International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org, editorijettcs@gmail.com
Volume 8, Issue 6, November - December 2019 ISSN 2278-6856

Input File Affinity Measure Running on Master-Worker Paradigm with MPI/PVM

Ewedafe Simon Uzezi

Department of Computing, Faculty of Science and Technology,
The University of the West Indies, Mona Kingston 7, Jamaica

Abstract: ***In this paper we implement a parallel performance of a type of alternating iterative scheme called Mitchell-Fairweather Double Sweep Method (MF-DS) on 2-D Telegraph Equation (TE) using an input file affinity measure running on a master-worker paradigm. The implementation was carried out using Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). The TE was discretized using the popular finite difference method resulting into MF-DS. Parallel performance and parallel effectiveness of the method using the Input File Affinity Measure (I_{aff}) was experimentally evaluated and numerical results show its effectiveness and parallel performance. We show in this paper that, the input file affinity measure has scalability performance with less information to schedule tasks.***

Keywords: Parallel Performance, Input File Affinity, Master-Worker, 2-D Telegraph Equation, MPI.

1. INTRODUCTION

Distributed application has become increasingly popular, not only because of their commodity component architecture that gives them an economic advantage, but also because of their potential to achieve high performance by exploiting application-level parallelism. However, this potential is often compromised by the high overheads of communication among cluster nodes [32]. Computing infrastructures are reaching an unprecedented degree of complexity. First, parallel processing is coming to mainstream, because of the frequency and power consumption wall that leads to the design of multi-core processors. Second, there is a wide adoption of distributed processing technologies because of the deployment of the Internet and consequently large-scale grid and cloud infrastructures. All these combined together make programming of computing infrastructure a very difficult challenge. Programmers have to face with both parallel and distributed programming paradigms when designing an application, and several software codes that are executed on various computing resources spread over the Internet within a grid or cloud-base infrastructure [29].

Parallel applications are composed of sequential, independent tasks. By independent we mean that there is no communication or dependencies among tasks. The input for each task is one or more files and one file can be input for more than one task. The output for each task is also one or more files, the output files and each task generates its own set of output files. It is worth noting that these applications are often referred to as parallel-sweep applications [21]. In master-worker paradigm a master node is responsible for scheduling computation among the worker and collecting the results [7]. The master-worker paradigm has fundamental limitations: both communications from the master and contention in accessing file repositories may become bottlenecks to the overall scheduling scheme, causing scalability problems. It is worth noting that related works on parallel implementation focus on the problem of efficiently scheduling application to execute on distributed architectures organized either as pure master-worker or hierarchical platforms, while lacking scalability analysis. There are several metrics available for measuring the scalability of algorithm-machine pairs.

Parallelization of Partial Differential Equations (PDEs) by time decomposition was first proposed by [34] following earlier efforts at space-time methods [27, 28, 49]. The application of the alternating iterative methods to solve problem of 2-D telegraph equations have shown that they need high computational power and communications. Numerical algorithms comprise of series of heuristics that can help to optimize a wide range of tasks required for parallel and distributed architectures to work efficiently. Genetic Algorithms (GAs), Genetic Programming (GP) or Simulated Annealing (SA) are nowadays helping computer designers on the advance of computer architecture, while improving on parallel architectures are allowing to run computing intensive numerical algorithms for solving other difficult problems. In the world of parallel computing MPI is the de facto standard for implementing programs on multiprocessors. To help with program development under a distributed computing environment a number of software tools have been developed MPI and PVM [23, 25] are chosen here since they have large user groups. Many applications are “embarrassingly” parallel and require minimal performance out of MPI. These applications exploit coarse grain parallelism and communicate rarely. Nevertheless, measuring the communication performance is useful for determining which applications are most suitable for MPI. The ultimate goal of running MPI on parallel computers is to increase programmer productivity and decrease the large software cost of using High Performance Computing (HPC) systems. Obtaining increased peak performance (i.e. exploiting more parallelism) requires more lines of code. MPI is a point-point message library; a significant amount of code can be added to any application to do parallel operations.

Sequential numerical methods for solving time dependable problems have been explored extensively [40, 47]. 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 (LAN) [10]. Relevant data need to be passed from processor to processor through a message passing mechanism [13, 22, 14, 42, 30]. The telegraph equation is important for modeling several relevant problems such as signal analysis [1], wave propagation [38], e.t.c. In this paper, we deal with an electrical transmission line with constant linear parameters resistance (R), inductance (L), capacitance (C) and leakage conductance (G) in both the space and time domains. A number of iterative methods are developed to solve the telegraph equations [18]. Some of these iterative schemes are employed in various parallel platforms [49, 27, 28]. Parallel algorithms have been implemented for the finite difference method [19, 20, 17], the discrete eigen functions method used in [2] and [17] used the AGE method on 1-D telegraph problem, but not implemented on parallel platform for parallel improvement. The boundary element method and the finite volume method using domain decomposition have been implemented [13].

We present in this paper algorithms which have scalability performance comparable to other algorithms in several circumstances. We describe the design of our parallel system through I_{aff} for understanding parallel execution. Through several implementations and performance improvement analysis of the alternating iterative methods, we explore how to use the I_{aff} on master-slave paradigm for identifying parallel performance on 2-D TE. We assume the amount of work is increased exclusively by increasing the number of tasks. The reason for this is that it is relatively easy to increase the range of parameters to be analyzed in an application. On the order hand, in order to increase the amount of work related to each individual task, the input data for that task should be changed, which may change the nature of the problem to be processed by those tasks and it is not always feasible. It is worth noting that some of the related works mentioned focus on the problem of efficient scheduling of the decomposition to execute on distributed architectures organized either as pure master-slave or hierarchical platforms, while lacking scalability analysis. We consider that the amount of computation associated with each task is fixed, each task depends on one or more input files for execution, and input file can be shared among tasks [26]. The implementation was carried out using MPI and PVM; we solve the 2-D TE using MF-DS [47]. The method involves the solution of sets of tridiagonal equations along lines parallel to the x and y axes at the first and second time steps respectively. The first is by employing the double sweep method of [37], while the second is [20, 46].

In this work we assume that both the size of the input files, the dependencies among input files and tasks are known. We also consider that the master node has access to a storage system which serves as the repository of all input and output files. This paper is organized as follows: Section 2 discusses previous research work. Section 3 presents the model for the 2-D TE and introduces the MF-DS scheme. Section 4 describes the input file affinity measure. Section 5 exemplifies the use of the input file affinity measure. Section 6 describes the parallel implementation of the algorithms. Section 7 presents the numerical and experimental result of the schemes under consideration. Finally, section 8 presents our conclusions.

1.1 Previous research work

Parallel performance of algorithms has been studied in several papers during the last years [6, 25, 11, 12, 15, 25]. Bag of Task (BoT) applications composed of independent tasks with file sharing have appeared in papers [7, 26, 31]. Their relevance has motivated the development of specialized environments which aim to facilitate the execution of large BoT applications on computational grids and clusters, such as the AppLes Parameter-Sweep Template (APST) [5]. Giersch et al., [26] proved theoretical limits for the computational complexity associated to the scheduling problem. The authors also proposed several new heuristics which produce schedules that approach the quality achieved by the heuristics proposed by [7], while keeping the computational complexity one order of magnitude faster. In [18], they proposed an iterative scheduling approach that produces effective and efficient schedules, compared to the previous works in [7, 26]. However, for homogeneous platforms, the algorithms proposed in [31] can be considerably simplified, in a way that it becomes equivalent to algorithms proposed previously. Fabricio [21] analyzes the scalability of BoT applications running on master-slave platforms and proposed a scalability related measure.

Eric [3] presents a detailed study on the scheduling of tasks in the parareal algorithm that achieves significantly better efficiency than the usual algorithm. It proposes two algorithms, one which uses a manager-worker paradigm with overlap of sequential and parallel phases, and a second that is completely distributed. Hinde et al. [29], proposed a generic approach to embed the master-worker paradigm into software component models and describes how this generic approach can be implemented within an existing software component model. Many works deal with the master-worker paradigm. With respect to distributed computing, they can be divided in two categories according to [29]. One side, some works focus on Network Enabled Servers (NES), usually based on the GridRPC operations, standardized by the OGF [39]. On the other side, the master-worker paradigm is very popular on desktop grids such as SETI@HOME [23], and BOINC [4]. They all rely on a more or less automatic management of non-functional properties such as worker management, the request scheduling and the transport of requests between master and workers. The desired level of transparency is provided through an application programming interface (API) that implements the user visible part of the system. For our concern, there are two main API, one for the master side and one for the worker side. Key message approach to prioritize communications along critical path to speed up

execution of parallel applications in a cluster environment was presented by [50].

On the numerical computing part, [44] went in search of numerical consistency in parallel programming and presented methods that can drastically improve numerical consistency for parallel calculations across varying numbers of processors. The study assesses the value of the enhanced numerical consistency in the context of general finite difference calculations. In this research paper, we consider the consistency approach of the alternating iterative methods from the descritization of PDEs using I_{aff} implemented on master-worker paradigm across varying numbers of processors with MPI and PVM, and proposed some performance improvement methods. In [19, 20], parallel implementation of 2-D telegraph equation using the SPMD technique and domain decomposition strategy have been researched and they presented a model that enhances overlap communication with computation to avoid unnecessary synchronization, thus yield significant speed up.

ADI method for the PDEs proposed by Peaceman and Rachford [41] 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, [43], has proposed a parallel ADI solver for linear array of processors. Chan and Saied [8] have implemented ADI scheme on hypercube. Later Lixing et al. [35], parallelized the ADI solver on multiprocessors. The ADI method in [41] has been used for solving heat equations in 2D. Several approaches to solve the telegraphic equations using different numerical schemes have been treated in [17, 18].

