CHAPTER 5

FINITE-DIFFERENCE METHOD FOR 1-D, 2-D AND 3-D TELEGRAPH EQUATIONS

5.1 Introduction

If we recall the first order equations in the general form:

$$\frac{a\partial u}{\partial x}+\frac{b\partial u}{\partial y}=g$$

this equation is similar to solving a first order ODE, and since these equations are rarely encountered in physics, we will not spend any more time on them. The most general second order PDE can be written in the form of Eq. (3.1). Notice that Eq. (3.1) is similar to the Quadratic Equation (QE)

$$ax^2+bxy+cy^2=d.$$

Solutions of this equation fall into three classes, depending on the relations between *a*, *b* and *c*. When $b^2 \prec 4ac$, the equation describes an ellipse; $b^2>4ac$ describes a hyperbola. A special case arises when $b^2=4ac$. In this case the resulting shape is parabolic. A typical parabolic equation is given in Eq. (3.11) – Eq. (3.12). Hence, an example of the hyperbolic equation takes the form of the wave equation

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

Notice that we again have to use a 2-D grid to solve this equation, only now we have to allow for the solution to move both forwards and backwards in time. In order to solve the hyperbolic equation, two sets of initial conditions must be specified. While they can take on many forms, for a traveling wave, the initial position and velocity of the wave pulse are the most frequently stated conditions. From the hyperbolic equation, we have the Telegraph Equation considered in this thesis work.

In this chapter we shall treat the Telegraph Equation in 1-D, 2-D and 3-D dimensions and its stability analysis for 2-D and 3-D Equations. The Telegraph Equation deals 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 in the literature to solve the Telegraph Equation using iterative solution. Some of these iterative schemes are employed in various parallel platforms (Evans & Hassan (2003), Aloy et al., (2007) and Mohanty et al., (1995)). The speed of convergence of iterative scheme is examined for the synchronous communication approaches in parallel environment. Here, we used the finite difference method that provides approximation solutions for the Telegraph Equation such that the derivatives at a point are approximated by difference quotients over a small interval. Each data point in the grid is given an initial value at the beginning of the execution. As time goes by, the value at each grid point is updated according to the difference equation provided based on the time step used in time differentiation and the order used in spatial differentiation. Hence, different dependencies among the grid points are resulted. This regular relationship among the data points is where data parallelization can be captured (Peizong & Kedem, (2002)).

This chapter begins by discussing the approximation of derivative by finite difference method for the 1-D. We then went further to develop the three level implicit method and the IADE-MF scheme for the 1-D case. Section 5.2 discusses the 2-D case Telegraph Equation and alternating schemes on 2-D case with stability analysis for the 2-D case given in section 5.3 with its linear runtime. Section 5.4 introduces the 3-D Equation and section 5.5 gives the ADI scheme on 3-D Equation. Section 5 briefs on the stability analysis for 3-D ADI scheme on 3-D Telegraph Equation with linear runtime.

5.1:1 1-D Telegraph Equation

We seek to discretize the second order 1-D Telegraph Equation in the form of general three level implicit schemes (Evans & Hassan (2003)):

$$\frac{\partial^2 U}{\partial x^2} = LC\frac{\partial^2 U}{\partial t^2} + (RC + GL)\frac{\partial U}{\partial t} + RGU, \qquad 0 \le x \le 1,\ 0 \le t \le 1 \tag{5.1}$$

where (R) is the measure of the opposition to the passage of a steady electrical current, (L) is the property of an electrical circuit where a change in the current flowing through that circuit induces an electromotive force that opposes the change in current, (C) is the ability to hold an electric charge and (G) is the conductance of the path over which leakage current flows.

5.1:2 Three-Level Implicit Schemes on 1-D Telegraph

A stable finite difference scheme in the form of the general three level implicit formula approximating (5.1) at the point $(i\Delta x, j\Delta t)$. Let $\Delta x,$ and Δt be the grid spacing in the x and t directions, where $\Delta x = 1/m$, m is a positive integer. The approximation values $U_{i,j}$ of the solution $U(x,t)$ for Eq.(5.1) are to be computed, where $x_i = i\Delta x,\ i = 0,1,2,\ldots m,\quad t_k = k\Delta t,\quad k = 1,2,\ldots$ the difference scheme is given by:

$$RGU_{i,j} + [RC + GL]\delta_t U_{i,j} + LC\left(\frac{1}{(\Delta t)^2}\right)\delta_t^2 U_{i,j} = \frac{1}{(\Delta x)^2}[\alpha\delta_x^2 U_{i,j+1} + (1-2\alpha)\delta_x^2 U_{i,j} + \alpha\delta_x^2 U_{i,j-1}] \tag{5.2}$$

where α is called the weighting factor and takes the value of $\alpha \ge 1/4$ for stability. The order of scheme is $O\left[(\Delta x)^2 + (\Delta t)^2\right]$ for $\alpha = 1/4$ and $1/2$. On expanding Eq. (5.2) we obtain:

$$
\begin{aligned}
&-\alpha\lambda^2 U_{i-1,j+1} + \left[LC + (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right] U_{i,j+1} - \alpha\lambda^2 U_{i+1,j+1} = \\
&(1-2\alpha)\lambda^2 U_{i-1,j} + 2(LC - (1-2\alpha)\lambda^2 - RG_{i,j}(\Delta t)^2 + (1-2\alpha)\lambda^2 U_{i+1,j} \qquad (5.3)\\
&+\alpha\lambda^2 U_{i-1,j-1} - \left[LC + (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right] U_{i,j-1} + \alpha\lambda^2 U_{i+1,j-1};\ i = 1,2,\cdots,m
\end{aligned}
$$

which gives a tridiagonal system of equations that can be displayed by the matrix (c,a,b) and the right hand side $(RHS)\,f$ where

$$
c = -\alpha\lambda^2,\ a = \left[LC + (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right],\ b = -\alpha\lambda^2,\ \lambda = \frac{\Delta t}{\Delta x},
$$

$$
\begin{aligned}
f_1 &= \left[2(LC - (1-2\alpha)\lambda^2 - RG(\Delta t)^2) U_{i,j} + (1-2\alpha)\lambda^2 U_{2,j} \right] - \\
&\left[LC - (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right] U_{1,j-1} + \alpha\lambda^2 U_{2,j-1} + \alpha\lambda^2 \left[U_{0,j+1} + U_{0,j-1} \right] \\
&+ (1-2\alpha)\lambda^2 U_{0,j};
\end{aligned}
$$

$$
\begin{aligned}
f_i &= \left[(1-2\alpha)\lambda^2 U_{i-1,j} + 2(LC - (1-2\alpha)\lambda^2 - RG(\Delta t)^2) U_{i,j} \right] + \\
&(1-2\alpha)\lambda^2 U_{i+1,j} + \alpha\lambda^2 U_{i-1,j-1} - \left[LC - (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right] U_{i,j-1} \\
&+ \alpha\lambda^2 U_{i+1,j-1},\ i = 2, 3, \cdots, m-1
\end{aligned}
$$

and

$$
\begin{aligned}
f_m &= \left[(1-2\alpha)\lambda^2 U_{m-1,j} + 2(LC - (1-2\alpha)\lambda^2 - RG(\Delta t)^2) U_{m,j} \right] \\
&+ \alpha\lambda^2 U_{m-1,j-1} - \left[LC - (RC+GL)\frac{\Delta t}{2} + 2\alpha\lambda^2 \right] U_{m,j-1} \\
&+ \alpha\lambda^2 \left[U_{m+1,j+1} + U_{m+1,j-1} \right] + (1-2\alpha)\lambda^2 U_{m+1,j}
\end{aligned}
$$

the U values on the first time level are given by the initial condition. Values on the second time level are obtained from applying the forward difference approximate of the first order, at $t = 0$.

$$
\frac{\partial U(x_i,0)}{\partial t} = \frac{U_{i,1} - U_{i,0}}{\Delta t} = g_i
$$

giving $U_{i,1} = U_{i,0} + \Delta t.g_i$. Solutions on the third and subsequent time levels are generated by applying the LU algorithm.

5.1:3 Formulation of the IADE Scheme (Mitchell-Fairweather Variant)

