A Software Platform for Solving PDEs on Distributed Systems: Implementation Issues and Performance Prediction

Chi-Chung Hui, Mounir Hamdi and Ishfaq Ahmad

Department of Computer Science, Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

This paper describes the implementation and performance of a parallel platform for solving partial differential equations (PDEs) on distributed systems. The platform has been implemented using PVM for a network of workstations. It allows the inclusion of a wide range of parameters and programming aids. The PDEs are specified in the form of finite difference equations. With a given set of parameters and a partitioning strategy, the platform provides facilities to record and predict the performance of an application before running it. The performance prediction model helps the user to identify the major bottlenecks of the platform such that by reducing them, the speedup can be improved. We also present analysis of various factors that can have drastic effect on the speedup, which allows the user to tune a number of parameters to maximize the performance.

1 Introduction

The use of cluster of networked workstations as a virtual parallel computer has become an effective and economical alternative to expensive supercomputers. Workstation cluster usually provides a powerful aggregate of computing power. Moreover, the emergence of high-bandwidth networks also improves the scalability of executing parallel applications.

We have implemented a parallel platform for solving time-dependent partial differential equations (PDEs) on distributed systems. Solving PDEs is generally regarded as one of the most computationally intensive tasks. PDEs are encountered in numerous problems in science and engineering which involve rates of change with respect to several independent variables [6]. For example, the heat equation deals with the conduction of heat in solids and fluids with respect to time [6]. Numerical weather prediction also requires the solution of parabolic PDEs to model the time-dependent climate behavior [8]. Sequential numerical methods for solving time-dependent PDEs have been explored extensively [4], [5]. On the other hand, only a few attempts have been made toward parallel solutions using either transputers [7] or distributed-memory MIMD machines [2]. However, these parallel solutions are specific to particular kinds of problems and are not general in nature.

Our platform can be used to solve any application that requires the solution of time-dependent PDEs. It also provides facilities to predict the performance of the given application accurately. Moreover, the user can tune various system parameters, employ various partitioning strategies and load balancing schemes to obtain maximal performance.

The rest of the paper is organized as follows. Section 2 presents the functional description of our parallel platform. Section 3 presents the user interface while Section 4 presents the parallel implementation of the platform respectively. Section 5 describes the experimental results and the performance prediction model. Finally, in Section 6 we present some concluding remarks.

2 Functional Description

The platform allows the specification of PDEs using the *finite difference method* in which the derivatives are approximated by difference quotients over small intervals [5]. Each *grid point* in the *domain* is given an initial value and is updated according to the finite difference equations. This regular relationship among the grid points is where the parallelization can be captured using a data-parallel computing paradigm [3].

The PVM system has been used in implementing the platform [1]. The platform views PVM as a parallel computing resource where individual processors communicate via message passing. The *single program multiple data* (SPMD) paradigm is adopted, in which the whole application consists of a *host process* and a number of *node processes* so that each processor is associated with a node process. The host process computes the data partition scheme and spawns the node processes while the node processes compute the results in different *regions* in the domain. The user is required to describe the solutions of the PDEs in the *application specification*. The parallel implementation consists of a number of steps, which include data partitioning, processor allocation, load balancing, data I/O, computation and communication. The platform estimates the performance of a given application through a number of sta-

tistical modules including computation time, communication time, processor idle time, total execution time, speedup and system overhead. If the result is not satisfactory, the user can redefine the partitioning scheme, load balancing strategy and/or the I/O methodology to reduce system overheads. The platform is decomposed into several components and each of them is modeled by a separate formula. Any change on a particular component does not affect the other components. This makes the platform more portable to different hardware configurations.

3 User Interface

The PDEs are specified in the *application specification*. The platform then parses the application specification and builds the executable programs. The application specification is divided into three categories which are the *parameter section*, the *definition section* and the *auxiliary section*. It is flexible enough to allow the user to define C-style parameters, variables and functions.

3.1 The Parameter Section

The parameter section allows the user to define all global data structures and constant parameters needed by the platform. These include the following:

Structure of the data point: A number of variables must be maintained at each data point such as the temperature in the heat equation. These variables are specified in the structure *point*.

Initial data at the grid points: The names of the data files containing the initial data for all grid points in the domain must be supplied.