2. TELEGRAPH EQUATION

This paper deals with an electrical transmission line with constant linear parameters R, L, C and G, in both the frequency and time domains. The speed of convergence of iterative scheme is examined for the synchronous communication approaches in parallel environment [48, 49]. In this present work we are dealing with the numerical approximation of the second order telegraph equation as shown in eq. (3.1), where *a* and *b* are known as constant coefficients. Equation (3.1) below referred to as second-order telegraph equation with constant coefficients, models mixture between diffusion and wave propagation by introducing a term that accounts for effects of finite velocity to standard heat or mass transport equation [16].

However, the TE (2.1) is commonly used in signal analysis for transmission and propagation of electrical signals [36] and also has applications in other fields (see [45]). In recent years, much attention has been given in the literature to the development, analysis and implementation of stable methods for the numerical solution of 2-D telegraph equation [38]. Recently, [38] developed unconditional stable difference scheme for the solution of multi-dimensional telegraph equations. The schemes are second-order accurate in space and time. Of concern are suspension flows. These combine directed and random motion and are traditionally modeled by parabolic partial differential equations. Sometimes they can be better modeled (in terms of fitting the data generated by certain blood flow experiments) by the TE. The existence of time-bounded solutions of nonlinear bounded perturbations of the telegraph equation with Neumann boundary conditions has recently been considered in [1]. We consider the second order TE as given in [17]:

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

$$0 \le x \le 1,\ 0 \le y \le 1,\ t > 0$$

with initial condition

$$v(x, y, 0) = f(x, y) \qquad (2.2a)$$

and boundary conditions

$$\left.\begin{aligned} v(0, y, t) &= f_1(y, t),\ \ v(1, y, t) = f_2(y, t) \\ v(x, 0, t) &= f_3(x, t),\ \ v(x, 1, t) = f_4(x, t) \end{aligned}\right\} \qquad (2.2b)$$

It is assumed that the initial and the boundary conditions are given with sufficient smoothness to maintain the order of accuracy and consistency of the different schemes under consideration. To obtain the different solutions of the above initial and boundary conditions, we divide the interval $[0,1]$ into n subintervals. Where $a = RC + GL$, let $\Delta x, \Delta y$ and Δt be the grid spacing in the x, y and t directions, where $\Delta x = 1/m,\ \Delta y = 1/n,$ *m* and *n* are the positive integers. The approximation values $v_{i,j,k}$ of the solution $v(x, y, t)$ for the problem (2.1) – (2.2b) are to be computed at the grid points (x_i, y_j, t_k), where $x_i = i\Delta x,\ i = 0,1,2,...,m,\ y_j = j\Delta j,\ j = 0,1,...n;$ $b = 1/LC$. The region and its boundary consisting of the lines $x = 0,\ x = 1$ and $t = 0,$ have thus discretized at the grid points (x_i, y_j, t_k). Let $v_{i,j}$ be the approximate values of $v(x,t)$ at the grid point (x_i, t_k). The analytical solution of the initial boundary value problem eq. (3.1 – 3.2b) cannot be determined for arbitrary $f(x,t)$ and the only alternative is the application of stable numerical methods. Out of the number of numerical methods available, the most important is finite difference method, because it is easy to implement and universally applicable. Evans and Hassan [17] have discussed alternating group explicit finite difference method for the solution of TE. Mohanty [38] has discussed finite difference schemes of $O(k^4 + h^4)$ for the solution of multi-dimensional linear telegraph equation and it has been shown that the schemes are conditionally stable. Furthermore, the standard central difference scheme of $O(k^2 + h^2)$ for the differential equation (3.1), is obtained by using the second-order central difference approximations to the derivatives. Let

$t_k = k\Delta t,\ k = 1,2,\ldots$ For simplicity, we take $\Delta x = \Delta y = \Delta d$, and sometimes denote (x_i, y_j, t_k) by $(\mathrm{i}, \mathrm{j}, \mathrm{k})$. Among the finite difference method for the numerical solution of the problem (2.1) - (2.2b), the classical explicit method is suitable in any case for parallel computing, but the method is stable only when $\Delta t/\Delta d^2 \le 1/4$, thus Δt must be restricted to a very small value. The central and forward operator is given by:

$$\begin{aligned}
\frac{\partial^2 \mathrm{v}}{\partial \mathrm{t}^2} &= \frac{\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1}} - 2\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n}} + \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n-1}}}{(\Delta \mathrm{t})^2}, \\
\frac{\partial \mathrm{v}}{\partial \mathrm{t}} &= \frac{\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1}} - \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n-1}}}{2\Delta \mathrm{t}} \\
\frac{\partial^2 \mathrm{v}}{\partial \mathrm{x}^2} &= \frac{\mathrm{v}_{\mathrm{i+1,j}}^{\mathrm{n+1}} - 2\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1}} + \mathrm{v}_{\mathrm{i-1,j}}^{\mathrm{n+1}}}{(\Delta \mathrm{x})^2}, \\
\frac{\partial^2 \mathrm{v}}{\partial \mathrm{y}^2} &= \frac{\mathrm{v}_{\mathrm{i,j+1}}^{\mathrm{n+1}} - 2\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1}} + \mathrm{v}_{\mathrm{i,j-1}}^{\mathrm{n+1}}}{(\Delta \mathrm{y})^2}
\end{aligned} \tag{2.3}$$

extending the finite difference scheme on the telegraph equation of (2.1) becomes:

$$\frac{v_{i,j}^{n+1} - 2v_{i,j}^{n} + v_{i,j}^{n-1}}{(\Delta t)^2} + a\frac{v_{i,j}^{n+1} - v_{i,j}^{n-1}}{2\Delta t} - b\left\{\frac{v_{i+1,j}^{n+1} - 2v_{i,j}^{n+1} + v_{i-1,j}^{n+1}}{(\Delta x)^2} + \frac{v_{i,j+1}^{n+1} - 2v_{i,j}^{n+1} + v_{i,j-1}^{n+1}}{(\Delta y)^2}\right\} = 0 \tag{2.4}$$

Although this simple implicit scheme is unconditionally stable, we need to solve a penta-diagonal system of algebraic equations at each time step. Therefore, the computational time is huge.

2.1 DS-MF

We recall from [19] and [20] that the 2-D ADI method resulted into:
sub-iteration 1 is given by:

$$\begin{aligned}
&-\rho_x \mathrm{v}_{\mathrm{i+1,j}}^{\mathrm{n+1(1)}} + (1+2\rho_x)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(1)}} - \rho_x \mathrm{v}_{\mathrm{i-1,j}}^{\mathrm{n+1(1)}} = \\
&-(\mathrm{A}_2)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(*)}} + (2\mathrm{C}_o \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n}} - \mathrm{C}_1 \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n-1}}) \quad \forall\, \mathrm{i,j}
\end{aligned} \tag{2.5}$$

let $\mathrm{a} = 1 + 2\rho_x$, $\mathrm{b} = \mathrm{c} = -\rho_x$. For various values of i and j, (2.5) can be written in a more compact matrix form at the (k + ½) time level as:

$$\mathrm{Av}_{\mathrm{j}}^{(\mathrm{k+1/2})} = \mathrm{f}_{\mathrm{k}}, \quad \mathrm{j} = 1,2,\ldots \mathrm{n}. \tag{2.6}$$

where

$$\mathrm{v} = (\mathrm{v}_{1,\mathrm{j}}, \mathrm{v}_{2,\mathrm{j}}, \ldots, \mathrm{v}_{\mathrm{m,j}})^{\mathrm{T}},\ \mathrm{f} = (\mathrm{f}_{1,\mathrm{j}}, \mathrm{f}_{2,\mathrm{j}}, \ldots \mathrm{f}_{\mathrm{m,j}})^{\mathrm{T}}$$

at the (k + 1) time level, sub-iteration 2 is given by:

$$\begin{aligned}
&-\rho_y \mathrm{v}_{\mathrm{i,j+1}}^{\mathrm{n+1(2)}} + (1+2\rho_y)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(2)}} - \rho_y \mathrm{v}_{\mathrm{i,j-1}}^{\mathrm{n+1(2)}} = \\
&\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(1)}} + \mathrm{A}_2 \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(*)}}. \quad \forall\, \mathrm{i,j}
\end{aligned} \tag{2.7}$$

let $\mathrm{a} = 1 + 2\rho_y$, $\mathrm{b} = \mathrm{c} = -\rho_y$. For various values of i and j, (2.7) can be written in a more compact matrix form as:

$$\mathrm{Bv}_{\mathrm{i}}^{(\mathrm{k+1})} = \mathrm{g}_{\mathrm{k+1/2}}, \qquad \mathrm{i} = 1,2,\ldots \mathrm{m} \tag{2.8}$$

where

$$\mathrm{v}_{\mathrm{i}}^{(\mathrm{k+1})} = (\mathrm{v}_{\mathrm{i,1}}, \mathrm{v}_{\mathrm{i,2}}, \ldots, \mathrm{v}_{\mathrm{i,n}})^{\mathrm{T}},\ \mathrm{g} = (\mathrm{g}_{\mathrm{i,1}}, \mathrm{g}_{\mathrm{i,2}}, \ldots, \mathrm{g}_{\mathrm{i,n}})^{\mathrm{T}}$$

and set

$$v_{i,j}^{n+1(*)} = 2v_{i,j}^{n} - v_{i,j}^{n-1} \tag{2.9}$$

this is a prediction of $v_{i,j}^{n+1}$ by the extrapolation method. Splitting by using an ADI procedure as in [41], we get a set of recursion relations as follows:

$$\begin{aligned}
&(I + A_1)v_{i,j}^{n+1(1)} = -(A_2)v_{i,j}^{n+1(*)} + \\
&(2C_o v_{i,j}^{n} - C_1 v_{i,j}^{n-1})
\end{aligned} \tag{2.10}$$

$$(I + A_2)v_{i,j}^{n+1(2)} = v_{i,j}^{n+1(1)} + A_2 v_{i,j}^{n+1(*)} \tag{2.11}$$

where $v_{i,j}^{n+1(1)}$ is the intermediate solution and the desired solution is $v_{i,j}^{n+1} = v_{i,j}^{n+1(2)}$.

