International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org
Volume 3, Issue 5, September-October 2014 **ISSN 2278-6856**

Domain Decomposition of 2-D Bio-Heat Equation on MPI and PVM Clusters

Ewedafe Simon Uzezi[1], **Rio Hirowati Shariffudin**[2]

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

[2]Institute of Mathematical Sciences, Faculty of Science,
University of Malaya, 50603 Kuala Lumpur, Malaysia.

Abstract
In this paper, a parallel implementation of the Iterative Alternating Direction Explicit method by D'Yakonov (IADE-DY) to solve 2-D Bio-Heat equation on a distributed system using Message Passing Interface (MPI) and Parallel Virtue Machine (PVM) are presented. The parallelization of the program is implemented by a domain decomposition strategy. A Single Program Multiple Data (SPMD) model is employed for the implementation. The Single Program Multiple Data (SPMD) model was employed for the implementation. The platform gives us better predictions of the effects of thermal physical properties on the transient temperature of biological tissues, which can be used for thermal diagnosis in medical practices. The implementation is discussed in relation to means of the parallel performance strategies and analysis. The model enhances overlap communication and computation to avoid unnecessary synchronization, hence, the method yields significant speedup. We present some analyses that are helpful for speedup and efficiency. It is concluded that the efficiency is strongly dependent on the grid size, block numbers and the number of processors for both MPI and PVM. Different strategies to improve the computational efficiency are proposed.

Keywords: 2-D Bio-Heat Equation, Domain Decomposition, PVM, MPI

1. INTRODUCTION

Studying Bio-heat transfer in human body has been a hot topic and is useful for designing clinical thermal treatment equipments, for accurately evaluating skin burn and for establishing thermal protections for various purposes. The Bio-Heat transfer is the heat exchange that takes place between the blood vessels and the surrounding tissues. Monitoring the blood flow using the techniques has great advantage in the study of human physiological aspects. This require a mathematical model which relate the heat transfer between the perfuse tissue and the blood. The theoretical analysis of heat transfer design has undergone a lot of research over years, from the popular Penne's Bio-heat transfer equation proposed in 1948 to the latest one proposed by [10]. Many of the Bio-heat transfer problem by Penne's account for the ability of the tissue to remove heat by diffusion and perfusion of tissue by blood. Predictions of heat transport have been carried out in [6, 23]. The major concern in the modeling of the Bio-heat is the accurate continuum representation of the heat transfer in the living tissue incorporating the effect of blood flow, due to presence of two heat sources. Penne's assumes that for heat transfer to take place there must be two heat sources, as in heat produced by the metabolism and the heat transfer from the blood flow surrounding tissue at each point of the forearm. Effects of thermal properties and geometrical dimensions on the skin burn injuries have been discussed, [23, 33] proposed a comparison of 1D and 2D programmed for predicting the state of skin burn. Among the various model proposed to study the heat transfer in living tissues, the Penne's equation is the most widely used one. It is based on the classical Fourier law [28], and has been greatly simplified after introducing the intuitive concept of blood perfusion, the blood flow rate per unit tissue volume, to study the Bio-heat transfer and assessment of skin burns. 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 [1, 2, 3 and 7]. Attempts have also been made towards parallel solutions on distributed memory MIMD machines; see [5, 9 and 27]. 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 [5]. Relevant data need to be passed from processor to processor through a message passing mechanism [9, 14 and 27]. In this paper, we use the SPMD as a technique to achieve parallelism under the domain decomposition strategy. SPMD is the most common style of parallel programming [13]. The SPMD model contains only a single program with multiple data and each process element will execute the sequential parts individually, while all the processing elements will execute the concurrent parts jointly using the message passing communication primitives. Moreover, the code of an SPMD application can typically be structured into three major components: (1) the single code which is replicated for the execution of a task, (2) the load balancing strategy and (3) a skeleton which initializes and terminates the application, manages the tasks and controls the execution of other parts. Hence, the Master-Slave concurrent programming model is typical of the SPMD model. To help with the program development under a distributed computing environment, a number of software tools have been developed. PVM is chosen here since it has a large user group [16] and the MPI [18 and 37] which simplified numerous issues for both application developers. Historically, users have written scientific applications for large distribution memory computers using explicit communications as the programming model. In this paper, we present the implementation of the alternating schemes on MPI/PVM clusters with the domain decomposition method and the Single Program Multiple Data (SPMD) model to obtain results with sufficient accuracy taking note on different speedup and efficiency of various mesh sizes. We solve the 2-D Bio-Heat equation by using the double sweep methods of Peaceman and Rachford (DS-PR) [26] and Mitchell and Fair-weather (DS-MF) [24, 25 and 36]. Each 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 tridiagonal system of equations that arises from the difference method applied is then solved by using the two-stage Iterative Alternating Decomposition Explicit method of D'Yakonov (IADE-DY) developed in [34]. We computed some examples to test the parallel algorithm. The effects of the various parameters on the performance of the algorithms are discussed. The prime objective of our platform is not to be specific to one problem; it should be able to solve a wide variety of time-dependent partial differential equations (PDE) for various applications [5, 12, 38 and 39]. This paper is organized as follows: Section 2 introduces the model for the 2-D Bio-Heat and introduces the ADI and IADE-DY method. Section 3 introduces the parallel implementation. Section 4 introduces the results and discussions. Finally, a conclusion is included in section 5.

1.1 Previous research work

The Alternating Direction Implicit [ADI] method for the partial differential equations (PDEs) proposed by Peaceman and Rachford [26 and 19] 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., [32] have proposed a parallel ADI solver for linear array of processors. Chan and Saied have implemented ADI scheme on hypercube. The ADI method in [26] has been used to solve heat equation in 2-D. Several approaches to solve the Bio-Heat equation numerically have been carried out in [4, 10, 23, 28 and 33]. Our approach compared to [6, 10, 23, 28 and 33] is the application of the difference scheme on a distributed system of MPI/PVM cluster to evaluate the effects of the algorithm on different mesh sizes with the use of the domain decomposition parallel implementation. We determine the various speedups and efficiency through domain decomposition method. Our results compared to [15] give better conformity to linearity for speedup and closeness to unity for efficiency. Our implementation compared to [6, 10 and 28] is a way of proofing stability and convergence in parallel platform on a distributed system. We also note the various constant improvements on speedup, efficiency and performance analysis in [10].

