International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 4– No.11, December 2012 – www.ijais.org

Parallelization of 3-D ADI Scheme on Telegraph Problem using Domain Decomposition with PVM

Ewedafe Simon Uzezi
Faculty of Computing and IT Baze University
Abuja, Nigeria

Rio Hirowati Shariffudin
Institute of Mathematical Sciences
Faculty of Science
University of Malaya

ABSTRACT

A parallel implementation of 3-D Alternating Direction Implicit (3-D ADI) method on 3-D Telegraph problem on a distributed computing environment through Parallel Virtual Machine (PVM) is reported. The numerical method is implicit and is based on a splitting strategy which is applied alternately at each half time step. The parallelization is implemented by a Domain Decomposition (DD) strategy on a distributed system with Single Program Multiple Data (SPMD) model on a PVM platform. The parallelization strategy and performance are discussed. Different strategies to improve the computational efficiency are proposed.

General Terms

Parallel Computing, Numerical Computing, Mathematical Computation.

Keywords

Telegraph, 3-D ADI, PVM, Domain Decomposition, and Parallelization.

1. INTRODUCTION

Due to the ever-increasing clock frequency and the aggressively shrinking feature sizes of the very large scale integration technology, robust power distribution network is crucial to ensure the quality of power delivery. This makes the issues of parallel and distributed applications very important. Parallel computing environments based on distributed computing has become effective and economical platforms for high performance computing by providing controlled access to a much larger and richer computational resource [20]. Distributed systems can increase application performance by significant amount and the incremental enhancement of a network based concurrent computing environment is usually straight forward because of the availability of high bandwidth networks [2, 3]. Attempts have also been made towards parallel solutions on distributed memory MIMD machines. Large scale computational scientific and engineering problem, such as time dependent and 3D flows of viscous elastic fluids, required large computational resources with a performance approaching some tens of giga (10^9) floating point calculations per second, 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 [4]. Relevant data need to be passed from processor to processor through a message passing mechanism [12, 6, 19, 16]. The natural programming style under a distributed system is therefore the Multiple Instructions Multiple Data (MIMD) [22]. The basic idea is to split a program into a few smaller tasks and to allocate these tasks to several processors to be executed simultaneously. Thus the total execution time can be reduced to just a fraction of that of a uni-processor computer. Generally, there are three issues relating to parallel computing: (1) The first issue is how to break up the program into smaller tasks, and how to properly sequence these tasks, (2) The second is the communication between processors, which is necessary because some intermediate results have been exchanged among these processors and (3) The issue of the synchronization of computations on different processors. All these three problems strongly influence the performance of the parallel computation.

Methods by which processors exchange information; parallel computers can either be of shared memory or distributed memory (message-passing) architectures. With the shared memory architectures, there is a global share memory which can be accessed by all processors. Processors communicate by writing into and reading from the global memory. This mode of communication is convenient in terms of program writing, but memory access by different processors can generate potential memory conflicts. With distributed memory architectures, each processor has its local memory, and processors communicate through an interconnection network. Inefficient communication is the main problem for this type of communication. To help with the program development under a distributed computing environment, a number of software tools have been developed: PVM, Theoretical Chemistry Message Passing Tool Kit (TCMSG), Parasoft Express (EXPRESS), Network Linda (LINDA), and Message Passing Interface (MPI). PVM [13, 14] is chosen here since it has a large user group with possibly the best support for heterogeneous environment.

Parallel algorithms have been implemented for the finite difference method [15], discrete eigen functions method in [1] and [8] use the AGE method on 1-D Telegraph problem. Boundary element method and the finite volume method using DD have also been implemented. In addition, time and functional decompositions have also been used in parallel implementations.

In this paper, we present a parallel program for the solution of the 3-D Telegraphic problem using the 3-D ADI method. We employed the DD strategy, and mapped it onto a distributed computing environment with a synchronous iteration and a message passing master – slave construction. The program was executed in a distributed memory system under PVM. The numerical method is based on the double sweep methods of Peaceman and Rachford (DS-PR) [19]. We computed some examples to test the parallel algorithm. The effects of the various parameters on the performance of the algorithm 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 partial differential equations (PDE) for various applications [24, 25]. This paper is organized as follows: Section 2 introduces the model for the 3-D Telegraphic problem and introduces the 3-D ADI method. Section 3 introduces the parallel implementation. Section 4 introduces the results and discussion. Finally, a conclusion is included in section 5.

1.1 Previous Research Work

The ADI method for the partial differential equations (PDEs) proposed by Peaceman and Rachford [16] has been widely used for solving algebraic systems resulting from finite difference method analysis of PDEs in several scientific and engineering applications. On the parallel computing front, Rathish Kumar, et. al., have proposed a parallel ADI solver for linear array of processors. Chan and Saied have implemented ADI scheme on hypercube. Later Lixing et. al., have parallelized the ADI solver on multiprocessors. The ADI method in [18] has been used for solving heat equation in 2-D. Other works on parallel implementation of 2-D Telegraph problem on cluster systems have been done in [10, 11]. Hence, parallelization of the ADI method has been tried in [22]. Several numerical approaches dealing with telegraphic equation problems have been carried out in [1, 8] and [9]. In [15] the unconditional stability of the alternating difference schemes has similarity to our scheme and shows that the unconditional stability application is useful to its speedup and efficiency as studied. Our implementation compared to others is a way of proofing stability and convergence in parallel platform of a distributed system. We also note the various constant improvement on speedup, efficiency performance analysis in [25] using the overlapping domain decomposition method.