The numerical representative of Eq. (2.10) and (2.11) using the Mitchell and Fairweather scheme is as follows:

$$\begin{aligned}
&\left(1 - \frac{1}{2}\left(\rho_x - \frac{1}{6}\right)\mathrm{A}_1\right)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(1)}} = \\
&\left(1 + \frac{1}{2}\left(\rho_y + \frac{1}{6}\right)\mathrm{A}_2\right)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(*)}} \\
&+\left(2\mathrm{C}_o \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n}} - \mathrm{C}_1 \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n-1}}\right)
\end{aligned} \tag{2.12}$$

$$\begin{aligned}
&\left(1 - \frac{1}{2}\left(\rho_y - \frac{1}{6}\right)\mathrm{A}_2\right)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(2)}} = \\
&\left(1 + \frac{1}{2}\left(\rho_x + \frac{1}{6}\right)\mathrm{A}_1\right)\mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(1)}} + \mathrm{A}_2 \mathrm{v}_{\mathrm{i,j}}^{\mathrm{n+1(*)}}
\end{aligned} \tag{2.13}$$

The horizontal sweep (2.12) and the vertical sweep (2.13) formulas can be manipulated and written in a compact matrix. Let $a = \frac{5}{6} + \rho_x$, $b = c = \frac{1}{12} - \frac{\rho_x}{2}$. For various values of *i* and *j* , (2.12) can be written in a more compact matrix form at the (*k* + ½) time level as in (2.6). Similarly, let $a = \frac{5}{6} + \rho_y$, $b = c = \frac{1}{12} - \frac{\rho_y}{2}$. For various values of *i* and *j* , (2.13) can be written in a more compact matrix form at the (*k* + *1*) time level as in (2.16). By defining $a = \frac{5}{6} + \rho_y$, $b = c = \frac{1}{12} - \frac{\rho_y}{2}$ the resulting tri-diagonal system of equations are solved using similar iterative procedure as in the DS-PR, that is, the two-stage IADE-DY algorithm.

3 THE I_{AFF}

With reference to [11], we introduce the I_{aff}. First, we describe a simplified execution model that classifies several issues related with the execution of the telegraphic equation on the master-worker paradigm. Typically, each task goes through three phrases during execution of a parameter-sweep application: (1) an initialization phase, where the necessary files are sent from the master to the slave node and the task is started. The duration of this phase is equal to t_{init}. Note that this phase includes the overhead incurred by the master to initiate a data transfer to a slave, (2) a computational phase, where the task processes the parameter file at the slave node and produces an output file. The duration of this phase is equal to t_{comp}. Any additional overhead related to the reception of input files by a worker node is also included in this phase and (3) a completion phase, where the output file is sent back to the master and the master task is completed. The duration of this phase is equal to t_{end}. This phase may require some processing at the master, mainly related to writing out files to the repository. Since this writing may be deferred until the disk is available, we assume that this processing time is negligible. Therefore, the initialization phase of one slave can occur concurrently with the completion phase of another slave node. Given these three phases, the total execution time of a task is equal to

$$t_{total} = t_{init} + t_{comp} + t_{end} \quad (3.1)$$

as the machine model, we consider a cluster composed of *P + 1* processors. For the rest of this paper we assume that *T » P*. One processor is the master and the other processors are workers. Communication among master and workers is carried out through Ethernet and master can only send files through the network to a single worker at a given time. We assume the communication link is full-duplex, i.e., the master can receive an output file from a worker at the same time it sends an input file to another worker. We also assume that communication computation begins as soon as the input files are completely received. This assumption is coherent to the master-worker paradigm that is currently available in [14].

For the sake of simplicity and without loss of generality, we consider in this section that there is no contention related to the transmission of output files from worker nodes to the master. Indeed, it is possible to merge the computation phase with the completion phase without affecting the results of this section. Therefore, we merge both phases as t'_{comp} in the equations that follow. A worker is idle when it is not involved with the execution of any of the three phases of a task. For the results below, the task model is composed of *T* heterogeneous tasks. All tasks and files have the same size and each task depends upon a single non-shared file. Note that the problem of scheduling an application where each task depends upon a single non-shared file, all tasks and files have the same size and the master-worker paradigm is heterogeneous, has polynomial complexity [14]. These assumptions are considered in the analysis that follows. We define the effective number of processors P_{eff} as the maximum number of workers needed to run an application with no idle periods on any worker processor. Taking into account the task and platform models described in this section, a processor may have idle periods if:

$$t'_{comp} < (P-1)t_{init} \quad (3.2)$$

P_{eff} is then given by the following equation:

$$P_{eff} = \left\lfloor \frac{t'_{comp}}{t_{init}} + 1 \right\rfloor \quad (3.3)$$

the total number of tasks to be executed on a processor is at most

$$M = \left\lceil \frac{T}{P} \right\rceil \quad (3.4)$$

for a platform with P_{eff} processors, the upper bound for the total execution time (makespan) will be

$$\left\lceil t_{makespan} \right\rceil = M(t_{init} + t'_{comp}) + (P-1)t_{init} \quad (3.5)$$

the second term in the right hand side of Eq. (3.5) shows the time needed to start the first (P – 1) tasks in the other P – 1 processors. If we have a platform where the number of processors is larger than P_{eff} the overall makespan is dominated by communication times between the master and the workers. We then have:

$$\left\lceil t_{makespan} \right\rceil = MPt_{init} + t'_{comp} \quad (3.6)$$

the set of Eqs. (3.2) – (3.6) will be considered in subsequent sections of this paper. It is worth noting that Eq. (3.5) is valid when workers are constantly busy, either performing computation or

communication. Eq. (3.6) is applicable when workers have idle periods, i.e., are not performing either computation or communication. Eq. (3.2) occurs mainly in two cases:

(1) For very large platforms (P large).

(2) For applications with small $\frac{t_{comp}}{t_{init}}$ ratio, such as fine-grain applications.

In order to measure the degree of affinity of a set of tasks concerning their input files, we introduce the concept of input file affinity. Given a set of *G* of tasks, composed of *K* tasks, $G = \{T_1, T_2, \ldots, T_k\}$, and the set *F* of the *Y* input files needed by the tasks belonging to group *G*, $F = \{f_1, f_2, \ldots, f_y\}$, we define I_{aff} as follows:

$$I_{aff}(G) = \frac{\sum_{i=1}^{y}(N-1)|f_i|}{\sum_{i=1}^{y} N_i |f_i|} \qquad (3.7)$$

where $|f_i|$ is the size in bytes of file f_i and $N_i (1 \le N_i \le K)$ is the number of tasks in group G which have file f_i as an input file. The term $(N_i - 1)$ in the numerator of the above equation can be explained as follows: if N_i tasks share an input file f_i, that file may be sent only once (instead of N_i times) when the group of tasks is executed on a worker node. The potential reduction of the number of bytes transferred from a master node to a worker node considering only input file f_i is then $(N_i - 1)|f_i|$. Therefore, the input file affinity indicates the overall reduction of the amount of data that needs to be transferred to a worker node, when all tasks of a group are sent to one node. Note that $0 \le I_{aff} \le 1$. For the special case where all tasks share the same input file $I_{aff} = \frac{k-1}{k}$, where k is the number of tasks of a group.

3.1 Using the I_{aff}

In this section, we exemplify the utilization of the input file affinity concept to schedule the iterative methods discussed above on 2-D TE. The dynamic clustering algorithm is oblivious to task execution time, i.e., it does not need to know in advance actual or estimated task execution times. This class of algorithm is important especially when information about task execution time is not available and cannot be estimated accurately. The initial of the algorithm is to have an initial grouping based only on static information and then tasks are replicated between nodes taking into account the rate of task completion, which will depend on the load of individual processors. A description of the algorithm follows (see Fig.1):

-1. Group tasks using the input file affinity metric.

-2. When dispatching tasks to processors, the input files of the groups of tasks are sent in a pipelined way, overlapping communication with computation whenever possible. The master node only sends a file to a worker node if the file is not already stored on the worker node, in order to save network bandwidth. The task that is executed first is the one whose sum of input files bytes to be transferred is the smallest, in order to begin execution on the worker node as soon as possible. Note that this also reduces t_{init} when file transfer are pipelined.

```
1.  Group tasks according to Input File Affinity;
2.  Schedule Groups of tasks on Worker Processors;
3.  Wait;
4.  On Pi returning results after x tasks completed do
    {
5.          Compute average execution time on
processor Pi;
6.          Update task queue;
7.          Abort still running replicas of completed
tasks;
8.          If Processor Pi is idle
9.              If there are unfinished groups not yet
replicated on slower processors
10.                 Replicate unfinished group into
processor Pi;
    }
```

Fig.1 Clustering Algorithm

-4. For every *x* tasks completed (where *x* is an adjustable parameter), the worker machine should send the corresponding results back to the master node. This mechanism is similar to a regular check-pointing. If the worker machine fails, the only tasks that have to be executed are those for which the results were not received yet. It is also possible to obtain information about the current load of a machine by measuring the number of tasks still to be executed.

-8. If a machine becomes idle, send a replica of the remaining tasks (not yet replicated) of the machine with the largest amount of unfinished computation to be executed on the idle machine.

-10. If processor P_i is idle, the master identifies the processor P_s that made the smaller amount of progress in processing its tasks and replicates some of the tasks into P_i. Tasks for replication are chosen in a way that input file affinity is maximized.

The complexity of this algorithm is clearly dominated by the function that generates the groups. It is impractical for an algorithm to exhaustively search the solution space to find an optimal clustering of tasks. There is a large number

of possible heuristics to cluster tasks into groups. Our objective here is to use the input file affinity measure for clustering tasks in a way that scalability is improved with the use of the iterative techniques explained above. For that reason we propose a heuristic for clustering the groups, which we call I-Group and is described below in Fig.2:

-1. Define the number of tasks to be assigned to each processor depending on its relative speed, based on the average speed of the processors of the cluster. For instance, if the relative speed is 1.0 (equal to the average of the slave processors of the cluster), the number of tasks to be assigned to that processor is at most $\left\lceil \frac{T}{P} \right\rceil$. It is worth noting that this is only a first approximation, to be adjusted later by the heuristic.