Dimensionality and resolution of the domain: The dimensionality are defined by *DIMSIZEn* where n is the dimension of the problem. The resolution of the domain is defined by *DIMINFO* with the following format

$$DIMINFO = \{d_1, d_2, \ldots, d_n\}$$

where d_i is the number of grid points in the i-th. dimension. In the current implementation, the value of n is at most 3.

Data Dependency: In order to calculate the value of a grid point, say $P = (p_0, p_1)$ at time t, the values of some surrounding grid points in the previous time steps must be referred. This dependency is specified by two parameters *LEVEL* and *CALINFO*. *LEVEL* specifies the number of the previous time steps in which the domain must be kept in order to calculate P at time t. Given a particular value of *LEVEL*, the domain at time steps (t - *LEVEL*), ... , and (t - 1) are kept at time step t. *CALINFO* dictates the range of the surrounding grid points in the previous time steps that point P requires. It has the following format

$$CALINFO = \{R_1, R_2, \ldots, R_n\}$$

where R_i is an integer that specifies the maximal distance in the i-th. dimension of the required grid points from P. For instance, if *CALINFO* = {1,1}, nine points are needed as shown in Figure 1.

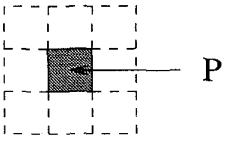

Figure 1: The nine points needed by P when *CALINFO* = {1,1}.

Checkpointing information: The user may want to see intermediate results at different checkpoints. Two parameters *CPSIZE* and *CP* serve this purpose. *CPSIZE* is an integer depicting the number of checkpoints, while *CP* has the following format

$$CP = \{c_1, c_2, \ldots, c_{CPSIZE}\} .$$

Intermediate results are stored at c_i for all i. The platform terminates when the time step reaches c_{CPSIZE}.

Output files: The final results are written to a number of files. These files are generated both by the host process and the node processes for providing the final results as well as error messages in case of abnormal execution.

3.2 The Definition and the Auxiliary Sections

In the definition section, the user must provide four functions to define the computation and I/O procedures, including `read_point()`, `write_point()`, `init_compute()` and `compute()`. The platform invokes `read_point()` to input the initial grid points and `write_point()` to output the final results. Before the computation process begins, it calls `init_compute()` to initialize the variables. On the other hand, it calls `compute()` to calculate the values of the new grid points. The auxiliary section contains user supplied sub-programs that are required by the functions defined in the definition section. The sub-programs should be self-contained.

4 Implementation

The platform carries out a number of tasks to solve PDEs, which include processor allocation, domain partitioning, load balancing, computation, communication and disk I/O. It distributes the regions among the processors and monitors the flow of the grid points across processor boundaries. The reader is referred to [9] for detailed descriptions of the platform implementation.

Two algorithms are implemented to compute the values of the grid points in the domain. They are the *two-phase algorithm* and the *pre-computation algorithm*.

To calculate the values of grid points at time t, the two-phase algorithm performs alternating communication and

computation phases. In the communication phase, the node processes wait until all boundary points in time (t - $LEVEL$) to time (t - 1) are arrived. In the computation phase, the node processes compute the values of the grid points at time t. The boundary grid points at time t are then sent to its neighbor node processes at the beginning of the next communication phase. Since there is no overlap between the computation phase and the communication phase, the node process must wait until it receives all the required grid points from its neighbors. In an environment where workstations are connected by a shared bus, the communication time tends to become a dominant factor, and algorithms like this may become inefficient.

To reduce the waiting time in the two-phase algorithm, we implement the pre-computation algorithm: For each node process, if the boundary grid points from time (t - $LEVEL$) to time (t - 1) have not yet arrived, it computes the grid points in time t which do not need the boundary grid points. This process is repeated until no data can be pre-computed anymore.

5 Performance Prediction

The platform provides facilities to record and predict the performance of a given application. The performance prediction model helps the user to identify the major bottlenecks of the platform such that by reducing them, the speedup can be improved. A set of equations are derived in this section to predict the performance of the platform. Experiments were conducted to obtain the timings of various components of the total elapsed times using different number of processors. Regression analyses were then carried out to model the experimental sample data with minimal errors.