2. TELEGRAPH EQUATION

the frequency and time domains. A number of iterative methods are developed in the literature to solve the Telegraph equation using iterative solution [18]. Some of these iterative schemes are employed in various parallel platforms [2, 20]. The speed of convergence of iterative scheme is examined for the synchronous communication approaches in parallel environment. We consider the second order Telegraph equation:

$$\frac{\partial^2 v}{\partial t^2} + a\frac{\partial v}{\partial t} - \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) = 0 \qquad (2.1)$$

where $a = RC + GL$, let $\Delta x, \Delta y\ and\ \Delta z$ be the grid spacing in the x, y, z and t directions, where $\Delta x = \Delta y = \Delta z = 1/m$, m is a positive integer. Hence, we can solve (2.1) by extending the 1-D simple implicit finite difference method [8] of the Telegraph equation to the above 3-D Telegraph equation, (2.1) becomes:

$$\frac{v_{i,j,k}^{n+1} - 2v_{i,j,k}^{n} + v_{i,j,k}^{n-1}}{(\Delta t)^2} + a\frac{v_{i,j,k}^{n+1} - v_{i,j,k}^{n-1}}{2\Delta t} - \left\{\begin{array}{l} \dfrac{v_{i+1,j,k}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i-1,j,k}^{n+1}}{(\Delta x)^2} \\ + \dfrac{v_{i,j+1,k}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i,j-1,k}^{n+1}}{(\Delta y)^2} \\ + \dfrac{v_{i,j,k+1}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i,j,k-1}^{n+1}}{(\Delta z)^2} \end{array}\right\} = 0 \qquad (2.2)$$

although this simple implicit scheme is unconditionally stable, we need to solve a heptadiagonal system of algebraic equations at each time step. Therefore, the computational time is extremely huge.

2.1 3-D ADI Method on the Telegraph Problem

In this section, we derive the 3-D ADI method of the simple implicit finite difference method by using a general ADI procedure [18] extended to (2.1). The ADI method is a well-known method for solving the partial differential equation (PDE). The main feature of ADI is to sweep directions alternatively. In contrast to the standard finite-difference formulation with only one iteration to advance from the *nth* to (*n + 1*)*th* time step, the formulation of the ADI method requires multilevel intermediate steps to advance from the *nth* to (*n + 1*)*th* time step. Equation (2.2) can be rewritten as:

$$\left(I + \sum_{m=1}^{3} A_m\right) v_{i,j,k}^{n+1} - 2C_o v_{i,j,k}^{n} + C_1 v_{i,j,k}^{n-1} = 0 \qquad (2.3)$$

where the operators of *I*, *A_ms*, and the constants of C_o, C_1 are define as:

$$I v_{i,j,k}^{n} \equiv v_{i,j,k}^{n} \qquad (2.4)$$

$$A_1 v^n_{i,j,k} \equiv -\rho_x \left(v^n_{i+1,j,k} - 2v^n_{i,j,k} + v^n_{i-1,j,k} \right) \qquad (2.5)$$

$$A_2 v^n_{i,j,k} \equiv -\rho_y \left(v^n_{i,j+1,k} - 2v^n_{i,j,k} + v^n_{i,j-1,k} \right) \qquad (2.6)$$

$$A_3 v^n_{i,j,k} \equiv -\rho_z \left(v^n_{i,j,k+1} - 2v^n_{i,j,k} + v^n_{i,j,k-1} \right) \qquad (2.7)$$

$$C_o \equiv \frac{1}{(\Delta t)^2} \bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \qquad (2.8)$$

$$C_1 \equiv \left(\frac{1}{(\Delta t)^2} - \frac{a}{2\Delta t} \right) \bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \qquad (2.9)$$

the constant of ρ_x , ρ_y and ρ_z are:

$$\rho_x = \frac{b}{(\Delta x)^2} \bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \qquad (2.10)$$

$$\rho_y = \frac{b}{(\Delta y)^2} \bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \qquad (2.11)$$

$$\rho_y = \frac{b}{(\Delta y)^2} \bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \qquad (2.12)$$

Table 1. The 3-D ADI Algorithm

```
The 3-D ADI Algorithm

Input = $v^n_{i,j,k}, v^{n-1}_{i,j,k} \ \forall i,j,k$

Output = $v^{n+1}_{i,j,k} \ \forall i,j,k$

Begin

  Sub-Iteration 1:
    $-\rho_x v^{n+1(1)}_{i+1,j,k} + (1+2\rho_x) v^{n+1(1)}_{i,j,k} - \rho_x v^{n+1(1)}_{i-1,j} = -(A_2 + A_3) v^{n+1(*)}_{i,j,k} + (2C_o v^n_{i,j,k} - C_1 v^{n-1}_{i,j,k}) \ \forall i,j,k$
  Sub-Iteration 2:
    $-\rho_y v^{n+1(2)}_{i,j+1,k} + (1+2\rho_y) v^{n+1(2)}_{i,j,k} - \rho_y v^{n+1(2)}_{i,j-1,k} = v^{n+1(1)}_{i,j,k} + A_2 v^{n+1(*)}_{i,j,k}. \quad \forall i,j,k$
  Sub-Iteration 3:
    $-\rho_z v^{n+1(3)}_{i,j,k+1} + (1+2\rho_z) v^{n+1(3)}_{i,j,k} - \rho_z v^{n+1(3)}_{i,j,k-1} = v^{n+1(2)}_{i,j,k} + A_3 v^{n+1(*)}_{i,j,k}. \quad \forall i,j,k$
End
```

and set

$$v^{n+1(*)}_{i,j,k} = 2v^n_{i,j,k} - v^{n-1}_{i,j,k} \qquad (2.13)$$

which is a prediction of $v^{n+1}_{i,j,k}$ by the extrapolation method. Then splitting (2.3) by using an ADI procedure as in [18], we get a set of recursion relations as follows:

$$(I + A_1) v^{n+1(1)}_{i,j,k} = -(A_2 + A_3) v^{n+1(*)}_{i,j,k} + (2C_0 v^n_{i,j,k} - C_1 v^{n-1}_{i,j,k}) \qquad (2.14)$$

$$(I + A_2) v^{n+1(2)}_{i,j,k} = v^{n+1(1)}_{i,j,k} + A_2 v^{n+1(*)}_{i,j,k} \qquad (2.15)$$

$$(I + A_3) v^{n+1(3)}_{i,j,k} = v^{n+1(2)}_{i,j,k} + A_3 v^{n+1(*)}_{i,j,k} \qquad (2.16)$$

where $v^{n+1(1)}_{i,j,k}, v^{n+1(2)}_{i,j,k}$ are the intermediate solutions and the desired solution is $v^{n+1}_{i,j,k} = v^{n+1(3)}_{i,j,k}$. Finally, expanding A_1, A_2 and A_3 on the left side of (2.14) and (2.16), we get the 3-D ADI algorithm as in Table 1.