2. BIO-HEAT EQUATION

Modern clinical treatments and medicines such as cryosurgery, cryopreservation, cancer hyperthermia, and thermal disease diagnostics, require the understanding of thermal life phenomena and temperature behavior in living tissues [22]. Studying Bio-heat transfer in human body has been a hot topic and is useful for designing clinical thermal treatment equipments, for accurately evaluating skin burn, and for establishing thermal protections for various purposes. The theoretical analysis of heat transfer design has undergone a lot of research over years, from the popular Penne's Bio-heat transfer equation proposed in 1948 to the latest one proposed by [23]. The well known Penne's equation and the energy balance for a control volume of tissue with volumetric blood flow and metabolism yields the general Bio-heat transfer equations. ρ , c_p are densities and specific heat of tissue, ω_b and c_b are blood perfusion rate and specific heat of blood, q'''_m is the volumetric metabolic heat generation, U_a is the arterial temperature, U is the nodal temperature. The bio-heat problem is given as:

$$\rho c_p \frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \omega_b c_b (U_a - U) + q'''_m,$$

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

$$\frac{\partial U}{\partial t} = \frac{1}{pc_p}\frac{\partial^2 U}{\partial x^2} + \frac{1}{\rho c_p}\frac{\partial^2 U}{\partial y^2} + \frac{\omega_b c_b}{\rho c_\rho} U_a - \frac{\omega_b c_b}{\rho c_\rho} U + \frac{q'''_m}{pc_p}, \qquad (2.1)$$

This further simplifies into the form:

$$\frac{\partial U}{\partial t} = \frac{1}{pc_p}\frac{\partial^2 U}{\partial x^2} + \frac{1}{\rho c_\rho}\frac{\partial^2 U}{\partial y^2} - \frac{\omega_b c_b}{\rho c_\rho} U + \frac{\omega_b c_b}{\rho c_\rho}\left(U_a + \frac{q'''_m}{\omega_b c_b}\right) \qquad (2.2)$$

assume q'''_m to be constant, and denote $U^{\oplus} = U_a + \frac{q'''_m}{\omega_b c_b}$, $b = \frac{\omega_b c_b}{\rho c_p} (> 0)$ and $c = \frac{1}{\rho c_p} (> 0)$, we can obtain the simplified form of the 2-D Penne's equation with the initial and boundary conditions given below:

$$\frac{\partial U}{\partial t} = c\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right) - bU + bU^{\oplus}, \qquad (2.3)$$

with initial condition

$$U(x, y, 0) = f(x, y) \qquad (2.4)$$

and boundary conditions

$$\left.\begin{array}{l} U(0, y, t) = f_1(y, t),\ U(1, y, t) = f_2(y, t) \\ U(x, 0, t) = f_3(x, t),\ \ U(x, 1, t) = f_4(x, t) \end{array}\right\} \qquad (2.5)$$

when the explicit scheme is used, we write using the same finite-difference scheme:

$$\frac{U_{i,j}^{n+1} - U_{i,j}^{n}}{\Delta t} = c\left(\begin{array}{c} \dfrac{U_{i+1,j}^{n} - 2U_{i,j}^{n} + U_{i-1,j}^{n}}{\Delta x^2} + \\ \dfrac{U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}}{\Delta x^2} \end{array}\right)$$
$$- bU_{i,j}^{n} + bU^{\oplus}$$

the temperature of the node in the scheme formulation takes the form:

$$U_{i,j}^{n+1} = c\frac{\Delta t}{\Delta x^2}\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j+1}^{n} + U_{i,j-1}^{n}\right)$$
$$-\left(1 - 4c\frac{\Delta t}{\Delta x^2} - b\Delta t\right)U_{i,j}^{n} + b\Delta t U^{\oplus} \qquad (2.6)$$

2.1 ADI Method (2-D Bio-Heat)

The ADI method is originally developed by [26], a time step $(n \rightarrow n+1)$ is provided into two half time steps $(n \rightarrow n+1/2 \rightarrow n+1)$. The time derivative is represented by forward difference and the spatial derivatives are represented by central differences. In the first half time step $(n \rightarrow n+1/2)$, $\partial^2 U/\partial x^2$ is expressed at the end $n+1/2$ and $\partial^2 U/\partial y^2$ is expressed at the start n. Therefore:

$$\frac{U_{i,j}^{n+1/2} - U_{i,j}^{n}}{\Delta t/2} = \frac{c}{2}\left[\begin{array}{l} \dfrac{U_{i+1,j}^{n+1/2} - 2U_{i,j}^{n+1/2} + U_{i-1,j}^{n+1/2}}{(\Delta x)^2} \\ + \\ \dfrac{U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}}{(\Delta y)^2} \end{array}\right] \quad (2.7$$
$$-\frac{b}{2}\left(U_{i,j}^{n+1/2} + U_{i,j}^{n}\right) + \frac{bU^{\oplus}}{2}$$

In the second time half-time step, $\partial^2 U/\partial x^2$ is expressed at the start $n+1/2$ and $\partial^2 U/\partial y^2$ is expressed at the end $n+1$. Therefore:

$$\frac{U_{i,j}^{n+1} - U_{i,j}^{n+1/2}}{\Delta t/2} = \frac{c}{2}\Big(\frac{U_{i+1,j}^{n+1/2} - 2U_{i,j}^{n+1/2} + U_{i-1,j}^{n+1/2}}{(\Delta x)^2}$$
$$+ \frac{U_{i,j+1}^{n+1} - 2U_{i,j}^{n+1} + U_{i,j-1}^{n+1}}{(\Delta y)^2}\Big) \qquad (2.8$$
$$-\frac{b}{2}(U_{i,j}^{n+1} + U_{i,j}^{n+1/2}) + \frac{bU^{\oplus}}{2}$$

to see why the spatial derivatives can be written at different time in the two half-time steps in (2.7) and (2.8), we add them to get:

$$\frac{U_{i,j}^{n+1} - U_{i,j}^{n}}{2(\Delta t/2)} = c\Big[\frac{U_{i+1,j}^{n+1/2} - 2U_{i,j}^{n+1/2} + U_{i-1,j}^{n+1/2}}{(\Delta x)^2}$$
$$+ \frac{1}{2}\left(\begin{array}{l} \dfrac{U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}}{(\Delta y)^2} \\ + \dfrac{U_{i,j+1}^{n+1} - 2U_{i,j}^{n+1} + U_{i,j-1}^{n+1}}{(\Delta y)^2} \end{array}\right)\Big] \qquad (2.9)$$
$$- b\left(U_{i,j+1}^{n+1} + U_{i,j}^{n}\right) + bU^{\oplus}$$

this shows that by going through the two half-time steps, the Bio-heat equation is effectively represented at the half-time step $n+1/2$, using central difference for the time derivative, central difference for the *x*-derivatives, and central difference for the *y*-derivative by averaging at the $(n+1/2)^{th}$ and $(n+1)^{th}$ step. The ADI method for one complete time step is thus second-order accurate in both time and space. Re-arranging (2.8) and (2.9), we get:

$$-F_x U_{i-1,j}^{n+1/2} + (2F_x + b\Delta t + 2)U_{i,j}^{n+1/2} -$$
$$F_x U_{i+1,j}^{n+1/2} = F_y U_{i,j-1}^{n} + (-2F_y - b\Delta t + 2)U_{i,j}^{n} \quad (2.10)$$
$$+ F_y U_{i,j+1}^{n} + b\Delta t U^{\oplus}$$