-2. Compute the byte sum of all input files needed to execute each task on a slave processor ($Ifile_{sum}$).

-4. Sort all computed results by $Ifile_{sum}$. Smaller $Ifile_{sum}$ should appear on top of list.

-6. Tasks are group in this loop. In the heuristic presented in this section there is at most one group per processor at any time during execution. In the very beginning of the execution the number of groups is equal to the number of processors, since initially there is one group per processor. First the time with the smallest $Ifile_{sum}$ not yet assigned to a group is selected. This is done in order to minimize the time needed to start execution of the first task in a slave node.

-10. If the set of files associated to the next task in the list defined in (4) have an input file affinity greater than β to the set of files belonging to the task just assigned, then the next task in the list is assigned to the same processor. Note that β is a variable parameter, but normally it should be

greater than 0.5. The reason to do that is to maximize the input file affinity (I_{aff}) inside a group. Therefore, sending of similar sets of files to multiple processors can be avoided. If the set of files belonging to the next task in the list is different enough (i.e., (I_{aff}) between the two sets of files < 0.5), then the task located P positions after the task just assigned is selected. This is done for two reasons: first, to guarantee that the tasks with smaller $Ifile_{sum}$ are dispatched first to the processors. Second, to create groups as possible regarding the number of bytes that should be sent from the master node. Our objective here is to avoid the possibility of one group having tasks that depends only on small files, while other groups have tasks that depends only on larger files.

1. **For all** Processors define the number of tasks to be assigned
2. **For all** tasks
3. Compute the input files byte sum of the task ($Ifile_{sum}$).
4. Sort results of step (3) according to $Ifile_{sum}$ in list L
5. P = Number of processors
6. **For all** groups defined in step (1) (in non-increasing order of number of tasks, beginning with the largest group)
   {
7. Assign the task with the smallest $Ifile_{sum}$, not yet assigned, to the group
8. position = 1
9. **Until** the group is completed **do**
   {
10. **If** (Iaff(L[position], L[position+1]) < β) then position = (position + P) mod size(L)
11. **else** position = (position + 1) mod size (L)
12. Assign to the group the task located at position in list L
13. Remove assigned tasks from List L
    }
14. **End do**
15. P = P – 1
    }

Fig.2 I-Group – Heuristic for grouping tasks using the I_{aff} measure

Remember that input file transfers are done in a pipelined way, overlapping computation with communication when possible. Depending on the size of the group and the increment of position variable, the end of list L could be reached before completing the group. Note that the complexity of the heuristic I-Group, shown in Fig.2, is dominated by steps (2), (4) and (10). The loop represented by step (2) computes the $Ifile_{sum}$ for all tasks. The execution of step (2) can be implemented by processing all the T lists (only once) in time $O(T + D)$, where D is the number of dependency relations and T is the number of tasks. As step (4) sorts the tasks according to $Ifile_{sum}$, its complexity is $O(T.\log T)$. The complexity of steps (6) – (13) is dominated by step (10), which computes the I_{aff} for all pairs of adjacent tasks in the list. If ΔT denotes the maximum number of files a tasks depends upon, then the calculation of the I_{aff} for every single pair of tasks can be executed in time $O(\Delta T)$, and the calculation for all pairs can be executed in time $O(T.\Delta(T)$. Therefore, the complexity of the heuristic I-Group is $O(T + D + T.\log T + T.\Delta T)$. Furthermore, it is worth noting that $T.\Delta T \geq D$ and $T.\Delta T \geq T$. Therefore, the complexity of the heuristic can be simply denoted as $O(T.\log T + T.\Delta T)$.

4 PARALLEL IMPLEMENTATION

The implementation is done on a distributed computing environment (Armadillo Generation Cluster) consisting of 48 Intel Pentium at 1.73GHZ and 0.99GB RAM. Communication is through a fast Ethernet of 100 MBits per seconds 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 [25]. PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. 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. The Geranium Cadcam Cluster consisting of 48 Intel Pentium at 1.73GHZ and 0.99GB RAM. Communication is through a fast Ethernet of 100 MBits per seconds connected and running Linux. The cluster performance has high memory bandwidth with a message passing supported by MPI [23]. The program written in C, provided access to MPI through calling MPI library routines.

At each time-step we have to evaluate v^{n+1} values at $'lm'$ grid points, where $'l'$ is the number of grid points along x-axis. Suppose we are implementing this method on $R \times S$ mesh connected computer. Denote the workers by $P_{i1,j1} : i1 = 1,2,\ldots,R \text{ and } R < l,\ j1 = 1,2,\ldots,S \text{ and } S < M$. The workers P_{i1j1}, are connected as shown in Fig.3. Let $L_1 = \left[\frac{1}{R}\right]$ and $M_1 = \left[\frac{M}{S}\right]$ where $[\ ]$ is the smallest integer part. Divide the $'lm'$ grid points into $'RS'$ groups so that each group contains at most $(L_1+1)(M_1+1)$ grid points and at least $L_1 M_1$ grid points. Denote these groups by

$$G_{i1j1} : i1 = 1,2,\ldots,R,\ \ j1 = 1,2,\ldots,S.$$

$$\begin{array}{ccccccccc}
[P_{11}] & \leftrightarrow & [P_{21}] & \leftrightarrow & [P_{31}] & \leftrightarrow \cdots & \leftrightarrow & [P_{R1}] \\
\updownarrow & & \updownarrow & & \updownarrow & & & \updownarrow \\
[P_{12}] & \leftrightarrow & [P_{22}] & \leftrightarrow & [P_{32}] & \leftrightarrow \cdots & \leftrightarrow & [P_{R2}] \\
\updownarrow & & \updownarrow & & \updownarrow & & & \updownarrow \\
[P_{13}] & \leftrightarrow & [P_{23}] & \leftrightarrow & [P_{33}] & \leftrightarrow \cdots & \leftrightarrow & [P_{R3}] \\
\updownarrow & & \updownarrow & & \updownarrow & \cdots & & \updownarrow \\
\vdots & & \vdots & & \vdots & \cdots & & \vdots \\
\updownarrow & & \updownarrow & & \updownarrow & \cdots & & \updownarrow \\
[P_{1S}] & \leftrightarrow & [P_{2S}] & \leftrightarrow & [P_{3S}] & \leftrightarrow \cdots & \leftrightarrow & [P_{RS}]
\end{array}$$

5

Fig.3 R x S mesh connected workers

$$[P_{1j1}] \leftrightarrow [P_{2j1}] \leftrightarrow [P_{3j1}] \leftrightarrow \cdots \leftrightarrow [P_{Rj1}]$$

Fig.4 R x S mesh connected row-wise worker communication

Design G_{i1j1}, such that it contains the following grid points

$$G_{i1j1} = \left\{ \begin{array}{l} (X_{(i1-1)+1}, Y_{(j1-1)+j}): \\ i = 1,\ 2, \cdots,\ L_1 \text{ or } L_i + 1 \\ j = 1,\ 2, \cdots,\ M_1 \text{ or } M_1 + 1 \end{array} \right\}$$

Assign the group G_{i1j1}, to the workers $P_{i1,j1} : i_1 = 1,2,\ldots,R, \text{For, } j_1 = 1,2,\ldots,S.$ Each worker computes its assigned group $v_{i,j}^{n+1}$ values in the required number of sweeps. At the $(p+1/2)^{th}$ sweep the workers compute $v_{i,j}^{(p+1/2)th}$ values of its assigned groups. For the $(p+1/2)^{th}$ level the worker P_{i1j1} requires one value from the worker P_{i1-1j1} or P_{i1+1j1}, worker. In the $(p+1/2)^{th}$ level the communication between the workers is done row-wise as shown in Fig.4. After communication between the workers is completed then each worker P_{ij} computes the $v_{i,j}^{p+1/2}$ values. For the $(p+1)^{th}$ sweep each worker P_{i1j1} requires one value from the P_{i1-1j1} or P_{i1+1j1} worker. Here the communication between processors is done column-wise as shown in Fig.5. Then each worker computes the values $v_{i,j}^{(p+1)th}$ of its assigned group.

$$\begin{array}{c}
[P_{i1,1}] \\
\updownarrow \\
[P_{i1,2}] \\
\updownarrow \\
\vdots \\
\updownarrow \\
[P_{i1S}]
\end{array}$$

Fig.5 R x S mesh connected column-wise worker communication

Statements need to be inserted to select which portions of the code will be executed by each processor. The copy of the program is started by checking *pvm_parent,* it then spawns multiple copies of itself and passes them the array of *tids.* At this point, each copy is equal and can work on its data partition in collaboration with other workers. In the master model, the master program spawns and direct a number of worker program which perform computations. Any *pvm* task can initiate processes on the machine. The

master calls *pvm_mytid*, which as the first *pvm* call, enrolls this task in the *pvm* system. It then calls *pvm_spawn* to execute a given number of slave programs on other machines in *pvm*. Each worker calls *pvm_tid* to determine its task *id* in the virtual machine, and then uses the data broadcast from the master to create a unique ordering from 0 to *nprocessor-1*. Subsequently, *pvm_send* and *pvm_recv* are used to pass messages between processors. When finished, all *pvm* programs call *pvm_exit* to allow *pvm* to disconnect any sockets to the process and keep track of which processes are currently running.

4.1 MPI Communication Service Design

MPI like most other network-oriented middleware services communicates data from one worker to another across a network. However, MPI's higher level of abstraction provides an easy-to-use interface more appropriate for distributing parallel computing applications. We focus our evaluation on [23] because MPI serves as an important foundation for a large group of applications. Conversely, MPI provides a wide variety of communication operations including both blocking and non-blocking sends and receives and collective operations such as broadcast and global reductions. We concentrate on basic message operations: blocking send, blocking receives, non-blocking send, and non-blocking receive. Note that MPI provides a rather comprehensive set of messaging operations. MPI primitive communication operation is the blocking send to blocking receive. A blocking send (*MPI_Send)* does not return until both the message data and envelope have been safely stored. When the blocking sends returns, the sender is free to access and overwrite the send buffer. Note that these semantics allow the blocking send to compute even if no matching receive has been executed by the receiver. A blocking receives (*MPI_Recv)* returns when a message that matches its specification has been copied to the buffer.