3. PARALLEL IMPLEMENTATION

3.1 The Parallel Platform

The implementation was on a distributed computing environment (Armadillo Generation Cluster) consisting of 16 Intel Pentium at 1.73GHZ and 0.99GB RAM. Communication is through a fast Ethernet of 100 MBits per seconds connected through fast Ethernet running Linux. The cluster performance has high memory bandwidth with a message passing supported by PVM which is public-domain software from Oak Ridge National Laboratory [13]. PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. PVM consist of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and for changing the configuration of machines. PVM supports program executed on each machine in a user-configurable pool, and present a unified, general, and powerful computational environment for concurrent applications. The program, written in Fortran, C, or C++, are provided access to PVM through calling PVM library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers or rendezvous. PVM is ideally suited for concurrent applications composed o many interrelated parts, and is very useful for the study of large-scale parallel computation.

3.2 Domain Decomposition

The parallelization of the computations is implemented by means of grid partitioning technique [5]. The computing domain is decomposed into many blocks with reasonable geometries. Along the block interfaces, auxiliary control volumes containing the corresponding boundary values of the neighboring block are introduced, so that the grids of neighboring blocks are overlapped at the boundary. When the domain is split, each block is given an I-D number by a “master” task, which assigns these sub-domains to “slave” tasks running in individual processors. In order to couple the sub-domains’ calculations, the boundary data of neighboring blocks have to be interchanged after each iteration. The calculations in the sub-domains use the old values at the sub-domains’ boundaries as boundary conditions. This may affect the convergence rate; however, because the algorithm is implicit, the blocks strategy can preserve nearly same accuracy as the sequential program.

3.3 Parallel Algorithm

The algorithm can be transformed to master-slave model by sending out the computing tasks on each block to each processor in the Armadillo Generation Cluster system. The master task reads in the input data file, generates the grid data, initializes the variables, and sends all the data and parameters to the slaves. It then sends a block 3-D to each slave process, which in turn computes the coefficients of the relevant equations and solves for the solution of this block. This solution is then sent back to the master task and this processor wait for the next task. The master task receives the solution results from the slaves sequentially in an asynchronous manner, rearranges the data, calculates the global residuals of each equation, and determines if convergence has been reached. If the convergence has not been reached, the current solution vector is sent to all slaves, and a new iteration is started. Therefore, all the variables stored in the local memory of slaves are updated at every iteration. The following pseudo codes summarize the algorithm for the master and the slave program:

Master Program

```
Read in the input data
Compute mesh data
Initialize the variables
Enroll master program in pvm
Startup a slave program on each processor in the pvm farm
Send mesh data to all slave programs
Loop over iteration
    Set block numbers and it value
    Set the block index to it (ip = it)
    Do while (slaves working on ip < n blocks)
        if (ip < nblocks) then
            do (for each processor)
                if (processor is idle) then
                    ip = ip + 1
                    Send a message to the idle processor to
                    Compute a solution on block ip
                end if
                if (ip = nblocks) break
            end do
        end if
        wait here for a result from slave program
        get block solution and residual of equation
    end do while
    Global residual calculation
    Convergence check
    if (solution converged) break out loop
    send the current solution to all slave programs
end loop
Terminate all slave programs
Leave pvm
Time calculation
Save data file
Terminate the program
```

Slave Program

```
Enroll slave program in pvm
Do (forever)
    Wait for a message from the master program
    Case:
        GET MESH DATA
            Get mesh data from the master program
            Break
        DO DOMAIN DECOMPOSITION
            Break
        GET CURRENT SOLUTION
        Get the current solution vector
            Break
        DO CALCULATION
            Get the block id, ip
            Do calculation on block ip
            Solve for all variables equation on block ip
            Return the solution over ip to the master program
            Break
        FINISH UP
        Leave pvm
        Terminate the program
    End case
End do
```

3.4 Speedup and Efficiency

The performance metric most commonly used is the speedup and efficiency which gives a measure of the improvement of performance experienced by an application when executed on a parallel system [21]. Speedup is the ratio of the serial time to the parallel version run on N processors. Efficiency is the ability to judge how effective the parallel algorithm is expressed as the ratio of the speedup to N processors.The concept of speedup has yet to find a widely accepted definition. In traditional parallel systems it is widely define as:

$$S(n) = {T(s)}/{T(n)}, \qquad E(n) = {S(n)}/{n} \tag{3.1}$$

where *S(n)* is the speedup factor for the parallel computation, *T(s)* is the CPU time for the best serial algorithm, *T(n)* is the CPU time for the parallel algorithm using N processors, *E(n)* is the total efficiency for the parallel algorithm. However, this simple definition has been focus on constant improvements. A generalized speedup formula is the ratio of parallel to sequential execution speed. A thorough study of speedup models, together with their advantages and disadvantages, is presented by Sahni [23], and observed that speedup is normally defined as the execution time of the best sequential algorithm also known as absolute speedup, therefore implying that the sequential and parallel might be different. A different approach known as relative speedup, considers the parallel and sequential algorithm to be the same. While the absolute speedup calculates the performance gain for a particular problem using any algorithm, relative speedup focuses on the performance gain for a specific algorithm that solves the problem. The total efficiency is usually decomposed into the following equations

$$E(n) = E_{num}(n)E_{par}(n)E_{load}(n), \tag{3.2}$$

where E_{num}, is the numerical efficiency, represents the loss of efficiency relative to the serial computation due to the variation of the convergence rate of the parallel computation. E_{load} is the load balancing efficiency, which takes into account the extent of the utilization of the processors, and E_{par} is the parallel efficiency, which is define as the ratio of CPU time taken on one processor to that on N processors. The parallel efficiency and the corresponding speedup are commonly written as follows:

$$S_{par}(n) = {T(1)}/{T(n)}, \quad E_{par}(n) = {S_{par}(n)}/{n} \tag{3.3}$$

the parallel efficiency takes into account the loss of efficiency due to data communication and data management owing to domain decomposition. The CPU time for the parallel computations with N processors can be written as follows:

$$T(n) = T_m(n) + T_{sd}(n) + T_{sc}(n) \tag{3.4}$$

where $T_m(n)$ is the CPU time taken by the master program, $T_{sd}(n)$ is the average slave CPU time spent in data communication in slaves, $T_{sc}(n)$ is the average CPU time expressed in computation in slaves. Generally,