let $a = (2F_x + b\Delta t + 2),\ b = c = -F_x$. For various values of i and j, (2.10) can be written in a more compact matrix form at the $(n+1/2)^{th}$ time level as:

$$AU_j^{(n+1/2)} = f_n, \quad j = 1, 2, \ldots n. \qquad (2.11)$$

where

$U = (U_{1,j}, U_{2,j}, \ldots, U_{m,j})^T,\ f = (f_{1,j}, f_{2,j}, \ldots f_{m,j})^T$

at the $(n+1)^{th}$ time level, sub-iteration 2 is given by:

$$-F_y U_{i,j-1}^{n+1} + (2F_y + b\Delta t + 2)U_{i,j}^{n+1} - F_y U_{i,j+1}^{n+1} = F_x U_{i-1,j}^{n+1/2} + (-2F_x - b\Delta t + 2)U_{i,j}^{n+1/2} + F_x U_{i+1,j}^{n+1/2} + b\Delta t U^{\oplus} \quad (2.12$$

let $a = (2F_y + b\Delta t + 2),\ b = c = -F_y$. For various values of *i* and *j* , (2.11) can be written in a more compact matrix form as:

$$BU_i^{(n+1)} = g_{n+1/2}, \qquad i = 1,2,\ldots m \qquad (2.13)$$

where

$U_i^{(n+1)} = (U_{i,1}, U_{i,2}, \ldots, U_{i,n})^T,\ g = (g_{i,1}, g_{i,2}, \ldots, g_{i,n})^T$

and $F_x = c\Delta t/(\Delta x)^2,\ F_y = c\Delta t/(\Delta y)^2$.

2.2 IADE-DY (2-D Bio-Heat)

The matrices derived from the discretization resulting to A in Eq. (2.11) and B in Eq. (2.13) are respectively tri-diagonal of size (*mxm*) and (*nxn*). Hence, at each of the $(n+1/2)^{th}$ and $(n+1)^{th}$ time levels, these matrices can be decomposed into $G_1 + G_2 - \frac{1}{6}G_1G_2$, where $G_1\ and\ G_2$ are lower and upper bi-diagonal matrices given respectively by

$$G_1 = [l_i, 1], \qquad and \qquad G_2 = [e_i, u_i], \qquad (2.14)$$

Where

$$e_1 = \frac{6}{5}(a-1),\ u_i = \frac{6}{5}b,\ e_{i+1} = \frac{6}{5}\left(a - \frac{1}{6}l_i u_i - 1\right),\ l_i = \frac{6c}{6-e_i}\ (e_i \neq 6)\ i = 1,2,\ldots,m-1$$

Hence, by taking *p* as an iteration index, and for a fixed acceleration parameter *r > 0*, the two-stage IADE-DY scheme of the form:

$$(rI + G_1)u^{(p+1/2)} = (rI - gG_1)(rI - gG_2)u^{(p)} + hf \quad and$$
$$(rI + G_2)u^{(p+1)} = u^{(p+1/2)} \qquad 2.15)$$

can be applied on each of the sweeps (2.11) and (2.13). Based on the fractional splitting strategy of D'Yakonov, the iterative procedure is accuracy, and is found to be stable and convergent. By carrying out the relevant multiplications in Eq. (2.15), the following equations for computation at each of the intermediate levels are obtained:

(i) at the $(p+1/2)^{th}$ iterate,

$$u_1^{(p+1/2)} = \frac{1}{\hat{d}}(s_1\hat{s}u_1^{(p)} + w_1\hat{s}u_2^{(p)} + hf_1)$$

$$u_i^{(p+1/2)} = \frac{1}{\hat{d}}(-l_{i-1}u_{i-1}^{(p+1/2)} + v_{i-1}s_{i-1}u_{i-1}^{(p)} + (v_{i-1}w_{i-1} + s_i\hat{s})u_i^{(p)} + w_i\hat{s}u_{i+1}^{(p)} + hf_i),$$

$i = 2,3,\ldots,m-1$

$$u_m^{(p+1/2)} = \frac{1}{\hat{d}}(-l_{m-1}u_{m-1}^{(p+1/2)} + v_{m-1}s_{m-1}u_{m-1}^{(p)} + (v_{m-1}w_{m-1} + s_m\hat{s})u_m^{(p)} + hf_m) \qquad (2.16)$$

where,

$$g = \frac{6+r}{6},\ h = \frac{r(12+r)}{6},\ \hat{d} = 1+r,\ \hat{s} = r-g,\ s_i = r - ge_i,\ i = 1,2,\ldots,m$$

and $v_i = -gl_i,\ \ w_i = -gu_i \quad i = 1,2,\ldots,m-1$.

(ii) at the $(p+1)^{th}$ iterate,

$$u_m^{(p+1)} = \frac{u_m^{(p+1/2)}}{d_m},$$

$$u_i^{(p+1)} = \frac{1}{d_i}(u_i^{(p+1/2)} - \hat{u}_i u_{i+1}^{(p+1)}),\ where\ d_i = r + e_i,\ i = m-1, m-2, \ldots, 2, 1 \qquad (2.17)$$

3. PARALLEL IMPLEMENTATION

3.1 The cluster system

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, located at the University of Malaya. The cluster performance has high memory bandwidth with a message passing supported by PVM which is public-domain software from Oak Ridge National Laboratory [17]. 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 running Linux, located at the University of Malaya. The cluster performance has high memory bandwidth with a message passing supported by MPI [18]. The program is written in C and provides access to MPI through calling MPI library routines..

3.2 Domain decomposition

The parallelization of the computations is implemented by means of grid partitioning technique [8 and 20]. 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 implementation of the algorithm

At each time-step we have to evaluate u^{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 processors by $P_{i1,j1} : i1=1,2,\ldots,R$ and $R<l$, $j1=1,2,\ldots,S$ and $S<M$.