Our distributed computing environment consists of 20 SPARCstation IPX workstations connected by an Ethernet network. A single file system is being shared by all workstations in the network. The 2-D heat equation was used to perform experiments in this paper. It has the format

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

with the finite difference equation

$$u_{i,j}^{k+1} = r_x u_{i-1,j}^{k} + r_x u_{i+1,j}^{k} + (1 - 2r_x - 2r_y) u_{i,j}^{k} + r_y u_{i,j-1}^{k} + r_y u_{i,j+1}^{k}$$

where $r_x = \Delta t / (\Delta x)^2$, $r_y = \Delta t / (\Delta y)^2$ and $u_{i,j}^{k}$ denotes the temperature at grid point (i,j) at time $t_k = t_0 + k \times \Delta t$. The number of time steps was set to 500, the size of the domain was chosen to be 3800×100 and the temperature outside the domain was defined to be zero. Notice that the domain was partitioned into region along one dimension only.

5.1 Modeling

In this section, different time symbols have different meanings: T stands for the total elapsed time, F stands for the fixed elapsed time that is independent of the number of processors, D stands for the computation elapsed time that can be shared by different node processes, and M stands for the elapsed time that increases as the number of node processes increases. The main purpose of these definitions is to make the equations more readable.

The total elapsed time for solving the PDEs consists of the time spent in the host process as well as the node processes. The host process is responsible for performing data partitioning and setting up the node processes, while the node processes perform disk I/O, computation and communication. The total elapsed time using N node processes for s time steps can be described by the following equation

$$T_{total}(N, s) = T_{setup}(N) + T_{io}(N) + T_{comp}(N, s) + T_{comm}(N, s)$$

where $T_{setup}(N)$ is the time used by the host process in setting up the node processes and exchanging control information between the node processes, $T_{io}(N)$ is the time spent by the node processes in reading the initial data domain and writing final results, $T_{comp}(N,s)$ is the time spent by the node processes in setting up the buffers and computing the results, and $T_{comm}(N,s)$ is the time spent by the node processes in processing and waiting the messages. Here, $T_{total}(N,s)$ and $T_{setup}(N)$ are measured in the host process while $T_{io}(N)$, $T_{comp}(N,s)$ and $T_{comm}(N,s)$ are the average values measured at the node processes.

Host Setup Time: The host setup time using N nodes is modeled by the equation

$$T_{setup}(N) = F_{setup} + M_{setup} \times N \ .$$

From the curves shown in Figure 2, F_{setup} and M_{setup} are found to be 0.55s and 0.23s respectively, which are the same for both the two-phase algorithm and the pre-computation algorithm. The major task that constitutes T_{setup} is setting up the node processes which is relatively constant for different applications.

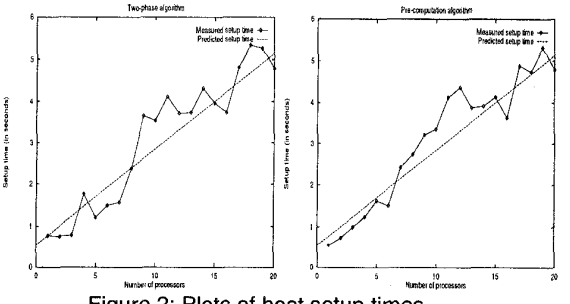

Figure 2: Plots of host setup times versus the number of processors

Disk I/O time: The disk I/O time is represented by the equation

$$T_{io}(N) = F_{io} + \frac{D_{io}}{N} - M_{io} \times N$$

where F_{io} is the operating system overhead which includes the time to maintain the file pointers and the file buffers, and D_{io} is the time that is used to perform disk I/O on the data domain sequentially. Interestingly, there is a reduction factor M_{io} as N increases. This is due to the overlapping of the disk I/O processing among the processors. Although the file server can only process one I/O request at anytime, the N node processes can buffer, read and write the file system simultaneously.

From the curves in Figure 3, F_{io}, D_{io} and M_{io} are found to be equal to 9.403s, 6.3418s and 0.172s for the two-phase algorithm, and 8.4867s, 6.144s and 0.1343s for the pre-computation algorithm. The curves are close to

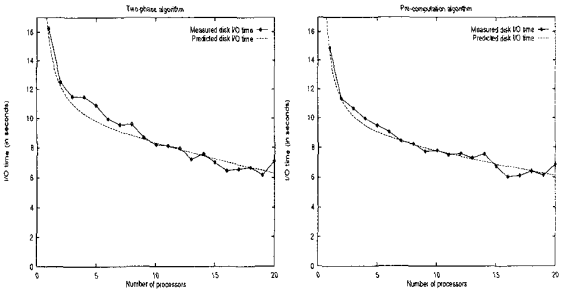

Figure 3: Plots of disk I/O times versus the number of processors

each other, implying that the two algorithms have no significant difference in disk I/O.