$$T_m(n) = T_m(1), \quad T_{sd}(n) = T_{sd}(1), \quad T_{sc}(n) = {T_{sc}(1)}/{n}, \tag{3.5}$$

therefore, the speedup can be written as:

$$S_{par}(n) = \frac{T(1)}{T(n)} = \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1) + T_{sc}(1)/n} < \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1)} \tag{3.6}$$

where $T_{ser}(1) = T_m(1) + T_{sd}(1),$ which is the part that cannot be parallelized. This is called Amdahl's law, showing that there is a limiting value on the speedup for a given problem. The corresponding efficiency is given by:

$$Epar(n) = \frac{T(1)}{nT(n)} = \frac{Tser(1)+Tsc(1)}{nTser(1)+Tsc(1)} < \frac{Tser(1)+Tsc(1)}{nTser(1)} \tag{3.7}$$

the parallel efficiency represents the effectiveness of the parallel program running on N processors relative to a single processor. However, it is the total efficiency that is of real significance when comparing the performance of a parallel program to the corresponding serial version. Let $T_s^{No}(1)$ denotes the CPU time of the corresponding serial program to reach a prescribed accuracy with *No* iterations, and $T_{B=B}^{N_1L}(n)$ denotes the total CPU time of the parallel version of the program with B blocks run on N processors, to reach the same prescribed accuracy with N_i iterations, including any idle time. The superscript *L* acknowledges degradation in performance due to the load balancing problem. The total efficiency in (3.2) can be decomposed as follows:

$$E(n) = \frac{T_s^{N_o}(1)}{n.T_{B=B}^{N_1L}(n)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)}\frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)}\frac{T_{B=B}^{N_0}(1)}{T_{B=B}^{N_1}(1)} \frac{T_{B=B}^{N_1}(1)}{T_{B=B}^{N_1}(n)}\frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1L}(n)}, \tag{3.8}$$

where $T_{B=B}^{N_1}(n)$ has the same meaning as $T_{B=B}^{N_1L}(n)$ except the idle time is not included. Comparing (3.5) and (3.2), we obtain:

$$E_{load}(n) = \frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1L}}, E_{par}(n) = \frac{T_{B=B}^{N_1}(1)}{n.T_{B=B}^{N_1}(n)},$$
$$Enum(n) = \frac{T_s^{N_o}(1)}{T_{B=B}^{N_1}(1)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)}\frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \frac{T_{B=B}^{N_o}(1)}{T_{B=B}^{N_1}(1)}, \tag{3.9}$$

when *B=1* and *n = 1, $T_m(1) + T_{sd}(1) << T_{sc}(1)$,* then $T_{B=1}^{N_o}(1)/T_s^{N_o}(1) \approx 1.0$

we note that $T_{B=B}^{N_o}(1)/T_{B=B}^{N_1}(1) = N_o / N_1$. Therefore,

$$E_{num}(n) = E_{dd}\frac{N_o}{N_1}, \quad E_{dd} = \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \tag{3.10}$$

we call (3.10) domain decomposition efficiency (DD), which includes the increase of CPU time induced by grid overlap at interfaces and the CPU time variation generated by DD techniques. The second term N_o / N_1 in the right hand side of (3.10) represents the increase in the number of iterations required by the parallel method to achieve a specified accuracy compared to the serial method.

3.5 Load Balancing

With static load balancing, the computation time of parallel subtasks should be relatively uniform across processors; otherwise, some processors will be idle waiting for others to finish their subtasks. Therefore, the domain decomposition should be reasonably uniform. A better load balancing is achieved with the pool of tasks strategy, which is often used in master – slave programming [7]: the master task keeps track of idle slaves in the distributed pool and sends out the next task to the first available idle slave. With this strategy, the processors are kept busy until there is no further task in the pool. If the tasks vary in complexity, the most complex tasks are sent out to the most powerful processor first. With this strategy, the number of sub-domains should be relatively large compared to the number of processors. Otherwise, the slave solving the last sent block will force others to wait for the completion of this task; this is especially true if this processor happens to be the least powerful in the distributed system. The block size should not be too small either, since the overlap of nodes at the interfaces of the sub-domains become significant. This results in a doubling of the computations of some variables on the interfacial nodes, leading to a reduced efficiency. Increasing the block number also lengthens the execution time of the master program, which leads to a reduced efficiency.

4. Results and Discussion

4.1 Benchmark Problem

In order to test the validation and the performance of the distributed code for the 3-D Telegraph problem at various grid sizes was computed. The solution domain was divided into rectangular blocks. The numerical results are indistinguishable to those obtained from a serial finite difference code. Consider the Telegraph Equation of the form:

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial v}{\partial z^2} = \frac{\partial^2 v}{\partial t^2} + \frac{\partial v}{\partial t} + v \tag{4.1}$$

The boundary condition and initial condition posed are:

$$\left.\begin{aligned} v(0,y,z)&=0\\ v(1,y,z)&=100\\ v(x,0,z)&=0\\ v(x,1,z)&=100\\ v(x,y,0)&=0\\ v(x,y,1)&=100 \end{aligned}\right\} \quad t\geq 0 \tag{4.2}$$

$$v(x,y,z)=e^{xyz}, \tag{4.3}$$

4.2 Parallel Efficiency

To obtain a high efficiency, the slave computational time $T_{sc}(1)$ should be significantly larger than the serial time T_{ser}. In this present program, the CPU time for the master task and the data communication is constant for a given grid size and sub-domain. Therefore the task in the inner loop should be made as large as possible to maximize the efficiency. The speed-up and efficiency obtained for various sizes, from 70x70x6 to 210x210x6, are for various numbers of sub-domains, from *B* = 50 to 200 are listed in Tables 2 – 10. In this tables we also listed the wall (elapsed) time for the master task, T_W, (this is necessarily greater than the maximum wall time returned by the slaves), the master CPU time, T_M, the average slave computational time, T_{SC}, and the average slave data communication time, T_{SD}, all in seconds. The speed-up and efficiency versus the number of processors are also shown in fig. 4 and Fig. 5, respectively, with block number *B* as a parameter.The results above show that the parallel efficiency increases with increasing grid size for given block number, and decreases with the increasing block number for given grid size. Given other parameters the speed-up increases with the number of processors. At a large number of processors, Amdahl's law starts to operate, imposing a limiting speed-up due to the constant serial time. Note that the elapsed time is a strong function of the background activities of the cluster. When the number of processors is small, the wall time decreases with the number of processors. As the number of processors become large, however, the wall time increases with the number of processors. The total CPU time is composed of three parts: the CPU time for the master task, the average slave CPU time for data communication and the average slave CPU time for computation, $T=T_M+T_{SD}+T_{SC}$. Data communication at the end of every iteration is necessary in this strategy. Indeed, the updated values of the solution variables on full domain are multicast to all slaves after each iteration since a slave can be assigned a different sub-domain under the pool-of-task paradigm. The master task includes sending updated data to slaves, assigning the task tid to slaves, waiting for message from processors, and receiving the result from slaves. For given grid size, the CPU time for send task tid to slaves increase with block number, but the timing for other tasks does not change significantly with block number. In Tables 2 – 10 we see that the master time T_M, is constant when the number of processors increases, for a given grid size and number of sub-domains. The master program is responsible for (1) sending updated variables to slaves (T_1), (2) assigning task to slaves (T_2), (3) waiting for the slaves to execute tasks (T_3), (4) receiving the results (T_4).