The processors P_{i1j1}, are connected as shown in Fig. 1. 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_1M_1 grid points. Denote these groups by

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

$$\begin{tikzcd}
{[P_{11}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{21}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{31}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & \cdots \arrow[r, leftrightarrow] & {[P_{R1}]} \arrow[d, updownarrow] \\
{[P_{12}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{22}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{32}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & \cdots \arrow[r, leftrightarrow] & {[P_{R2}]} \arrow[d, updownarrow] \\
{[P_{13}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{23}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & {[P_{33}]} \arrow[r, leftrightarrow] \arrow[d, updownarrow] & \cdots \arrow[r, leftrightarrow] \arrow[d, phantom, "\cdots"] & {[P_{R3}]} \arrow[d, updownarrow] \\
\vdots \arrow[d, updownarrow] & \vdots \arrow[d, updownarrow] & \vdots \arrow[d, updownarrow] & \cdots \arrow[d, phantom, "\cdots"] & \vdots \arrow[d, updownarrow] \\
{[P_{1S}]} \arrow[r, leftrightarrow] & {[P_{2S}]} \arrow[r, leftrightarrow] & {[P_{3S}]} \arrow[r, leftrightarrow] & \cdots \arrow[r, leftrightarrow] & {[P_{RS}]}
\end{tikzcd}$$

Fig. 1

$$\begin{tikzcd}
{\lfloor P_{1j1} \rfloor} \arrow[r, leftrightarrow] & {\lfloor P_{2j1} \rfloor} \arrow[r, leftrightarrow] & {\lfloor P_{3j1} \rfloor} \arrow[r, leftrightarrow] & \cdots \arrow[r, leftrightarrow] & {\lfloor P_{Rj1} \rfloor}
\end{tikzcd}$$

Fig. 2

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 processor $P_{i1,j1} : i_1 = 1,2,\ldots,R$, For, $j_1 = 1,2,\ldots,S$. Each processor computes its assigned group $u_{i,j}^{n+1}$ values in the required number of sweeps. At the $(p+1/2)^{th}$ sweep the processors compute $u_{i,j}^{(p+1/2)th}$ values of its assigned groups. For the $(p+1/2)^{th}$ level the processor P_{i1j1} requires one value from the processor P_{i1-1j1} or P_{i1+1j1} processor. In the $(p+1/2)^{th}$ level the communication between the processors is done row-wise as shown in Fig. 2. After communication between the processors is completed then each processor P_{ij} computes the $u_{i,j}^{p+1/2}$ values. For the $(p+1)^{th}$ sweep each processor P_{i1j1} requires one value from the P_{i1-1j1} or P_{i1+1j1} processor. Here the communication between processors is done column-wise as shown in Fig. 3. Then each processor computes the values $u_{i,j}^{(p+1)th}$ of its assigned group. The algorithm can be transformed to master-slave model by sending out the computing tasks on each block to each processor in the 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 1-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. 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.

3.4 Parallel algorithm description

With the data allocation scheme our various iterative schemes for computing the solution at $(p+1)-\text{th}$ time-step from the current $(p+1/2)-th$ time-step is the following:

1) Compute the solution $u^{p+1/2}$ on the interface boundary (i.e. on B). Then send the computed solution $u^{p+1/2}$ on interface boundaries to its neighbor as follows.

a) For processor $P_{i,j}$, if a right-neighbor is present (i.e. processor $P_{i+1,j}$), then processor $P_{i,j}$ sends the interior vertical interface boundary to the processor $P_{i+1,j}$. A total of $n-1$ elements are sent to the right neighbor (i.e. processor $P_{i,j}$).

b) If processor $P_{i,j}$ has a left neighbor (i.e. processor $P_{i-1,j}$), $P_{i,j}$ receives the predicted interior vertical interface boundary from processor $P_{i-1,j}$. A total of $n-1$ elements are received by processor $P_{i,j}$.

c) If processor $P_{i,j}$ has an upper-neighbor (i.e. processor $P_{i,j+1}$), the $P_{i,j}$ sends the interior horizontal interface boundary to processor $P_{i,j+1}$ (i.e. its upper-neighbor). A total of $m-1$ data elements are sent to the upper-neighbor (i.e. processor $P_{i,j+1}$).

d) If processor $P_{i,j}$ has a down-neighbor (i.e. processor $P_{i,j-1}$), then $P_{i,j}$ receives the interior horizontal boundary from processor $P_{i,j-1}$. A total of $m-1$ data elements are received by processor $P_{i,j}$.

e) If upper-neighbor (i.e. processor $P_{i,j+1}$), is present to processor $P_{i,j}$,

i) If processor $P_{i,j}$ also has a left neighbor (i.e. processor $P_{i-1,j}$), $P_{i,j}$ sends the solution at the sampling point neighboring ip_4 to processor $P_{i-1,j}$. Here a total of 1 data is sent to the left-neighbor (i.e. processor $P_{i-1,j}$).

ii) If right-neighbor (i.e. processor $P_{i+1,j}$) is also present to processor $P_{i,j}$, then $P_{i,j}$ receives the solution at the sampling point neighboring intersection point ip_3 from processor $P_{i+1,j}$. Here 1 element is received by $P_{i,j}$.

f) If right-neighbor (i.e. processor $P_{i+1,j}$) is present

i) If processor $P_{i,j}$ also has a lower- neighbor (i.e. processor $P_{i,j-1}$), $P_{i,j}$ sends the computed solution at the sampling point neighboring ip_2 to processor $P_{i,j-1}$. The amount of data sent to processor $P_{i,j-1}$ is 1.

ii) If processor $P_{i,j}$ also has an upper-neighbor (i.e. processor $P_{i,j+1}$), $P_{i,j}$ receives the computed solution at sampling point neighboring ip_3 from processor $P_{i,j+1}$. The amount of data received by processor $P_{i,j}$ is 1.

The data computed in step1 provide the solution at time-step $p+\frac{1}{2}$ at the interface boundary conditions.

2) Compute solution u^{p+1}. The data communicated in this step is the following:

a) Considering processor $P_{i,j}$ assigned sub-domain $\Omega_{i,j}$, if $P_{i,j}$ has an upper-neighbor (i.e. processor $P_{i,j+1}$) and a right-neighbor (i.e. processor $P_{i+1,j}$), then $P_{i,j}$ sends the computed solution of the interface boundary intersection point to $P_{i,j+1}$ and $P_{i+1,j}$.

b) If processor $P_{i,j}$ has an upper-neighbor $P_{i,j+1}$ and also a left-neighbor $P_{i-1,j}$, then $P_{i,j}$ receives the computed value from $P_{i-1,j}$.

c) If processor $P_{i,j}$ has a right-neighbor $P_{i+1,j}$ and also a down-neighbor $P_{i,j-1}$, processor $P_{i,j}$ receives the computed solution from $P_{i,j-1}$.

d) If processor $P_{i,j}$ has a down-neighbor $P_{i,j-1}$, $P_{i,j}$ sends the solution at sampling points neighboring to $P_{i,j-1}$. A total of $(m-1)$ data elements are sent by $P_{i,j}$, where m is the number of grid-points in the sub-domain along x-axis.

e) If processor $P_{i,j}$ has an upper-neighbor $P_{i,j+1}$, $P_{i,j}$ receives the solution computed for the sampling points neighboring b_4 from processor $P_{i,j+1}$. Here $m-1$ data are received by processor $P_{i,j}$.

f) If processor $P_{i,j}$ has a left-neighbor (i.e. processor $P_{i-1,j}$), $P_{i,j}$ sends the solution computed at the sampling points neighboring to processor $P_{i-1,j}$. Here, $(n-1)$ elements are transferred by processor $P_{i,j}$ to processor $P_{i-1,j}$, where n is the number of grid-points in the sub-domain along y-axis.

g) If processor $P_{i,j}$ has a right-neighbor (i.e. processor $P_{i+1,j}$), $P_{i,j}$ receives the computed solution of the sampling points neighboring b_3 from processor $P_{i+1,j}$.

