessors. Several approaches to solve the Telegraph Equation numerically to determine stability have been exploited by Evans & Hassan (2003), and Alloy et al., (2007). Sun & Gustafson (1991), proposed a generalized speedup formula, and in our work we were able to show this formula is true using our parallel strategies. In relation to performance strategies implementation, a thorough study of speedup models together with their advantages is implemented in Sahni et al., (1996), and this show conformity to our implementation. Work done by Fan et al., (2003), on high-level abstractions for message-passing parallel programming, describe the Graph-Oriented Programming (GOP) model and environment for building and evaluating parallel applications. It is worth noting that of the related work Sahimi et al., (2002), used the AGE class of methods based on Brian variant (AGEB) for solving the heat equation on PVM clusters to assess performance in terms of speedup, efficiency and effectiveness. Compared to our implementation, we use the same domain decomposition on SPMD technique with PVM and MPI on a loosely distributed platform to assess speedup, efficiency and

effectiveness, and respective conformity was ascertained with improved linearity and closeness to unity.