Table 2: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time *T*, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 70x70x6, with *B* = 50 blocks and *Niter* = 100.

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	1334	46	4	571	621	1.000	1.000
2	658	45	3	339	387	1.605	0.803
3	356	45	3	254	302	2.056	0.685
4	273	45	3	195	243	2.556	0.639
5	259	45	3	173	221	2.810	0.562
6	243	45	3	156	204	3.044	0.507
7	227	45	3	138	186	3.339	0.477
8	215	45	3	120	168	3.696	0.462
12	195	45	3	67	115	5.400	0.450
16	230	45	3	54	102	6.088	0.381

Table 3: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 120x120x6, with B = 50 blocks and *Niter* = 100.

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	2934	143	16	2016	2175	1.000	1.000
2	1382	140	14	1141	1295	1.680	0.840
3	748	140	14	775	929	2.341	0.780
4	573	140	14	614	768	2.832	0.708
5	544	140	14	474	628	3.463	0.693
6	510	140	14	396	550	3.955	0.659
7	452	140	14	359	513	4.240	0.606
8	452	140	14	325	479	4.541	0.568
12	410	141	14	209	364	5.975	0.498
16	483	140	14	178	332	6.551	0.409

Table 4: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 210x210x6, with B = 50 blocks and *Niter* = 100

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	14963	400	62	9326	9788	1.000	1.000
2	7048	392	64	5072	5528	1.770	0.885
3	3815	392	62	3627	4081	2.400	0.800
4	2922	391	62	2903	3356	2.917	0.729
5	2774	392	62	2272	2726	3.591	0.718
6	2601	392	62	1921	2375	4.121	0.687
7	2305	392	62	1747	2201	4.447	0.635
8	2305	395	64	1597	2056	4.760	0.588
12	2091	394	62	1082	1538	6.364	0.530
16	24633	394	62	938	1394	7.022	0.439

Table 5: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 70x70x6, with B = 100 blocks and *Niter* = 100.

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	1142	82	4	662	748	1.000	1.000
2	679	82	3	344	429	1.744	0.872
3	396	84	3	270	357	2.095	0.698
4	321	84	3	192	279	2.681	0.670
5	295	82	3	163	248	3.016	0.603
6	274	82	3	146	231	3.238	0.540
7	257	82	3	127	212	3.528	0.504
8	238	82	3	110	195	3.836	0.480
12	211	82	3	44	129	5.798	0.483
16	264	81	3	33	117	6.393	0.400

Table 6: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 120x120x6, with B = 100 blocks and *Niter* = 100.

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	3672	213	16	2116	2345	1.000	1.000
2	1847	213	14	1178	1405	1.669	0.885
3	1247	212	14	889	1115	2.103	0.701
4	989	212	14	682	908	2.583	0.646
5	808	212	14	528	754	3.110	0.622
6	754	212	14	437	663	3.537	0.590
7	628	212	14	383	609	3.851	0.550
8	607	212	14	335	561	4.180	0.523
12	584	214	14	198	426	5.505	0.459
16	692	212	14	137	363	6.460	0.404

Table 7. The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 210x210x6, with B = 100 blocks and $Niter$ = 100.

N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	11095	431	62	8760	9253	1.000	1.000
2	7342	429	64	4864	5354	1.728	0.864
3	3218	429	62	3398	3889	2.379	0.793
4	2198	429	62	2763	3254	2.843	0.711
5	2131	429	62	2123	2614	3.539	0.708
6	1985	429	62	1763	2254	4.105	0.684
7	1764	429	62	1614	2105	4.395	0.628
8	1711	430	64	1596	2090	4.427	0.553
12	1536	429	62	957	1448	6.389	0.532
16	25897	430	64	805	1299	7.123	0.445

Table 8. The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 70x70x6, with B = 200 blocks and $Niter$ = 100.

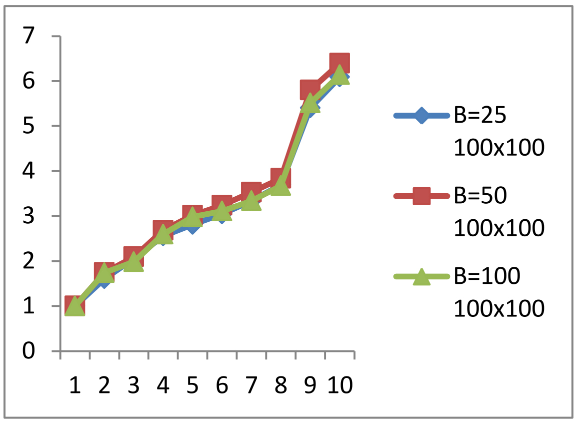
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	1538	157	4	864	1025	1.000	1.000
2	763	157	3	429	589	1.740	0.870
3	596	158	3	356	517	1.983	0.661
4	485	154	3	238	395	2.595	0.649
5	411	154	3	187	344	2.980	0.596
6	384	154	3	172	329	3.116	0.519
7	316	154	3	150	307	3.339	0.477
8	432	152	3	124	279	3.674	0.459
12	783	158	3	25	186	5.511	0.459
16	834	157	3	7	167	6.138	0.384

Table 9. The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 120x120x6, with B = 200 blocks and $Niter$ = 100.