Task 0	Task 1
#define size … *int sdata [size];* *int rdata[size];* *MPI_Status status;* *MPI_Request request;* *int tag = 20;* */* initialization */* */* … blocking send – receive 0 -> 1 */* */* fill sdata */* *MPI_Send(sdata, size, MPI_INT, 1, tag, MPI_COMM_WORLD);* */* use or overwrite sdata */* */* …non-blocking send-recv 1 -> 0 */* *MPI_Irecv(rdata, size, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);* */* computation excluding rdata */* *MPI_Wait(&request, &status);* */* use rdata */* */* finish */*	*#define size …* *int sdata [size];* *int rdata [size];* *MPI_Status status;* *MPI_Request request;* *int tag = 20;* */* initialization */* */* … blocking send-receive 0 -> 1 */* *MPI_Recv(rdata, size, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);* */* …non-blocking send-receive 1 -> 0 */* */* fill sdata */* *MPI_Isend(sdata, size, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);* */* computation excluding sdata */* *MPI_Wait(&request, &status);* */* use or overwrite sdata */* */* finish*/*

Fig.6 Example of message operations with MPI

However, an alternative to blocking communication operations MPI provides non-blocking communication to allow an application to overlap communication and computation. This overlap improves application performance. In non-blocking communication, initialization and completion of communication operations are distinct. The second message operation in Fig.6, illustrates one message transfer from task 1 to task 0 using non-blocking send and non-blocking receive, respectively.

A non-blocking send has both a send start call (*MPI_Isend)* initializes the send operation and it return before the message is copied from the send buffer. The send complete call (*MPI_Wait)* completes the non-blocking send by verifying that the data has been copied from the send buffer. It is this separation of send start and send complete that provides the application with the opportunity to perform computations. Task 1, in Fig.6, uses *MPI_Isend* to initiate the transfer of sdata to task 0. During the time between *MPI_Isend* and *MPI_Wait*, Task 1 can not modify sdata because the actual copy of the message from sdata is not guaranteed until the *MPI_Wait* call returns. After *MPI_Wait* returns, Task 1 is free to use or overwrite the data in sdata. Similarly, a non-blocking receive has both a receive start call and a receive complete call. The receive start call *(MPI_Irecv)* initiates the receive operation and it may return before the incoming message is copied into the receive buffer. The receive complete call *(MPI_Wait)* completes the non-blocking receive by verifying that the data has been copied into the buffer. As with non-blocking send, the application has the opportunity to perform computation between the receive start and receive complete calls. Task 0 in Fig.6, uses *MPI_Irecv* to initiate the receive of sdata from Task 1 during the time between *MPI_Irecv* and *MPI_Wait*, Task 0 cannot read or modify sdata because the message from Task 1 is not guaranteed to be in this buffer until the *MPI_Wait* call returns. After *MPI_Wait* returns, Task 0 is free to need rdata.

4.2 Speedup, Efficiency and Effectiveness

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 [15]. Speedup is the ratio of the serial time to the parallel version run on *N* workers. Efficiency is the ability to judge how effective the parallel algorithm is expressed as the ratio of the speedup to *N* workers. In traditional parallel systems it is widely define as:

$$S(N) = T(s)/T(N), \qquad E(N) = S(N)/N \quad (4.1)$$

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* workers, *E(n)* is the total efficiency for the parallel algorithm. However, this simple definition has been

focused on constant improvements. A generalized speedup formula is the ratio of parallel to sequential execution speed. 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.

$$E(N) = E_{num}(N)E_{par}(N)E_{load}(N), \quad (4.2)$$

where E_{num}, is the numerical efficiency that 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 workers. E_{par} is the parallel efficiency which is defined as the ratio of CPU time taken on one worker to that on N workers. The parallel efficiency and the corresponding speedup are commonly written as follows:

$$S_{par}(N) = T(1)/T(N), \; E_{par}(N) = S_{par}(N)/N \quad (4.3)$$

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 workers can be written as follows:

$$T(N) = T_m(N) + T_{sd}(N) + T_{sc}(N) \quad (4.4)$$

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

$$T_m(N) = T_m(1), \; T_{sd}(N) = T_{sd}(1), \quad T_{sc}(N) = T_{sc}(1)/N, \quad (4.5)$$

therefore, the speedup can be written as:

$$S_{par}(N) = \frac{T(1)}{T(N)} = \frac{T_{ser}(1)+T_{sc}(1)}{T_{ser}(1)+T_{sc}(1)/N} < \frac{T_{ser}(1)+T_{sc}(1)}{T_{ser}(1)} \quad (4.6)$$

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:

$$Epar(N) = \frac{T(1)}{nT(N)} = \frac{Tser(1)+Tsc(1)}{nTser(1)+Tsc(1)} < \frac{Tser(1)+Tsc(1)}{nTser(1)} \quad (4.7)$$

the parallel efficiency represents the effectiveness of the parallel program running on N workers 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, $T_{B=B}^{N_1L}(N)$ denotes the total CPU time of the parallel version of the program with B blocks run on N workers 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 can be decomposed as follows:

$$E(N) = \frac{T_s^{N_o}(1)}{n.T_{B=B}^{N_1L}(N)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)} \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \frac{T_{B=B}^{N_0}(1)}{T_{B=B}^{N_1}(1)} \frac{T_{B=B}^{N_1}(1)}{T_{B=B}^{N_1}(N)} \frac{T_{B=B}^{N_1}(N)}{T_{B=B}^{N_1L}(N)}, \quad (4.8)$$

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 (4.5) and (4.2), we obtain:

$$E_{load}(N) = \frac{T_{B=B}^{N_1}(N)}{T_{B=B}^{N_1L}}, E_{par}(n) = \frac{T_{B=B}^{N_1}(1)}{n.T_{B=B}^{N_1}(N)}, \quad Enum(N) = \frac{T_s^{N_o}(1)}{T_{B=B}^{N_1}(1)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)} \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \frac{T_{B=B}^{N_o}(1)}{T_{B=B}^{N_1}(1)}, \quad (4.9)$$

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,

$$E_{num}(N) = E_{dd}\frac{N_o}{N_1}, \quad E_{dd} = \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \quad (4.10)$$

we call (4.10) 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 (4.10) represents the increase in the number of iterations required by the parallel method to achieve a specified accuracy compared to the serial method. The effectiveness is given by:

$$L_n = S_n / C_n \qquad (4.11)$$

where $C_n = nT_n$, T_1 is the execution time on a serial machine and T_n is the computing time on parallel machine with N processors. Hence, effectiveness can be written as:

$$L_n = S_n / (nT_n) = E_n / T_n = E_n S_n / T_1 \qquad (4.12)$$

which clearly shows that L_n is a measure of both speedup and efficiency. Therefore, a parallel algorithm is said to be effective if it maximizes L_n and hence $L_n T_1 = S_n E_n$.

Table 1 The wall time T_W, the master time T_M, the worker data time T_{SD}, the worker computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 200x200, with B = 50 blocks and *Niter* = 100 for PVM and MPI.

Schemes	N	T_w	T_m	T_{sd}	T_{sc}	PVM			MPI		
						T	S_{par}	E_{par}	T	S_{par}	E_{par}
	1	3874	135	58	1639	1832	1.000	1.000	1528	1.000	1.000
	2	3014	134	56	798.13	988.13	1.854	0.927	791.30	1.931	0.966
	4	2491	134	56	437.61	627.61	2.919	0.703	440.60	3.468	0.867
	8	1938	134	56	278.19	468.19	3.913	0.489	369.44	4.136	0.517
MF-DS	16	1356	134	56	124.40	314.40	5.827	0.364	238.08	6.418	0.401
	20	1117	134	56	90.4	280.64	6.528	0.326	218.44	6.995	0.350
	24	938	134	56	51.27	241.27	7.593	0.316	195.12	7.831	0.326
	30	899	134	56	19.95	209.95	8.726	0.291	167.76	9.108	0.304
	38	806	124	56	10.58	190.58	9.613	0.253	149.35	10.231	0.269
	48	718	114	46	1.72	161.72	11.328	0.236	127.52	11.982	0.250
	1	2314	53	23	1169.6	1245.66	1.000	1.000	1083.63	1.000	1.000
	2	1968	53	19	632	704.13	1.769	0.885	566.46	1.913	0.957
	4	1341	51	19	407.61	477.61	2.608	0.652	365.10	2.968	0.742
	8	1089	51	19	305.97	375.97	3.313	0.414	285.99	3.789	0.474
MF-DS	16	862	51	19	164.62	234.62	5.309	0.332	182.92	5.924	0.370
	20	718	51	19	133.36	203.36	6.125	0.306	166.41	6.512	0.326
	24	701	51	19	107.54	177.54	7.016	0.292	146.73	7.385	0.308
	30	629	51	19	83.44	153.44	8.118	0.271	127.08	8.527	0.284
	38	611	51	19	68.2	138.20	9.013	0.237	114.51	9.463	0.249
	48	528	51	19	47.31	117.31	10.618	0.221	97.47	11.118	0.232

Table 2 The wall time T_W, the master time T_M, the worker data time T_{SD}, the worker computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 300x300, with B = 50 blocks and *Niter* = 100 for PVM and MPI.