3.5 Parallel computation performance (time)

To correlate the communication activity with computation, we counted events between significant PVM/MPI call sites [21]. The execution overhead decreases at the same rate that the number of tasks increases, which indicates good scaling. All the required input files are generated during the partitioning phase. Each processor reads the corresponding input file and grid file and performs computation on the local domain. At the end of the computation of each phase, data is exchange between the neighboring processors using the libraries and computation for the next phase proceeds in parallel in each processor [29]. For each processor, if the boundary data from time (t-level) to time (t-1) have not yet arrived, the node computes part of the points in time t which do not make use of the boundary data (pre-computation). The idea of pre-computation is to calculate portion of points in time t before all the boundary points arrived. When the pre-computation is completed at time t, and the data has not yet arrived, the node can pre-compute the data at time

$(t+1)$ using the available data from time $(t-level+1)$ to time t. This process is repeated until no data can be pre-computed anymore. It is necessary to allocate additional memory to hold the pre-computed results. 1 level of pre-computation is sufficient such that a maximum overall reduction in elapsed time can be achieved. For the two-phase algorithm, the total time used in time step $n, T_{2p}(n),$ is given by:

$$T_{2p}(n) = T_{comp}(n) + T_{send}(n) + T_{wait}(n) + T_{recv}(n) \quad (3.1)$$

where T_{comp} is the computation time, T_{send} is the time used in sending the messages, T_{wait} is the time used by the platform in waiting for the incoming messages, and T_{recv} is the time used in processing the incoming messages. Clearly, there is a period of idle time during the $T_{wait}(n)$ period, and it is possible to perform pre-computation so as to overlap $T_{wait}(n)$ and $T_{comp}(n+1)$. It is impossible to overlap $T_{wait}(n)$ and $T_{recv}(n)$ since the platform cannot process the incoming message before they arrive. Similarly, it is impossible to overlap $T_{postcompute}(n)$ and $T_{send}(n+1)$ since the data required to be sent in time $(n+1)$ is not ready. Considering two cases, we have:

Case 1. $T_{wait}(n) \le T_{pre-compute}(n+1)$: After pre-computation at time step $(n+1)$ is computed, the message at time step n arrives. Thus, no waiting time is needed for incoming messages, and the elapsed time is:

$$\begin{aligned} T_{best}(n) &= T_{postcompute}(n) + T_{send}(n) + \\ &\quad T_{precompute}(n) + T_{recv}(n) \quad (3.2) \\ &= T_{comp}(n) + T_{send}(n) + T_{recv}(n) \end{aligned}$$

where $T_{postcompute}(n)$ is the computation time of the points in time step n that cannot be computed during the pre-computation phase. Note that the term $T_{compute}(n)$ does not appear in the equation because it is decomposed into two terms, $T_{pre-compute}(n)$ $and T_{postcompute}(n)$. The above term is the shortest elapsed time achievable since all the three components involve computations and thus cannot be overlapped.

Case 2. $T_{wait}(n) > T_{precompute}(n+1)$: In this case, the computation time is less than the waiting time. The shortest time achievable in each time step is:

$$T_{best} = T_{send}(n) + T_{wait}(n) + T_{recv}(n) + T_{postcomput}(n) \quad (3.3)$$

here, $T_{precompute}(n)$ is included into $T_{wait}(n-1)$ while $T_{postcompute}(n)$ remains distinct. Although it is possible to perform more levels of pre-computation during the $T_{wait}(n)$ period, this waiting period cannot be reduced further since the incoming messages arrive only after this period (i.e., $T_{recv}(n)$ cannot be started before $T_{wait}(n)$ ends). Following the above argument, it is easy to see that 1 level of pre-computation is sufficient to obtain the shortest elapsed time since performing more than 1 level of pre-computation can only shift some computations earlier.

3.6 MPI communication service design

MPI like most other network-oriented middleware services communicates data from one processor 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 [15] 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. Figure 3: 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. 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. 3, 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. 3, 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.

3.7 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 [11 and 13]. 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. In traditional parallel systems it is widely define as:

$$S(n) = {T(s)}/{T(n)}, \qquad E(n) = {S(n)}/{n} \qquad (3.4)$$

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 focused on constant improvements. A generalized speedup formula is the ratio of parallel to sequential execution speed. A thorough study of speedup models with their advantages and disadvantages are presented by Sahni [35]. 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.

$$E(n) = E_{num}(n) E_{par}(n) E_{load}(n), \qquad (3.5)$$

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 processors. E_{par} is the parallel efficiency which is defined 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:

$$S_{par}(n) = {T(1)}/{T(n)}, \qquad E_{par}(n) = {S_{par}(n)}/{n} \qquad (3.6)$$

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:

$$T(n) = T_m(n) + T_{sd}(n) + T_{sc}(n) \qquad (3.7)$$

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,

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*/

$$T_m(n) = T_m(1), \quad T_{sd}(n) = T_{sd}(1), \quad T_{sc}(n) = {T_{sc}(1)}/{n}, \qquad (3.8)$$

therefore, the speedup can be written as:

$$S_{par}(n) = \frac{T(1)}{T(n)} = \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1) + T_{sc}(1)/n} < \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1)} \qquad (3.9)$$

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:

$$Epar(n) = \frac{T(1)}{nT(n)} = \frac{Tser(1) + Tsc(1)}{nTser(1) + Tsc(1)} < \frac{Tser(1) + Tsc(1)}{nTser(1)} \qquad (3.10)$$

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, $T_{B=B}^{N_1 L}(n)$ 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:

$$E(n)=\frac{T_s^{N_o}(1)}{n.T_{B=B}^{N_1 L}(n)}=\frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)}\frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \quad (3.11)$$

$$\frac{T_{B=B}^{N_0}(1)}{T_{B=B}^{N_1}(1)}\frac{T_{B=B}^{N_1}(1)}{T_{B=B}^{N_1}(n)}\frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1 L}(n)},$$

where $T_{B=B}^{N_1}(n)$ has the same meaning as $T_{B=B}^{N_1 L}(n)$ except the idle time is not included. Comparing (3.8) and (3.5), we obtain:

$$E_{load}(n)=\frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1 L}},E_{par}(n)=\frac{T_{B=B}^{N_1}(1)}{n.T_{B=B}^{N_1}(n)},$$

$$Enum(n)=\frac{T_s^{N_o}(1)}{T_{B=B}^{N_1}(1)}= \quad (3.12)$$

$$\frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)}\frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)}\frac{T_{B=B}^{N_o}(1)}{T_{B=B}^{N_1}(1)},$$

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,

$$E_{num}(n)=E_{dd}\frac{N_o}{N_1}, \quad E_{dd}=\frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \quad (3.13)$$

we call (3.13) 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 (3.13) represents the increase in the number of iterations required by the parallel method to achieve a specified accuracy compared to the serial method.