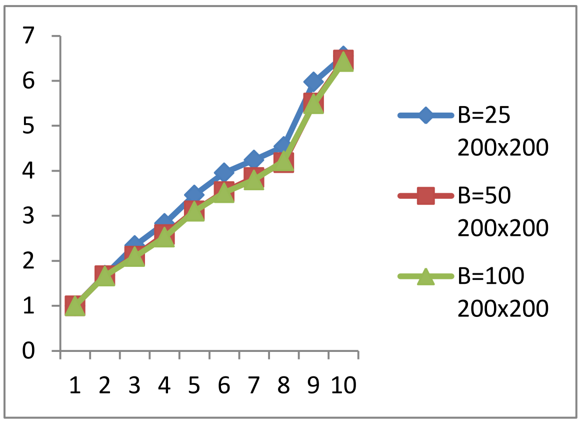
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	4752	278	16	2193	2487	1.000	1.000
2	1832	278	14	1204	1496	1.662	0.831
3	1238	278	14	895	1187	2.095	0.698
4	895	276	14	694	984	2.527	0.632
5	793	276	14	512	802	3.101	0.620
6	718	276	14	418	708	3.513	0.586
7	621	276	14	365	655	3.797	0.542
8	642	276	14	299	589	4.222	0.528
12	573	278	14	161	453	5.490	0.458
16	1158	278	14	95	387	6.426	0.402

Table 10: The wall time T_W, the master time T_M, the slave data time T_{SD}, the slave computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 210x210x6, with B = 200 blocks and $Niter$ = 100.

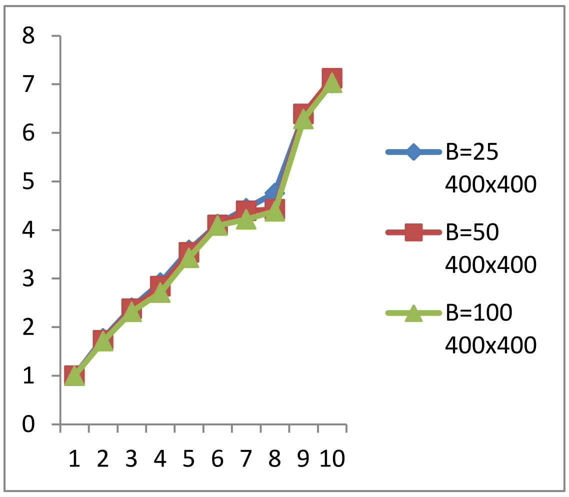
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}
1	10543	524	62	8597	9183	1.000	1.000
2	5383	518	64	4779	5361	1.713	0.857
3	3042	518	62	3385	3965	2.316	0.772
4	1985	518	62	2807	3387	2.711	0.678
5	1874	518	62	2099	2679	3.428	0.686
6	1743	518	62	1664	2244	4.092	0.682
7	1684	518	62	1596	2176	4.220	0.603
8	1619	516	64	1514	2094	4.385	0.548
12	1495	518	62	881	1461	6.285	0.524
16	27432	516	64	726	1306	7.031	0.439



a.

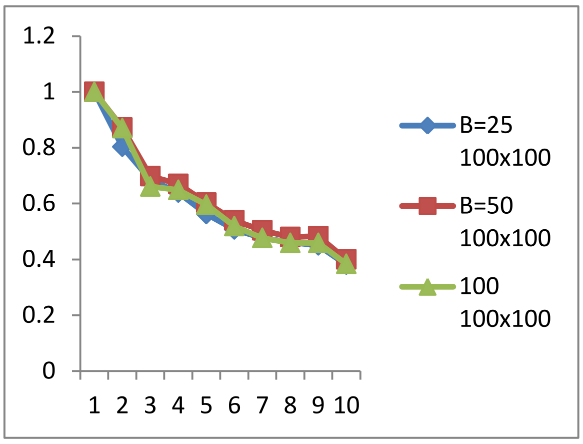


b.

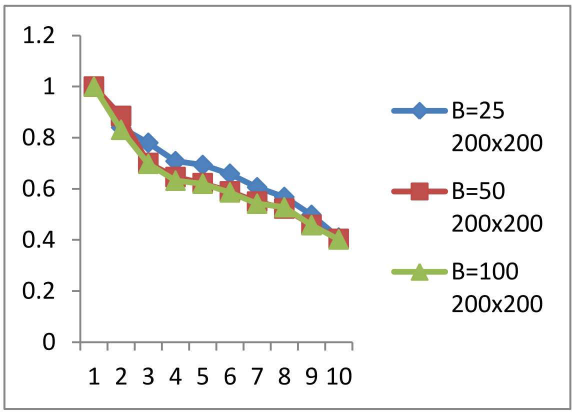


c.

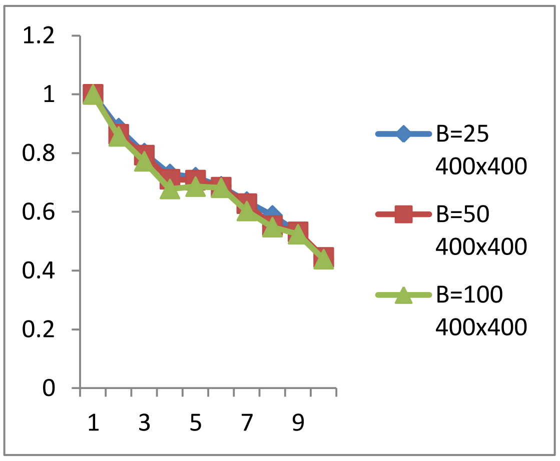
Fig. 1. Speed-up versus the number of processors. a mesh 70x70x6, b mesh 120x120x6, c mesh 210x210x6



a.



b.



c.