Schemes	N	T_w	T_m	T_{sd}	T_{sc}	PVM			MPI		
						T	S_{par}	E_{par}	T	S_{par}	E_{par}
	1	2968	116	53	1454	1623	1.000	1.000	1328	1.000	1.000
	2	2575	114	51	700.6	865.6	1.875	0.938	689.87	1.925	0.963
	4	2388	114	51	377.63	542.63	2.991	0.748	439.59	3.021	0.755
	8	2011	114	51	308.04	473.04	3.431	0.429	340.95	3.895	0.487
MF-DS	16	1932	114	51	131.06	296.06	5.482	0.343	217.6	6.103	0.381
	20	1634	114	51	84.19	249.19	6.513	0.326	195.35	6.798	0.340
	24	1538	114	51	57.79	222.79	7.285	0.304	175.48	7.568	0.315
	30	1184	114	51	24.56	189.56	8.562	0.285	154.19	8.613	0.287
	38	1099	114	51	10.71	175.71	9,237	0.243	136.89	9.701	0.255
	48	987	114	31	0.11	145.11	11.185	0.233	123.43	10.759	0.224

5 Results and Discussion

5.1 Benchmark Problem

We implement the MS-DF schemes on the 2-D TE using the input file affinity measure, with the values of the physical properties in our test case chosen in such a way that *LC* and *(RC + GL)* are equal to one. The application of the above mentioned algorithms were compared in terms of performance by simulating their executions in master-worker paradigm on several sizes. We assume a platform composed of variable number of heterogeneous processors. The solution domain was divided into rectangular blocks. The experiment is demonstrated on meshes of 200x200 and 300x300 for block sizes of 50, 100 and 200 respectively, both for MPI and PVM. Tables 1 - 14 show the various performance timing.

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 200x200, with B = 100 blocks and *Niter* = 100 for PVM and MPI.

Table 4 The wall time T_W, the master time T_M, the worker data time T_{SD}, the worker computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 300x300, with B = 100 blocks and *Niter* = 100 for PVM and MPI.

Schemes	N	T_w	T_m	T_{sd}	T_{sc}	PVM			MPI		
						T	S_{par}	E_{par}	T	S_{par}	E_{par}
	1	3382	151	96	2074	2321	1.000	1.000	1825	1.000	1.000
	2	3141	148	92	979.01	1219.01	1.904	0.952	945.11	1.931	0.966
	4	2961	148	92	437.07	677.07	3.428	0.857	489.93	3.725	0.931
	8	2749	147	92	401.98	640.98	3.621	0.453	413.08	4.418	0.552
MF-DS	16	2421	147	92	165.50	404.50	5.738	0.359	283.52	6.437	0.402
	20	2097	147	92	100.08	339.08	6.845	0.342	254.00	7.185	0.359
	24	1862	147	92	66.07	305.07	7.608	0.317	230.20	7.928	0.330
	30	1718	147	92	24.87	263.87	8.796	0.293	202.98	8.991	0.300
	38	1601	147	92	14.24	236.57	9.811	0.258	182.28	10.012	0.263
	48	1497	137	92	12.18	205.02	11.321	0.236	166.15	10.984	0.229

Consider the TE of the form:

$$\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2}=\frac{\partial^2 U}{\partial t^2}+\frac{\partial U}{\partial t}+U \quad (5.1)$$

The boundary condition and initial condition posed are:

$$\left.\begin{aligned} U(0,y,t)&=0\\ U(1,y,t)&=100\\ U(x,0,t)&=0\\ U(x,1,t)&=100 \end{aligned}\right\} \quad t\geq 0 \qquad (5.1a)$$

$$U(x,y,0)=e^{xy}, \qquad (5.1b)$$

5.2 Parallel Efficiency

To obtain a high efficiency, the worker computational time $T_{sc}(1)$ should be significantly larger than the serial time

Schemes	N	T_w	T_m	T_{sd}	T_{sc}	PVM			MPI		
						T	S_{par}	E_{par}	T	S_{par}	E_{par}
	1	5328	128	63	2327	2518	1.000	1.000	2267	1.000	1.000
	2	4692	127	60	1165.31	1352.31	1.862	0.931	1160.78	1.953	0.977
	4	3968	127	60	557.75	744.75	3.381	0.845	609.24	3.721	0.930
	8	2887	127	60	369.1	556.10	4.528	0.566	471.41	4.809	0.601
MF-DS	16	2126	127	60	213	400	6.295	0.393	337.65	6.714	0.420
	20	1974	127	60	173.13	360.13	6.992	0.350	309.87	7.316	0.366
	24	1684	127	60	134.54	321.54	7.832	0.326	282.46	8.026	0.334
	30	1322	127	60	94.09	281.09	8.958	0.299	243.29	9.318	0.311
	38	1181	127	60	68.04	255.04	9.873	0.260	213.18	10.634	0.280
	48	967	127	60	27.88	214.88	11.718	0.244	184.14	12.311	0.256

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 of 200x200 to 300x300 are for various numbers of sub-domains; from B = 50 to 200 are listed in Tables 1 – 6 for PVM and MPI application. In these tables we listed the wall (elapsed) time for the master task, T_W, (this is necessarily greater than the maximum wall time returned by the workers), the master CPU time, T_M, the average worker computational time, T_{SC} and the average worker data communication time T_{SD} all in seconds. The speed-up and efficiency versus the number of processors are shown in Fig.7(a,b,c) and Fig. 8(a,b) respectively, with block number B as a parameter. The results show that the parallel efficiency increases with increasing grid size for given block number both for MPI and PVM 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 the wall time increases with the number of processors as observed from the figures and table.

The total CPU time is composed of three parts: the CPU time for the master task, the average worker CPU time for data communication and the average worker 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 workers after each iteration since a worker can be assigned a different sub-domain under the pool-of-task paradigm. The master task includes sending updated data to workers, assigning the task tid to workers, waiting for message from processors and receiving the result from workers. For a given grid size, the CPU time to send task tid to workers increase with block number, but the timing for other tasks does not change significantly with block number. In Tables 1 – 6, 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 worker (T_1), (2) assigning task to worker (T_2), (3) waiting for the worker to execute tasks (T_3), (4) receiving the results (T_4).

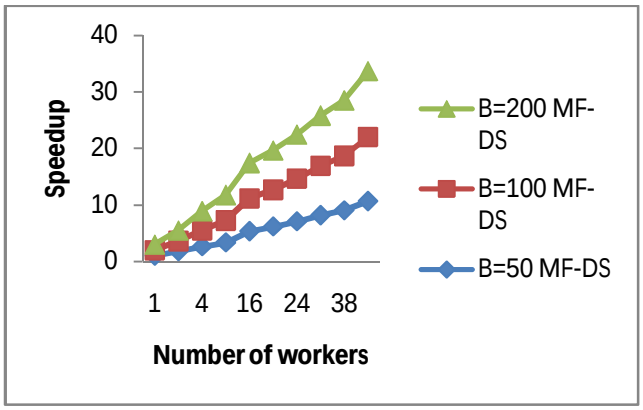
Table 5 The wall time T_W, the master time T_M, the worker data time T_{SD}, the worker computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 200x200, with B = 200 blocks and *Niter* = 100 for PVM and MPI.

Table 6 The wall time T_W, the master time T_M, the worker data time T_{SD}, the worker computational time T_{SC}, the total time T, the parallel speed-up S_{par} and the efficiency E_{par} for a mesh of 300x300, with B = 200 blocks and *Niter* = 100 for PVM and MPI.

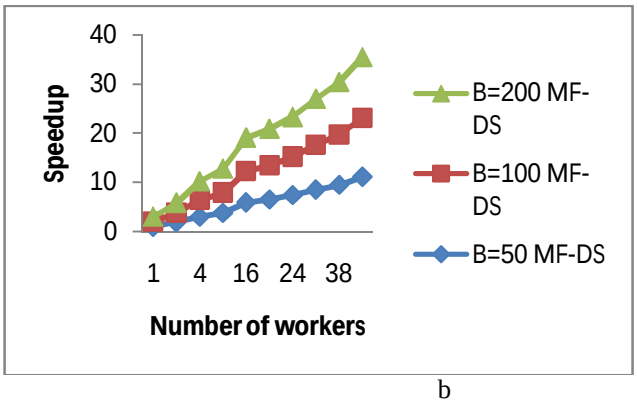
Schemes		T_w	T_m	T_{sd}	T_{sc}	PVM			MPI		
						T	S_{par}	E_{par}	T	S_{par}	E_{par}
	1	5982	172	174	2836	3182	1.000	1.000	2618	1.000	1.000
	2	4634	169	173	1284.79	1626.79	1.956	0.978	1329.61	1.969	0.985
	4	4182	169	173	475.78	817.78	3.891	0.973	667.35	3.923	0.981
	8	3763	169	173	296.19	638.19	4.986	0.623	495.18	5.287	0.661
MF-DS	16	3211	169	173	145.51	487.51	6.527	0.408	357.94	7.314	0.457
	20	2727	169	173	98.78	440.78	7.219	0.361	307.31	8.519	0.426
	24	2189	169	173	51.13	393.13	8.094	0.337	272.59	9.604	0.400
	30	1962	169	173	19.28	337.76	9.421	0.314	259.00	10.108	0.337
	38	1724	169	173	15.74	300.64	10.584	0.279	223.42	11.718	0.308
	48	1510	169	173	11.92	269.02	11.828	0.246	207.38	12.624	0.263

Table7 Effectiveness of the various schemes with PVM and MPI for 300 x 300 mesh size