3.8 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 [11, 30 and 31]: 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

We implemented the IADE-DY and the ADI scheme on the 2-D Bio-Heat equations. 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 100x100, 200x200 and 400x400 respectively both for MPI and PVM. Tables 1 - 5 show the various performance timing.

$$\frac{\partial U}{\partial t}=c\left(\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2}\right)-bU+bU^{\oplus}, \quad (4.1)$$

The boundary condition and initial condition posed are:

$$\left.\begin{array}{l}U(0,y,t)=0\\U(1,y,t)=0\\U(x,0,t)=0\\U(x,1,t)=0\end{array}\right\} \quad t\geq 0 \quad (4.2)$$

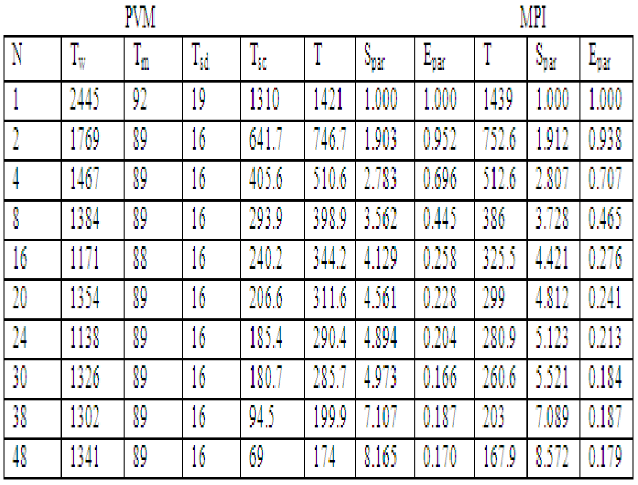
The cell size was chosen as $\Delta x=\Delta y$. The values of the physical properties in our test cases are chosen to be $\rho=1000kg/m^3,\ c=c_b=4200J/kg^0c,\ w_b=0.5kg/m^3$, temperature is set to be $U_0=12^0 c$. Table I provides a comparison of the accuracy of the methods under consideration in terms of absolute error.

Table 1 Sequential results for 2-D Bio-Heat Equation with various schemes

Method	ADI	IADE-DY
Av. Abs. Err.	5.8×10^{-5}	4.2×10^{-6}
RMS	5.6×10^{-6}	4.7×10^{-7}
It	7	5
Δx	1×10^{-1}	1×10^{-1}
Δt	2×10^{-4}	2×10^{-4}
λ	5×10^{-1}	5×10^{-1}
t	1.8×10^{-3}	1.8×10^{-3}
eps	1×10^{-4}	1×10^{-4}

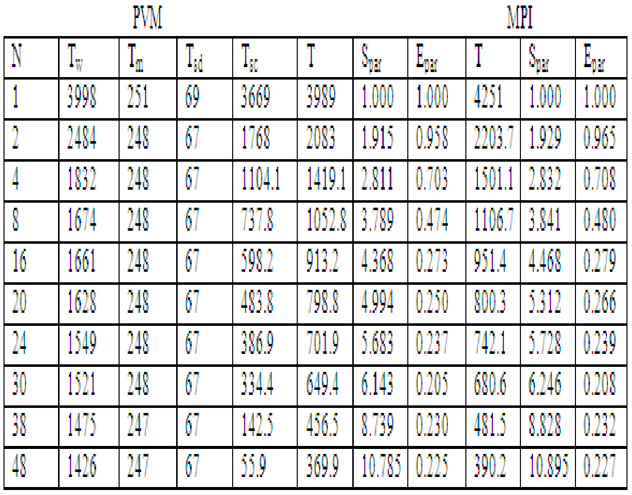
4.2. Parallel Efficiency

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 of 100x100 to 400x400 are for various numbers of sub-domains; from *B* = 50 to 200 are listed in Tables 3 – 11for PVM and MPI application. On the 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 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 shown in Fig. 5(a,b,c) and Fig. 6(a,b,c) respectively, with block number *B* as a parameter. The results below 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 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 a given grid size, the CPU time to send task tid to slaves increases with block number, but the timing for other tasks does not change significantly with block number. In Tables 3 – 11 we can 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).



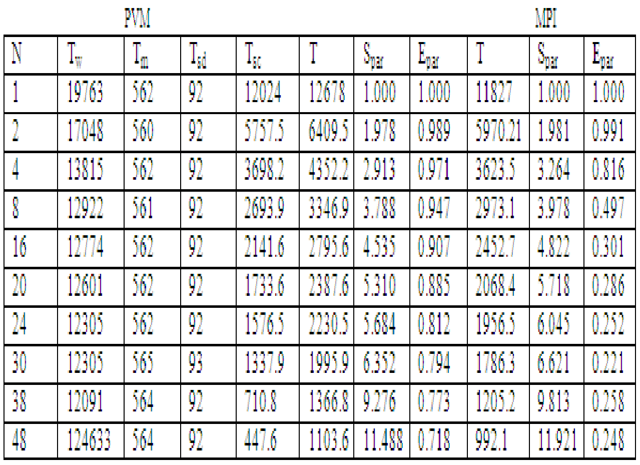
	PVM							MPI		
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}	T	S_{par}	E_{par}
1	2445	92	19	1310	1421	1.000	1.000	1439	1.000	1.000
2	1769	89	16	641.7	746.7	1.903	0.952	752.6	1.912	0.938
4	1467	89	16	405.6	510.6	2.783	0.696	512.6	2.807	0.707
8	1384	89	16	293.9	398.9	3.562	0.445	386	3.728	0.465
16	1171	88	16	240.2	344.2	4.129	0.258	325.5	4.421	0.276
20	1354	89	16	206.6	311.6	4.561	0.228	299	4.812	0.241
24	1138	89	16	185.4	290.4	4.894	0.204	280.9	5.123	0.213
30	1326	89	16	180.7	285.7	4.973	0.166	260.6	5.521	0.184
38	1302	89	16	94.5	199.9	7.107	0.187	203	7.089	0.187
48	1341	89	16	69	174	8.165	0.170	167.9	8.572	0.179

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 100x100, with *B* = 50 blocks and *Niter* = 100 for PVM and MPI.