Computation Time: The computation time using N nodes for s time steps is represented by the equation

$$T_{comp}(N, s) = F_{fixedcomp} + \left(F_{comp} + \frac{D_{comp}}{N} \right) \times s$$

where $F_{fixedcomp}$ is the system overhead for manipulating the region buffers and setting up the variables, F_{comp} is a

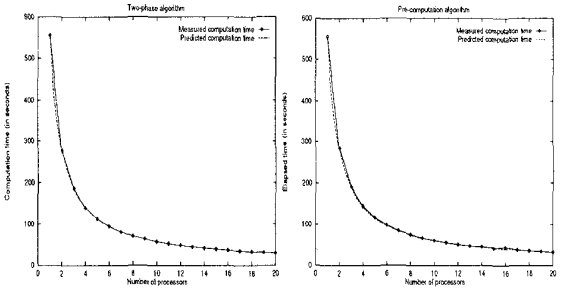

Figure 4: Plots of computation times versus the number of processors

fixed overhead for each time step, and D_{comp} is the time used in performing calculation of the region for each time step. In our experiments, s is equal to 500. From Figure 4, $F_{fixedcomp}$, F_{comp} and D_{comp} are equal to 0.122s, 0.004113s and 1.1062s for the two-phase algorithm, and are equal to 0.2635s, 0.009516s and 1.1005s for the pre-computation algorithm. Therefore, it can be shown that the pre-computation algorithm incurs additional overhead over the two-phase algorithm.

Communication Time: The communication time using N nodes for s time steps is given by the equation

$$T_{comm}(N, s) = (M_{PVM} + s \times M_{comm}(N)) \times f_{comm}(N)$$

where $M_{comm}(N)$ describes the elapsed time used by each internal node process[1] in processing and waiting for the messages for one time step, and M_{PVM} is a constant describing the overheads that the PVM system incurs in exchanging messages. Under the shared bus architecture, the value of $M_{comm}(N)$ is proportional to the volume of the boundary grid points to be sent. In our experiment, each internal node process sends and receives 200 points to and from its neighboring node processes on each time step, and so $M_{comm}(N)$ is a constant independent of the value of N. For all applications, $M_{comm}(N)$ can be estimated by the data partitioning scheme accurately.

The function $f_{comm}(N)$ is incorporated to consider the effect of the global boundary surrounding the data domain. It is defined as the ratio of the cross-section area of the data partitioning scheme over the sum of the area of the global boundary plus the cross-section area. In the experiment,

$$f_{comm}(N) = \frac{N-1}{N}$$

since the temperature outside the domain is always zero and the leftmost and rightmost node processes have only one neighbor. In general, $f_{comm}(N)$ is application specific. As depicted in Figure 5, M_{PVM} is equal to 0.1311s and

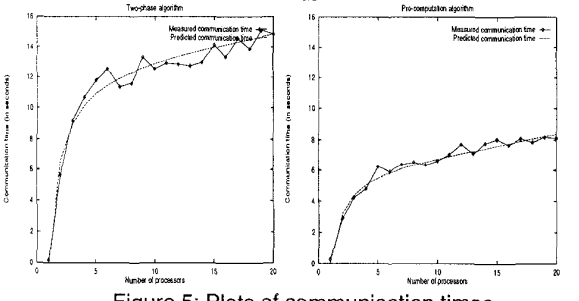

Figure 5: Plots of communication times versus the number of processors

1. An internal node is not adjacent to the global boundary, and thus needs to exchange all the boundary points with its neighbour nodes.

$M_{comm}(N)$ is confirmed to be constants for both algorithms. It is equal to 0.026s for the two-phase algorithm and is equal to 0.01238s for the pre-computation algorithm. We can see that $M_{comm}(N)$ is substantially smaller for the pre-computation algorithm, which means that the pre-computation algorithm can successfully reduce the communication time by overlapping with the computation time.