Fig. 2. Parallel efficiency versus the number of processors. a mesh 70x70x6, b mesh 120x120x6, c mesh 210x210x6

4.3 Numerical Efficiency

The numerical efficiency E_{num} includes the Domain Decomposition efficiency E_{DD} and convergence rate behavior N_o / N_1, as defined in Eq. (4.10). The DD efficiency $E_{dd} = T_{B=1}^{N_o}(1) / T_{B=B}^{N_o}(1)$ includes the increase of floating point operations induced by grid overlap at interfaces and the CPU time variation generated by DD techniques. In Table 12, we listed the total CPU time distribution over various grid sizes and block numbers running with only one processor. Using this table, the DD efficiency E_{DD} can be calculated, and the result are shown in Fig. 3. Note that the DD efficiency can be greater than one, even with one processor. Fig. 3 also shows that the optimum number of sub-domains, which maximizes the DD efficiency E_{DD}, increases with the grid size. The convergence rate behavior N_o / N_1, the ratio of the iteration number for the best sequential CPU time on one processor and the iteration number for the parallel CPU time on *n* processor, describes the increase in the number of iterations required by the to the serial method. This increase is caused mainly by the deterioration in the rate of convergence with increasing number of processors and sub-domains. Because the best serial algorithm is not known generally, we take the existing parallel program running on one processor to replace it. Now the problem is that how the decomposition strategy affects the convergence rate? The results are summarized in Table 13 and Fig. 4, and Table 14 and Fig. 5. It can be seen that N_o / N_1 decreases with increasing block number and increasing number of processors for given grid size. The larger the grid size, the higher the convergence rate. For a given block number, a higher convergence rate is obtained with less processors. This is because one processor may be responsible for a few sub-domains at each iteration. If some of this sub-domains share some common interfaces, the subsequent blocks to be computed will use the new updated boundary values, and therefore, an improved convergence rate results. The convergence rate is reduced when the block number is large. The reason for this is evident: the boundary conditions propagate to the interior domain in the serial computation after one iteration. But this is delayed in the parallel computation. In addition, the values of variables at the interfaces used in the current iteration are the previous values obtained in the last iteration. Therefore, the parallel algorithm is less "implicit" than the serial one. Despite these inherent short comes. A high efficiency is obtained for large scale problems.

Table 11: The slave computational time T_{SC}, for 100 iterations as a function of various block numbers

NI x NJ	*B = 1*	*B = 8*	*B = 16*	*B = 24*	*B = 50*	*B = 100*	*B = 200*
70x70x6	*625*	*409*	*395*	*486*	*571*	*662*	*864*
120x120x6	*3042*	*2456*	*2103*	*2064*	*2016*	*2116*	*2193*
210x210x6	*21658*	*14694*	*11423*	*11372*	*9326*	*8760*	*8597*

Table 12: The total computational time *T* for 100 iterations as a function of various block numbers

NI x NJ	*B = 1*	*B = 8*	*B = 16*	*B = 24*	*B = 50*	*B = 100*	*B = 200*
100x100	*732*	*565*	*563*	*572*	*621*	*748*	*1025*
200x200	*3107*	*2654*	*2361*	*2289*	*2175*	*2345*	*2487*
400x400	*28592*	*15587*	*12518*	*12493*	*9788*	*9253*	*9183*

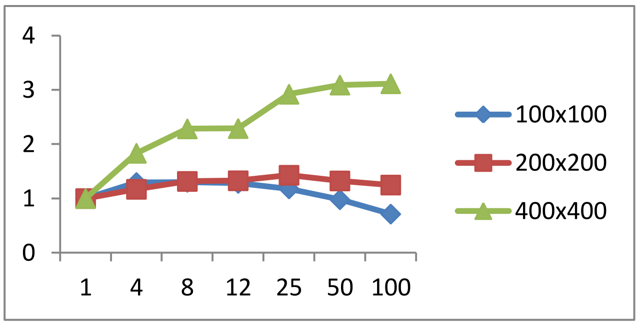


Fig. 3: The DD efficiency versus the number of sub-domains for various meshes.

Table 13: The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 70x70x6

N	*B = 1*	*B = 16*	*B = 50*	*B =100*
1	*2113*	*2269*	*2396*	*2422*
2	*2113*	*2327*	*2518*	*2564*
4	*2113*	*2384*	*2604*	*2682*
8	*2113*	*2412*	*2658*	*2704*
12	*2113*	*2412*	*2661*	*2712*
16	*2113*	*2418*	*2675*	*2728*

Table 14: The number of iteration to achieve a given tolerance of 10^{-2} for a grid of 120x120x6

N	*B = 1*	*B = 16*	*B = 50*	*B =100*
1	*2398*	*2443*	*2597*	*2602*
2	*2398*	*2451*	*2628*	*2724*
4	*2398*	*2473*	*2681*	*2765*
8	*2398*	*2473*	*2708*	*2986*
12	*2398*	*2473*	*2701*	*3032*
16	*2398*	*2473*	*2715*	*3161*

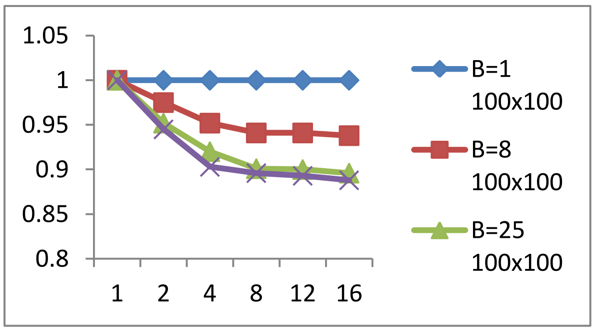


Fig.4: Convergence behavior with domain decomposition for mesh 70x70x6

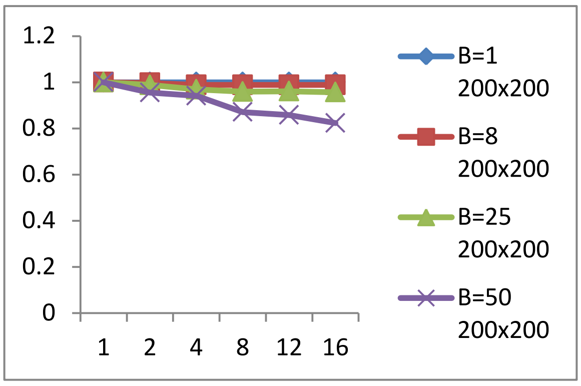


Fig. 5: Convergence behavior with domain decomposition for mesh 120x120x6

4.4 Total Efficiency

We implemented the serial computations on one of the processors, and calculated the total efficiencies. The total efficiency *E(n)* for grid sizes 70x70x6 and 120x120x6 have been showed respectively. From Eq. (4.8), we know that the total efficiency depend on N_o / N_1, E_{par} and DD efficiency E_{DD} since the load balancing is not the real problem here. For a given grid size and block number, the DD efficiency is constant. Thus, the variation of *E*(*n*) with processor number *n* is governed by E_{par} and N_o / N_1. When the processor number becomes large, *E*(*n*) decreases with *n* due to the effect of both the convergence rate and the parallel efficiency.