	PVM							MPI		
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}	T	S_{par}	E_{par}
1	3998	251	69	3669	3989	1.000	1.000	4251	1.000	1.000
2	2484	248	67	1768	2083	1.915	0.958	2203.7	1.929	0.965
4	1832	248	67	1104.1	1419.1	2.811	0.703	1501.1	2.832	0.708
8	1674	248	67	737.8	1052.8	3.789	0.474	1106.7	3.841	0.480
16	1661	248	67	598.2	913.2	4.368	0.273	951.4	4.468	0.279
20	1628	248	67	483.8	798.8	4.994	0.250	800.3	5.312	0.266
24	1549	248	67	386.9	701.9	5.683	0.237	742.1	5.728	0.239
30	1521	248	67	334.4	649.4	6.143	0.205	680.6	6.246	0.208
38	1475	247	67	142.5	456.5	8.739	0.230	481.5	8.828	0.232
48	1426	247	67	55.9	369.9	10.785	0.225	390.2	10.895	0.227

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 200x200, with *B* = 50 blocks and *Niter* = 100 for PVM and MPI



	PVM							MPI		
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}	T	S_{par}	E_{par}
1	19763	562	92	12024	12678	1.000	1.000	11827	1.000	1.000
2	17048	560	92	5757.5	6409.5	1.978	0.989	5970.21	1.981	0.991
4	13815	562	92	3698.2	4352.2	2.913	0.971	3623.5	3.264	0.816
8	12922	561	92	2693.9	3346.9	3.788	0.947	2973.1	3.978	0.497
16	12774	562	92	2141.6	2795.6	4.535	0.907	2452.7	4.822	0.301
20	12601	562	92	1733.6	2387.6	5.310	0.885	2068.4	5.718	0.286
24	12305	562	92	1576.5	2230.5	5.684	0.812	1956.5	6.045	0.252
30	12305	565	93	1337.9	1995.9	6.352	0.794	1786.3	6.621	0.221
38	12091	564	92	710.8	1366.8	9.276	0.773	1205.2	9.813	0.258
48	124633	564	92	447.6	1103.6	11.488	0.718	992.1	11.921	0.248

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

	PVM							MPI		
N	T_w	T_m	T_{sd}	T_{sc}	T	S_{par}	E_{par}	T	S_{par}	E_{par}
1	2542	112	16	1739	1867	1.000	1.000	1579	1.000	1.000
2	1679	112	12	911.5	1035.5	1.804	0.902	863.3	1.829	0.915
4	1396	114	12	586.1	712.1	2.622	0.874	566.8	2.786	0.697
8	1321	114	12	439.8	565.8	3.300	0.825	427.8	3.691	0.461
16	1295	112	12	350.5	474.5	3.935	0.787	360.1	4.385	0.274
20	1274	112	12	312.4	436.4	4.278	0.713	333.9	4.729	0.236
24	1257	112	12	272.3	396.3	4.711	0.673	314.8	5.016	0.209
30	1238	112	12	259.8	383.8	4.864	0.608	287.7	5.489	0.183
38	1211	112	12	149.4	273.4	6.828	0.569	228.3	6.917	0.182
48	1264	111	12	119.1	242.1	7.712	0.482	187.4	8.424	0.176

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 11 and 12, we listed the total CPU time distribution over various grid sizes and block numbers running with only one processor for PVM and MPI. 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 7 and 8 with Fig. 4 and 5, and Table 11 and 12 with Fig. 6, and 7. 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.

NI x NJ	B = 1	B = 8	B = 16	B = 24	B = 50	B = 100	B = 200
100x100	1534	885	967	998	1119	1739	2149
200x200	5342	4856	4203	3864	3588	4154	4304
400x400	29658	18694	16423	14372	12024	21053	23048

Table 6. 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
100x100	1984	1782	1524	1311	1221	1867	2422
200x200	5119	4628	4383	4107	3897	4588	4966
400x400	18453	16522	14332	13299	12678	21876	23964

Table 7. The total computational time T for 100 iterations as a function of various block numbers for the PVM

NI x NJ	B = 16	B = 24	B = 50	B = 100	B = 200
100x100	1316	1285	1185	1579	2168
200x200	3724	3486	3271	4241	4181
400x400	12694	12047	11827	19876	21196

Table 8. The total computational time T for 100 iterations as a fraction of various block numbers for the MPI

N	B = 20	B = 50	B = 100	B =200
2	6113	6269	6396	3522
4	6113	6327	6518	3664
8	6113	6384	6604	3782
16	6113	6412	6658	3804
30	6113	6412	6661	3812
48	6113	6418	6675	3828

Table 9. The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 100x100 for PVM

N	B = 20	B = 50	B = 100	B =200
2	5885	6126	6294	3188
4	5885	6293	6461	3241
8	5885	6328	6492	3362
16	5885	6394	6528	3451
30	5885	6394	6549	3478
48	5885	6385	6553	3482

Table 10. The number of iteration to achieve a given tolerance of 10^{-3} for a grid of 100x100for MPI

N	B = 20	B = 50	B = 100	B =200
2	6398	6443	6597	3702
4	6398	6451	6628	3824
8	6398	6473	6681	3865
16	6398	6473	6708	4086
30	6398	6473	6701	4132
48	6398	6473	6715	4261

Table 11. The number of iteration to achieve a given tolerance of 10^{-2} for a grid of 200x200 for PVM

N	B = 20	B = 50	B = 100	B =200
2	5996	6187	6242	3284
4	5996	6198	6318	3311
8	5996	6213	6412	3528
16	5996	6213	6412	3528
30	5996	6213	6415	3617
48	5996	6213	6441	3746

Table 12. The number of iteration to achieve a given tolerance of 10^{-2} for a grid of 200x200 for MPI

NIxNJ	B=1	B=8	B=16	B=24	B=50	B=100	B=200
100x100	1689	1428	1316	1285	1185	1579	2168
200x200	4817	4126	3724	3486	3271	4241	4181
400x400	16928	14993	12694	12047	11827	19876	21196