5.2 Performance Analysis and Validation

The speedup for N processors and s time steps is defined as

$$S(N,s) = \frac{T_{total}(1,s)}{T_{total}(N,s)} .$$

The corresponding speedup plots are shown in Figure 6.

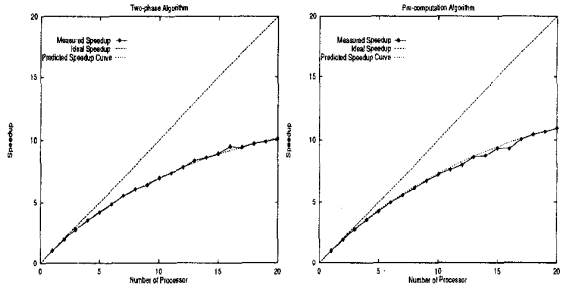

Figure 6: Plots of speedup versus the number of processors.

As shown in the diagrams, the pre-computation algorithm has higher speedup. Although it incurs additional computational overhead due to buffer management and variable set up, an overall gain can be obtained as long as the saving in communication time is larger. As a result, it can be claimed that the pre-computation scheme is efficient in improving the speedup for problems having sufficiently large communication requirements.

In order to validate the performance prediction model, extra experiments were carried out for the pre-computation case using 100 time steps. As shown in Figure 7, the experimental and predicted results closely match each other.

5.3 Impact of the Number of Processors

Now, we want to use our prediction model to find the number of processors that yields the maximal speedup for the equation. $T(N,s)$ can be rewritten as

$$T(N,s) = \alpha(s) + \beta(N,s) + \gamma(N,s)$$

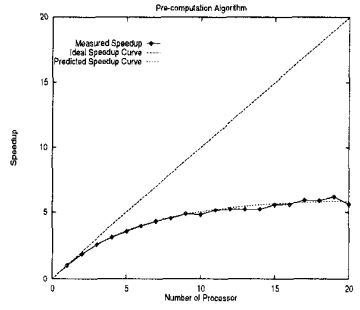

Figure 7: Plot of speedup versus number of processors (pre-computation case with s = 100).

where

$$\alpha(s) = F_{setup} + F_{io} + F_{fixedcomp} + F_{comp} \times s,$$

$$\beta(N,s) = \frac{D_{io} + D_{comp} \times s}{N}$$

and

$$\gamma(N,s) = (M_{setup} - M_{io} + M_{PVM} \times f_{comm}(N)) \times N + M_{comm}(N) \times s \times f_{comm}(N) .$$

The above three functions represent different aspects of the total elapsed time. The effects of these three functions are shown in Figure 8. Obviously, $\alpha(s)$ does not vary

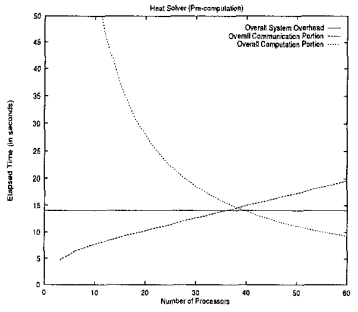

Figure 8: Plot of $\alpha(500)$, $\beta(x, 500)$ and $\gamma(x, 500)$ versus number of processors (pre-computation case).

with N, $\beta(N,s)$ decreases with N indicating that the computations are shared by the node processes, whereas $\gamma(N,s)$ increases with N indicating that the communication cost increases as the number of processors increases.

$N_{optimal}$, the optimal value of N such that the platform produces the minimum total elapsed time is obtained when

$$T_{total}(N_{optimal}, s) \leq T_{total}(N_{optimal} + 1, s) .$$

Since $\alpha(s)$ is a constant on N, $\beta(N,s)$ is a decreasing function of N and $\gamma(N,s)$ is an increasing function of N, Equation 13 is satisfied when

$$\beta(N_{optimal}, s) = \gamma(N_{optimal}, s) .$$

For the heat equation, this is a quadratic equation and can be solved without difficulty. When s = 500, for instance, $N_{optimal} \approx 37$ and the maximum speedup attainable is $S(37, 500) \approx 13.13$.