5. CONCLUSIONS

This paper presented a study on the parallelization of 3-D ADI scheme on 3-D Telegraph problem with PVM, on a distributed memory system using SPMD model on a master-slave platform. The system allows a parallel collection of overlapping communication to avoid unnecessary synchronization and to have the impact of parallel convergence. In addition to the use of ease of our platform, compared to other approaches show negligible overhead with effective load scheduling over various mesh sizes, which produce the expected inherent speedups. It was also confirmed that flexible scheduling for the overlapping communication are important, and this is easy on the master-slave platform with SPMD model as seen from the Tables and Figures. The convergence rate depends upon the block numbers and the number of processors for a given grid. For a given number of blocks, the convergence rate increases with decreasing number of processors, and for a given number of processors, it decreases with increasing block number. Computational results obtained have clearly shown the benefits of parallelization. The DD greatly influences the performance of the 3-D ADI scheme on the parallel computers. On the basis of the current parallelization strategy, more sophisticated models can be attacked efficiently.

6. REFERENCES

[1] Aloy R., Casaban M.C., Caudillomate L.A., Jodar L., 2007. Computing the Variable Coefficient Telegraph Equation using a Discrete Eigen Functions Method. *Computers and Mathematics with Applications* 54, pp. 448 – 458.

[2] W. Barry, A. Michael, 2003. Parallel Programming Techniques and Application using Networked Workstation and Parallel Computers. Prentice Hall, New Jersy

[3] A. Beverly, et al., 2005. The Algorithmic Structure Design Space in Parallel Programming. Wesley Professional

[4] R. Chypher, A. Ho, et al., 1993. Architectural Requirements of Parallel Scientific Applications with Explicit Communications. *Computer Architecture*, pp 2 – 13

[5] P.J Coelho, M.G Carvalho, 1993. Application of a Domain Decomposition Technique to the Mathematical Modeling of Utility Boiler. *Journal of Numerical Methods in Eng.*, 36 pp 3401 – 3419

[6] F. Durst, M. Perie, D. Chafer, E. Schreck, 1993. Parallelization of Efficient Numerical Methods for Flows in Complex Geometries. *Flow Simulation with High Performance Computing I*, pp 79 – 92, Vieweg, Braunschelweig

[7] J. H. Eduardo, M. A., H. Amaral (2007). Speedup and Scalability Analysis of Master-Slave Applications on Large Heterogeneous Clusters. *Journal of Parallel and Distributed Computing* 67(11), pp 1155 - 1167

[8] D.J Evans, B. Hassan, 2003. Numerical Solution of the Telegraph Equation by the AGE Method. *Int'l Journal of Computer Mathematics* Vol. 80, number 10, pp 1289 – 1297

[9] D.J. Evans, M.S. Sahimi, The Alternating Group Explicit Iterative Method for Parabolic Equations I: 2-Dimensional Problems, *Int'l. J. Compt. Math,* Vol. 24, (1988) pp. 311-341

[10] S. U. Ewedafe, H. S. Rio, 2011. Parallel Implementation of 2-D Telegraph Equation on MPI/PVM Cluster. *Int'l Jour. of Parallel Programming*, 39, Issue 2, 202 – 231

[11] S. U. Ewedafe, H. S. Rio, 2011. Armadillo Generation Distributed Systems & Geranium Cadcam Cluster for solving 2-D Telegraph Equation. *Int'l Jour. of Computer Mathematics*, 88, Issue 3, 589 – 609

[12] Fan C., Jiannong C., Yudong S. 2003. High Abstractions for Message Passing Parallel Programming. *Parallel Computing* 29, 1589 – 1621.

[13] A. Geist A. Beguelin, J. Dongarra, 1994. Parallel Virtual Machine (PVM). Cambridge, MIT Press

[14] G.A Geist, V.M Sunderami, 1992. Network Based Concurrent Computing on the PVM System. *Concurrency Practice and Experience*, pp 293 – 311

[15] Guang-Wei Y., Long-Jun S., Yu-Lin Z. 2001. Unconditional Stability of Parallel Alternating Difference Schemes for Semilinear parabolic Systems. *Applied Mathematics and Computation* 117, pp 267 – 283

[16] K. Jaris, D.G. Alan, 2003. A High-Performance Communication Service for Parallel Computing on Distributed Systems. *Parallel Computing* 29, pp 851 – 878

[17] Mitchell, A.R., Fairweather, G. (1964). Improved forms of the Alternating direction methods of Douglas, Peaceman and Rachford for solving parabolic and elliptic equations, *Numer. Maths*, 6, 285 – 292.

[18] D.W Peaceman, H.H Rachford, 1955. The Numerical Solution of Parabolic and Elliptic Differential Equations. *Journal of Soc. Indust. Applied Math*. 8 (1) pp 28 – 41

[19] Peizong L., Z. Kedem, 2002. Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers. *ACM Transactions on Programming Languages and Systems*, vol. 24, number 1, pp 1 – 50

[20] M.J Quinn, 2001. Parallel Programming in C. MC-Graw Hill Higher education New York.

[21] R. Rajamony, A. L. Cox, 1997. Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependence Analysis. *Performance Review* 25 (1), pp 75 – 87

[22] B. V Rathish Kumar, et al., 2001. A Parallel MIMD Cell Partitioned ADI Solver for Parabolic Partial Differential Equations on VPP 700. *Parallel Computing* 42, pp 324 – 340

[23] V. T Sahni, 1996. Performance Metrics: Keeping the Focus in Routine. *IEEE Parallel and Distributed Technology*, Spring pp 43 – 56.

[24] X.H Sun, J. Gustafson, 1991. Toward a Better Parallel Performance Metric. *Parallel Computing* 17.

[25] M. Tian, D. Yang, 2007. Parallel Finite-Difference Schemes for Heat Equation based upon Overlapping Domain Decomposition. *Applied Maths and Computation*, 186, pp 1276 – 1292