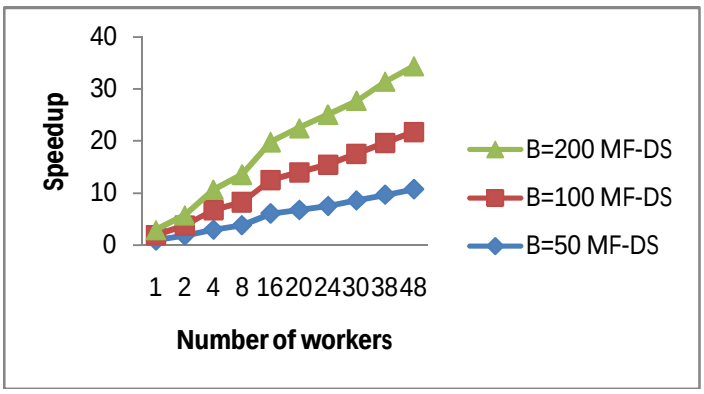
	N	PVM T(s)	L_n	MPI T(s)	L_n
	2	1626.79	0.060	1329.61	0.074
MF-DS	8	638.19	0.098	495.18	0.133
	16	487.51	0.084	357.94	0.128
	20	440.78	0.082	307.31	0.139
	30	337.76	0.093	259.00	0.130
	48	269.02	0.091	207.38	0.127



a

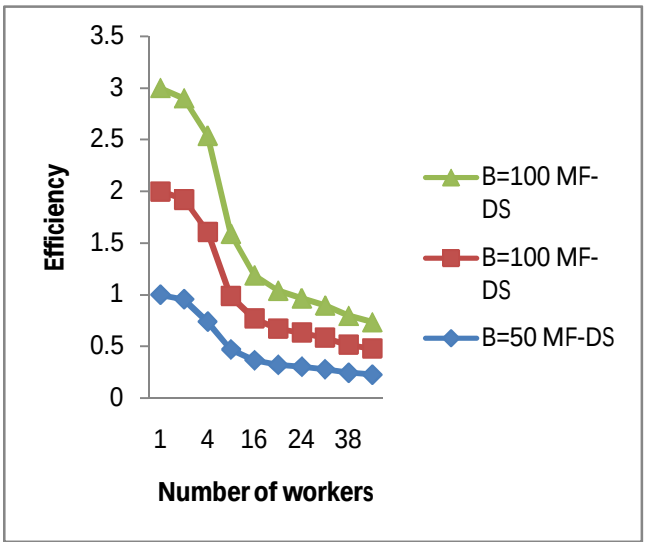


b

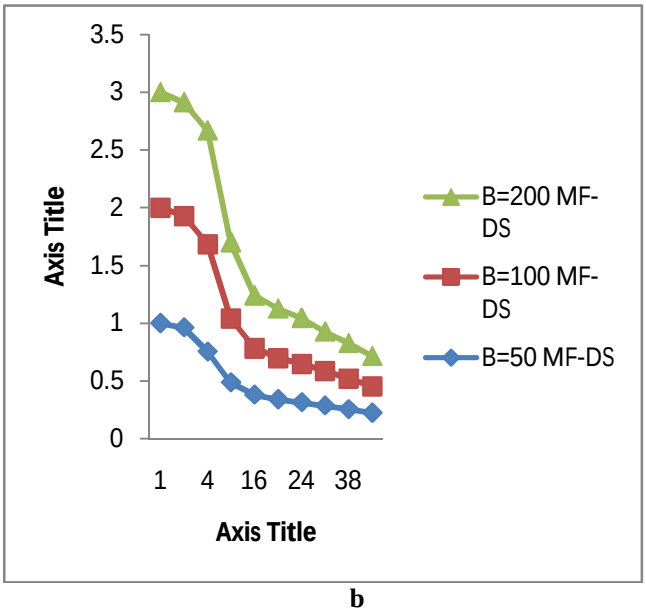


c.

Fig.7 Speed-up versus the number of workers for various block sizes. **a** mesh 200x200 PVM, **b** mesh 200x200 MPI, **c** mesh 300x300 MPI



a



b

Fig.8 Parallel efficiency versus the number of workers for various block sizes. a mesh 200x200 MPI, b mesh 300x300 MPI

Table 7 shows the effectiveness of the various schemes with PVM and MPI. As the number of processor increases, the effectiveness of MF-DS scheme performs significantly better than the ADI. As the total number of processor increases, the bottleneck of parallel computers appears and the global reduction consumes a large part of time, we anticipate that the improvement will move to be significant. Fig.7 and Fig.8 show the efficiency using PVM and MPI with varying block sizes, and observed that the MPI implementation in Fig. 8 converges slightly better using the I_{aff} measure.

5.3 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. (6.10). 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 Tables 9 and 10, we listed the total CPU time distribution over various grid sizes and block numbers running with only one processor for PVM and MPI. Using this table, the DD efficiency E_{DD} can be calculated and the results are shown in Fig.9 and Fig.10. Note that the DD efficiency can be greater than one, even with one processor. Fig.9 and 10 show 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 describe the increase in the number of iterations required by the parallel method to achieve a specified accuracy as compared 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 11 and 12 with Fig.12 and 13, and Table 13 and 14 with Fig.12, 13 and 14.

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 9 The worker computational time T_{SC}, for 100 iterations as a function of various block numbers

Scheme	*NIxNJ*	*PVM*				
		B=20	*B=30*	*B=50*	*B=100*	*B=200*
	200x200	*797*	*981*	*1169.6*	*1639*	*2327*
MF-DS	*300x300*	*943*	*1268*	*1454*	*2074*	*2836*

Table 10 The worker computational time T_{SC}, for 100 iterations as a function of various block numbers

Scheme	*NIxNJ*	*MPI*				
		B=20	*B=30*	*B=50*	*B=100*	*B=200*
	200x200	*728*	*864*	*992*	*1464*	*1996*
MF-DS	*300x300*	*852*	*1096*	*1218*	*1986*	*2334*

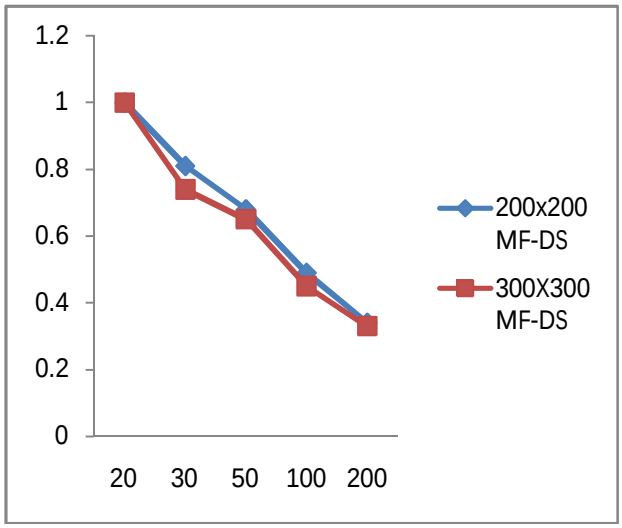


Fig.9 The DD efficiency versus the number of sub-domains for various blocks of PVM

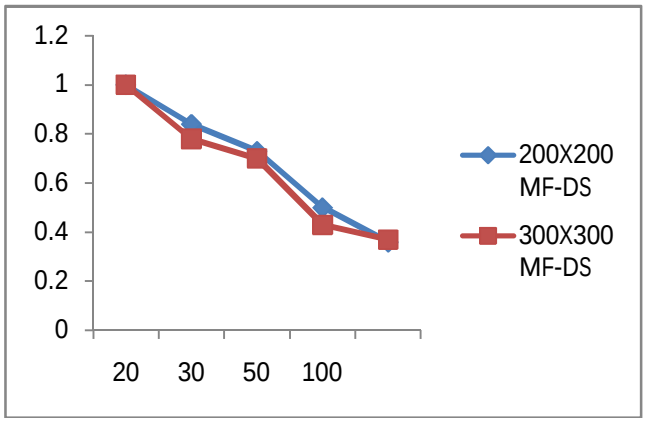


Fig.10 The DD efficiency versus the number of sub-domains for various blocks of MPI

Table11 The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 200x200 for PVM

Scheme	N	B=20	B=50	B=100	B=200
MF-DS	2	3621	4347	4621	3118
	4	3621	4508	4792	3346
	8	3621	4921	5121	3598
	16	3621	5248	5448	3771
	30	3621	5422	5611	3964
	48	3621	5745	5908	4125

Table12 The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 200x200 for MPI

Scheme	N	B=20	B=50	B=100	B=200
MF-DS	2	1914	3125	3541	2392
	4	1914	3448	3679	2586
	8	1914	3703	3824	2793
	16	1914	4094	4219	2899
	30	1914	4268	4387	3016
	48	1914	4491	4666	3325

Table13 The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 300x300 for PVM

Scheme	N	B=20	B=50	B=100	B=200
MF-DS	2	1914	3125	3541	2392
	4	1914	3448	3679	2586
	8	1914	3703	3824	2793
	16	1914	4094	4219	2899
	30	1914	4268	4387	3016
	48	1914	4491	4666	3325

Table14 The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 300x300 for MPI

Scheme	N	B=20	B=50	B=100	B=200
MF-DS	2	3101	4321	4852	3211
	4	3101	4582	5074	3528
	8	3101	4718	5362	3769
	16	3101	4992	5621	3928
	30	3101	5284	5895	4325
	48	3101	5476	6122	4518