5.4 Impact of the Number of Time Steps

We want to obtain the theoretical speedup when the problem size of the heat equation is infinitely large and infinite workstations are available. The speedup with infinite number of time steps can be deduced as

$$\begin{aligned} S(N,\infty) &= \lim_{s\to\infty} S(N,s) \\ &= \frac{F_{comp} + D_{comp} + M_{comm}(1) \times f_{comm}(1)}{F_{comp} + \frac{D_{comp}}{N} + M_{comm}(N) \times f_{comm}(N)} \\ &\le N \ . \end{aligned}$$

Due to the non-negligible F_{comp}, $M_{comm}(N)$ and $f_{comm}(N)$, the speedup curves converge as s increases.

The ultimate speedup given infinite number of processors and time steps is

$$\begin{aligned} S(\infty,\infty) &= \lim_{N\to\infty} S(N,\infty) \\ &= \frac{F_{comp} + D_{comp} + M_{com}(1) \times f_{comm}(1)}{F_{comp} + \frac{D_{comp}}{N} + M_{comm}(\infty) \times f_{comm}(\infty)} \end{aligned}$$

where

$$M_{comm}(\infty) = \lim_{N\to\infty} M_{comm}(N)$$

and

$$f_{comm}(\infty) = \lim_{N\to\infty} f_{comm}(N) \ .$$

For the heat equation, $S(\infty,\infty)$ converges to a constant. For instance, the ultimate speedup for the pre-computation algorithm is

$$\begin{aligned} S(\infty,\infty) &= \frac{0.004113 + 1.1062 + 0.012380}{0.004113 + 0.012381} \\ &= 67.32 \ . \end{aligned}$$

5.5 Major Bottleneck and Possible Improvement

In Figure 8 it is known that the major overhead differs for different values of N. For $N < 37$, $\alpha(s)$ is larger than $\gamma(N,s)$. Therefore, better speedups can be achieved by reducing $\alpha(s)$ instead of $\gamma(N,s)$. Hence, it may be concluded that the fixed system overhead also limits the speedup substantially, especially when the number of workstations is small. Consequently, the programmer should be very careful in tuning the platform even when a faster communication mechanism is available.

6 Conclusion

A parallel platform for solving time-dependent partial differential equations is designed and implemented. Detailed investigations of all the time components that make up the total elapsed time have been approximated experimentally and closely verified by a performance prediction model. The performance prediction model helps the programmer to estimate the performance and identify the major bottlenecks of the platform for a given problem. In a networked workstation environment, it is always believed that the major bottleneck that limits the speedup is the communication overhead. However, in our implementation, it is found that the major bottleneck of the platform is the fixed software overhead which includes the time to setup the system and the buffers. By reducing this overhead, it is possible to achieve better speedups than by just improving the communication overhead itself. This effect is significant especially when using a small number of workstations.

References

[1] G. A. Geist and V. M. Sunderam, "Network-based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, 1992, pp. 293-311.

[2] Z. Cvetanovic, E.G. Freeman and C. Nofsinger, "Efficient Decomposition and Performance of Parallel PDE, FFT, Monte Carlo Simulations, Simplex and Sparse Solvers," *Proceedings of Supercomputing '90*, Nov., 1990, pp. 465-474.

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors, Vol. I: General Techniques and Regular Problems*, Prentice Hall, 1988.

[4] M.A.H. MacCallum, "An Ordinary Differential Equation Solver for REDUCE," *International Symposium ISSAC'88*, pp. 115-123.

[5] J. Noye, *Finite Difference Methods for Partial Differential Equations, Numerical Solutions of Partial Differential Equations*, North Holland Pub. Co., pp.3-137.

[6] M.A. Pinsky, *Introduction to Partial Differential Equations with Applications*, McGraw-Hill Publishing co., 1984.

[7] E. Verhulst, "A Prototype of a User Friendly Partial Differential Equation Solver on a Transputer Network," *Proceedings of the User 1 Working Conference*, 1988, pp. 232-239.

[8] G.R. Wightwick, L.M. Leslie, "Parallel Implementation of a Numerical Weather Prediction Model on a RISC System/6000 Cluster," *Fifth Australian Supercomputing Conference*, Oct. 12, 1992, pp.135-142.

[9] C.-C. Hui, G. K.-K. Chan, M. M.-S. Yuen, M. Hamdi and I. Ahmad, "Solving Partial Different Equations on a Network of Workstations", in *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, Aug. 2-5, 1994, pp. 194-201.