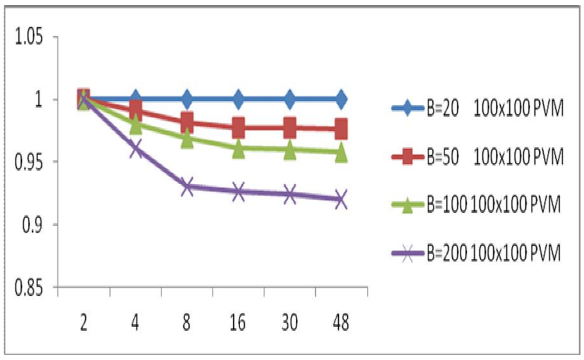


Fig.4. Convergence behavior with domain decomposition for mesh 100x100 for PVM

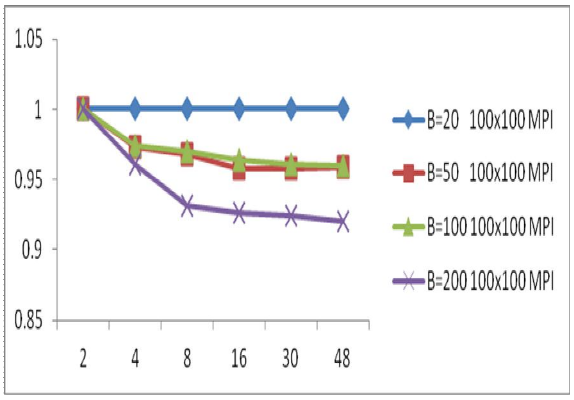


Fig.5. Convergence behavior with domain decomposition for mesh 100x100 for MPI

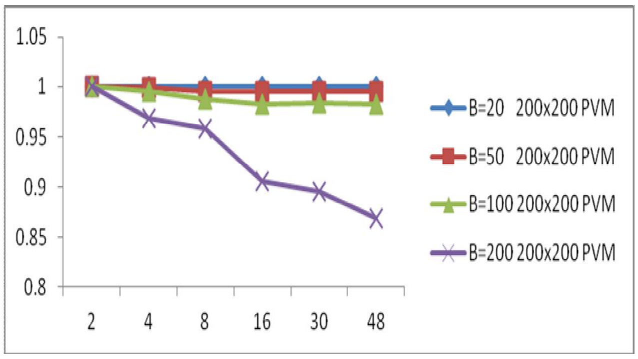


Fig.6. Convergence behavior with domain decomposition for mesh 200x200 for PVM

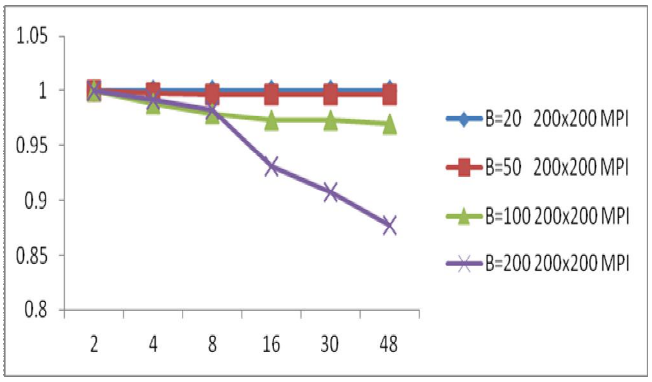


Fig.7. Convergence behavior with domain decomposition for mesh 200x200 for MPI

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 100x100 and 200x200 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. CONCLUSION

We have implemented the solution of the IADE-DY scheme on 2-D Bio-Heat equation on a parallel platform of MPI/PVM cluster via domain decomposition method. We have reported a detailed study on the computational efficiency of a parallel finite difference iterative alternating direction explicit method under a distributed environment with PVM and MPI. 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 Figures 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. 4 – 5). (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.6 – 7). On the basis of the current parallelization strategy, more sophisticated models can be attacked efficiently.

References

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

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

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

[4] J.C Chato, 1998. Fundamentals of Bio-Heat Transfer. Springer-Verlag, Berlin

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

[6] M. Chinmay, 2005. Bio-Heat Transfer Modeling. Infrared Imagine, pp 15 – 31

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

[8] 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

[9] 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.

[10] Z.S Deng, J. Liu, 2002. Analytic Study on Bio-Heat Transfer Problems with Spatial Heating on Skin Surface or Inside Biological Bodies. ASME Journal of Biomechanics Eng., 124 pp 638 – 649

[11] H.S Dou, Phan-Thien. 1997. A Domain Decomposition Implementation of the Simple Method with PVM. Computational Mechanics 20 pp 347 – 358

[12] 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

[13] Eduardo J.H., Yero M.A, Amaral H. (2007). Speedup and Scalability Analysis of Master-Slave Applications on Large Heterogeneous Clusters. Journal of Parallel & Distributed Computing, Vol. 67 Issue 11, 1155 – 1167.

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

[15] I. Foster, J. Geist, W. Groop, E. Lust, 1998. Wide-Area Implementations of the MPI. Parallel Computing 24 pp 1735 – 1749.

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

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

[18] W. Groop, E. Lusk, A. Skjellum, 1999. Using MPI, Portable and Parallel Programming with the Message Passing Interface, 2nd Ed., Cambridge MA, MIT Press

[19] 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

[20] M. Gupta, P. Banerjee, 1992a. Demonstration of Automatic Data Partitioning for Parallelizing Compilers on Multi-Computers. IEEE Trans. Parallel Distributed System, 3, vol. 2, pp 179 – 193

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

[22] J. Z. Jennifer, et al., 2002. A Two Level Finite Difference Scheme for 1-D Penne's Bio-Heat Equation.

[23] J. Liu, L.X Xu, 2001. Estimation of Blood Perfusion Using Phase Shift Temperature Response to Sinusoidal Heating at Skin Surfaces. IEEE Trans. Biomed. Eng., Vol. 46, pp 1037 – 1043

[24] 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.

[25] J. Noye, 1996. Finite Difference Methods for Partial Differential Equations. Numerical Solutions of Partial Differential Equations. North-Hilland Publishing Company.

[26] 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

[27] 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

[28] H.H Pennes, 1948. Analysis of Tissue and Arterial Blood Temperature in the Resting Human Forearm. Journal of Applied Physiol. I, pp 93 – 122

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

[30] R. Rajamony, A. L. Cox, 1997. Performance Debugging Shared Memory Parallel Programs Using

Run-Time Dependence Analysis. Performance Review 25 (1), pp 75 – 87

[31] J. Rantakokko, 2000. Partitioning Strategies for Structuring Multi Blocks Grids. Parallel Computing 26 pp 166 – 1680

[32] 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

[33] B. Rubinsky, et al., 1976. Analysis of a Steufen-Like Problem in a Biological Tissue Around a Cryosurgical Problem. ASME J. Biomech. Eng., vol. 98, pp 514 – 519

[34] Sahimi, M.S., Sundararajan, E., Subramaniam, M. and Hamid, N.A.A. (2001). The D'Yakonov fully explicit variant of the iterative decomposition method, International Journal of Computers and Mathematics with Applications, 42, 1485 – 1496

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

[36] G.D Smith, 1985. Numerical Solution of Partial Differential Equations: Finite Difference Methods 3rd Ed., Oxford University Press New York.

[37] M. Snir, S. Otto et. al., 1998. MPI the Complete Reference, 2nd Ed., Cambridge, MA: MIT Press.

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

[39] 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

AUTHOR

Ewedafe SimonUzezi received the B.Sc. and M.Sc. degrees in Industrial-Mathematics and Mathematics from Delta Stae University and The University of Lagos in 1998 and 2003 respectively. He further obtained his PhD in Numerical Parallel Computing in 2010 from the Universityof 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 a Senior Lecturer in UWI Jamaica in the Department of Computing.