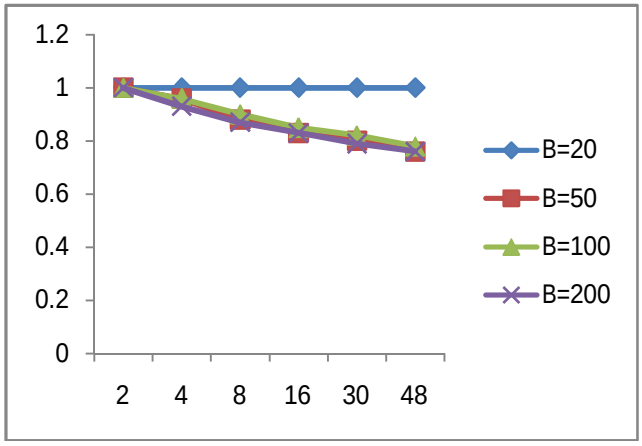


Fig.11 Convergence behavior with domain decomposition for mesh 200x200 MF-DS PVM

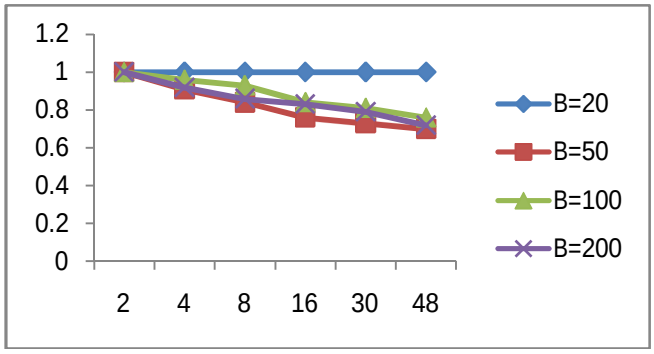


Fig.12 Convergence behavior with domain decomposition for mesh 200x200 MF-DS MPI

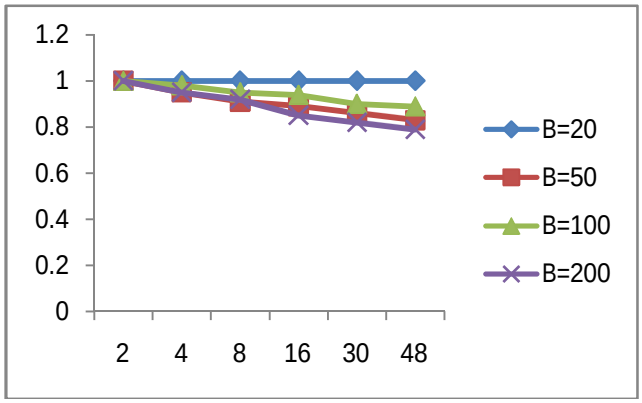


Fig.13 Convergence behavior with domain decomposition for mesh 300x300 MF-DS PVM

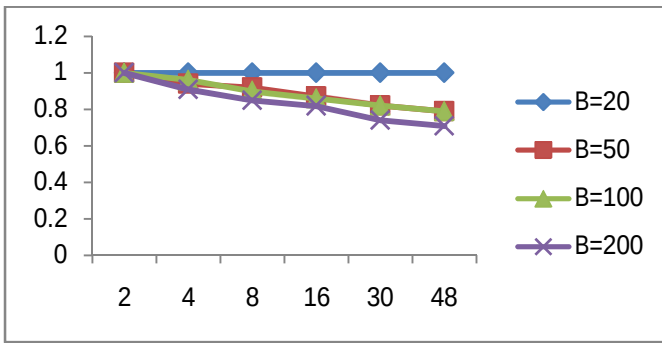


Fig.14 Convergence behavior with domain decomposition for mesh 300x300 MF-DS MPI

5.4 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 200x200 and 300x300 have been showed respectively. From Eq. (6.8), 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. With the MPI version we were able to achieve best implementation for the efficiency of different mesh sizes with different block numbers as observed in Fig. 14 and 16. In comparison of Fig. 14 and 16 on convergence behavior with domain decomposition for different mesh sizes, we observed that the implementation with MPI achieved more conformity to unity

6. CONCLUSION

In this paper we have presented the results of the experimental study into using input file affinity measure to schedule tasks in a master-worker paradigm using PVM and MPI on MF-DS scheme. The aim was to implement the execution performance of the said applications, the utilization of master-worker paradigm and provide some recommendations regarding the feasibility of this approach suggest a scheduling method and confirm experimental results achieved by other researchers. The performance results demonstrate the effectiveness of the alternating schemes. It not only outperforms stationary iterative schemes, but also approach good convergence. Computational results obtained have clearly shown the benefits of using parallel algorithms. We have come to some conclusions that: (1) the parallel efficiency is strongly dependent on the problem size, block numbers and the number of processors as observed in Fig.6 both for PVM and MPI. (2) A high parallel efficiency can be obtained with large scale problems. (3) The decomposition of domain greatly influences the performance of the parallel computation (Fig. 12 – 13). (4) 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 for both MPI and PVM (Fig.14 – 15). The speedup was because of the computation/communication interleaving approach as observed in the figures and tables. On the basis of the current parallelization strategy, more sophisticated models can be attacked efficiently.

References

[1.] J. M. Alonso, J. Mawhin, R. Ortega, (1999). Bounded Solution of Second-Order Semilinear Evolution Equations and Applications to Telegraph Equation. J. Math Pures Appl. 78, 49 - 63

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

[3.] E. Arubanel, (2011). Scheduling Tasks in the Parareal Algorithm. Parallel Computing 37, 172 – 182

[4.] Berkeley Open Infrastructure for Network Computing, 2002. http://boinc.berkeley.edu/

[5.] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, (2003). Adaptive Computing on the Grid using AppLes. IEEE Transactions on Parallel and Distributed Systems, 14(4), pp 369 - 382

[6.] D. Callahan, K. Kennedy. 1988. Compiling Programs for Distributed Memory Multiprocessors. Journal of Supercomputer 2, pp 151 – 169

[7.] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, (2000). Heuristics for Scheduling the Ninth Heterogeneous Computing Workshop, IEEE Computer Society Press

[8.] T. Chan, F. Saied, 1987. Hypercube Multiprocessors. SIAM, Philadelphia

[9.] H. Chi-Chung, G. Ka-Kaung, et. al., 1994. Solving Partial Differential Equations on a Network of Workstations. IEEE, pp 194 – 200.

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

[11.] 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

[12.] D'Ambra P., M. Danelutto, Daniela S., L. Marco, 2002. Advance Environments for Parallel and Distributed Applications: a view of current status. Parallel Computing 28, pp 1637 – 1662.

[13.]H.S Dou, Phan-Thien. 1997. A Domain Decomposition Implementation of the Simple Method with PVM. Computational Mechanics 20 pp 347 – 358
[14.]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
[15.]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
[16.]M. S. El-Azah, M. El-Gamel, (2007). Appl. Math Comput. 190, 757 - 764
[17.]D.J Evans, B. Hassan, 2003. Numerical Solution of the Telegraph Equation by the AGE Method. International Journal of Computer Mathematics Vol. 80, number 10, pp 1289 – 1297
[18.] D.J. Evans, M.S. Sahimi, The Alternating Group Explicit Iterative Method for Parabolic Equations I: 2-Dimensional Problems, *Intern. J. Compt. Math,* Vol. 24, (1988) pp. 311-341
[19.]S. U. Ewedafe, R. H. Shariffudin, (2011). Armadillo Generation Distributed System with Geranium Cadcam Cluster for solving 2-D Telegraph Problem. Intern. J. Compt. Math, vol. 88, 589 – 609
[20.] S. U. Ewedafe, R. H. Shariffudin, (2011). Parallel Implementation of 2-D Telegraphic Equation on MPI/PVM Cluster. Int. J. Parallel Prog, 39, 202 - 231
[21.]D.S. Fabricio, H. Senger (2009). Bag of Task running on Master-Slave with Input File. Parallel Computing 35, pp 57 – 71
[22.]C. Fan, C. Jiannong, S. Yudong (2003). High Abstractions for Message Passing Parallel Programming. Parallel Computing 29, 1589 – 1621.
[23.]I. Foster, J. Geist, W. Groop, E. Lust, 1998. Wide-Area Implementations of the MPI. Parallel Computing 24 pp 1735 – 1749.
[24.]A. Geist A. Beguelin, J. Dongarra, 1994. Parallel Virtual Machine (PVM). Cambridge, MIT Press
[25.]G. A Geist, V. M Sunderami, 1992. Network Based Concurrent Computing on the PVM System. Concurrency Practice and Experience, pp 293 – 311
[26.]A. Giersch, Y. Robert, F. Vivien, (2006). Scheduling Task Sharing Files on Heterogeneous Master-Slave Platforms. Journal of System Architecture, 52(2) pp 88 – 104
[27.]Y. Guang-Wei, 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
[28.]Y. Guangwei, H. Xudeng (2007). Parallel Iterative Difference Schemes Based on Prediction Techniques for Sn Transport Method. Applied Numerical Mathematics 57, 746 – 752.
[29.]L. B. Hinde, C. Perez, T. Priol, (2010). Extending Software Component Models with the Mastr-Worker Paradigm. Parallel Computing 36, 86 - 103
[30.]K. Jaris, D.G. Alan, 2003. A High-Performance Communication Service for Parallel Computing on Distributed Systems. Parallel Computing 29, pp 851 – 878
[31.]K. Kaya, C. Aykanat, (2006). Iterative Improvement-Based Heuristics for Adapting Scheduling of Task Sharing Files on Heterogeneous Master-Slave. IEEE Transactions on Parallel and Distributed Systems, 17(8), pp 883 - 896
[32.]A. Keren, A. Barak, (2003). Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster. IEEE Transactions on Computer 14(1), 39 - 50
[33.]E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Leboisky, (2001). SETI@home-Massively Distributed Computing for SETI. IEEE Computer Society, Los Alamitos, CA, USA, 78 – 83
[34.]J. L. Lions, Y. Maday, G. Turinici, (2001). A Parareal in Time Discretization of PDEs. Comptes Rendus de Academie des Sciences – Series 1 – Mathematics 332(7), 661 - 668
[35.]M. Lixing, C. Frederick, J. Harris, 1998. Technical Report Department of Computer Science University of Nevada Reno, NV 89557
[36.]A. C. Metaxas, R. J. Meredith, (1993). Industrial Microwave, Heating, Peter Peregrinus, London
[37.]A. R. Mitchell, G. Fairweather, (1964). Improved forms of the Alternating direction methods of Douglas, Peaceman and Rachford for solving parabolic and elliptic equations, Numer. Maths, 6, 285 – 292.

AUTHOR

Ewedafe Simon Uzezi received the B.Sc. and M.Sc. degrees in Industrial-Mathematics and Mathematics respectively from Delta State University and The University of Lagos in 1998 and 2003. He further obtained his Ph.D. in Numerical Parallel Computing 2010 from the University of Malaya, Malaysia. In 2011 he joined UTAR in Malaysia as an Assistant Professor and later lectured in Oman and Nigeria as Senior Lecturer in Computing. Currently he is an Associate Professor.