With the Mitchell-Fairweather variant (Sahimi et al., (2001)) accuracy can be improved. The matrices derived from the discretization of Eq. (5.3) resulting to a matrix form is tridiagonal. Hence, at each of the $(k+1/2)$ and $(k+1)$ time levels, the matrix can be decomposed into $G_1 + G_2 - \frac{1}{6}G_1G_2$, where G_1 and G_2 are lower and upper bidiagonal matrices given respectively by:

$$G_1 = [l_i, 1], \quad and \quad G_2 = [e_i, u_i],$$

where

$$e_1 = \frac{6}{5}(a-1)$$
$$u_i = \frac{6}{5}b, \qquad l_i = 6c/(6-e_i); \quad e_i \neq 6$$
$$e_{i+1} = \frac{6}{5}(a + \frac{1}{6}l_iu_i - 1)$$

for $i = 1, 2, \cdots, m-1$. Hence, by taking p as an iteration index and for a fixed acceleration parameter $r > 0$, the IADE scheme is therefore executed at each of the intermediate levels by effecting the following computations:

i) at level (p+1/2)

$$u_i^{(p+1/2)} = (-l_{i-1}u_{i-1}^{(p+1/2)} + s_iu_i^{(p)} + w_iu_{i+1}^{(p)} + f_i)/d \quad i = 1,2,\cdots m,$$

where

$$d = 1 + r$$
$$l_o = w_m = 0$$
$$s_i = r - ge_i, \qquad i = 1, 2, \ldots, m$$
$$w_i = -gu_i, \qquad i = 1, 2, \ldots, m-1$$

$$g = (6+r)/6$$

ii) at level (p+1)

$$u_{m+1-i}^{(p+1)} = (v_{m-1}u_{m-i}^{(p+1/2)} + su_{m+1-i}^{(p+1/2)} + gf_{m+1-i} - u_{m+1-i}u_{m+2-i}^{(p+1)}) / d_{m+1-i}$$

where

$$d_i = r + e_i$$
$$v_o = u_m = 0$$
$$s = r - g,$$
$$v = -gl_i.$$

The IADE algorithm is completed explicitly by using the required equations at levels $(p+1/2)$ and $(p+1)$ in alternate sweeps along the points in the interval $(0,1)$ until convergence is reached.

5.2 2-D Telegraph Equation

Mohanty et al., (1995) and Aloy et al., (2007) have worked on this area extensively (sequentially). We consider the second order Telegraph Equation:

$$\frac{\partial^2 v}{\partial t^2} + a\frac{\partial v}{\partial t} = b\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \quad 0 \le x \le 1,\ 0 \le y \le 1,\ t > 0 \tag{5.4}$$

with initial condition

$$v(x, y, 0) = f(x, y)$$

and boundary conditions

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

where $a = RC + GL$, let $\Delta x, \Delta y \, and \, \Delta t$ be the grid spacing in the x, y and t directions, while $\Delta x = 1/m,\ \Delta y = 1/n,\ m$ and n are the positive integers. The approximation values $v_{i,j,k}$ of the solution $v(x, y, t)$ for the problem Eq. (5.4) and their conditions are to be computed at the grid points (x_i, y_j, t_k) where

$$x_i = i\Delta x,\ i = 0,1,2,...,m,\ y_j = j\Delta j,\ j = 0,1,...n;\ b = 1/LC,\ t_k = k\Delta t,\ k = 1,2,...$$

for simplicity, we take $\Delta x = \Delta y = \Delta d,$ and sometimes denote (x_i, y_j, t_k) by (i, j, k). Among the finite difference method for the numerical solution of the problem Eq. (5.4), 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:

$$\frac{\partial^2 v}{\partial t^2} = \frac{v_{i,j}^{n+1} - 2v_{i,j}^{n} + v_{i,j}^{n-1}}{(\Delta t)^2}, \quad \frac{\partial v}{\partial t} = \frac{v_{i,j}^{n+1} - v_{i,j}^{n-1}}{2\Delta t}$$
$$\frac{\partial^2 v}{\partial x^2} = \frac{v_{i+1,j}^{n+1} - 2v_{i,j}^{n+1} + v_{i-1,j}^{n+1}}{(\Delta x)^2}, \quad \frac{\partial^2 v}{\partial y^2} = \frac{v_{i,j+1}^{n+1} - 2v_{i,j}^{n+1} + v_{i,j-1}^{n+1}}{(\Delta y)^2} \tag{5.5}$$

extending the finite difference scheme on the Eq. (5.5) becomes:

$$\frac{v_{i,j}^{n+1} - 2v_{i,j}^{n} + v_{i,j}^{n-1}}{(\Delta t)^2} + a\frac{v_{i,j}^{n+1} - v_{i,j}^{n-1}}{2\Delta t} - b\left\{\frac{v_{i+1,j}^{n+1} - 2v_{i,j}^{n+1} + v_{i-1,j}^{n+1}}{(\Delta x)^2} + \frac{v_{i,j+1}^{n+1} - 2v_{i,j}^{n+1} + v_{i,j-1}^{n+1}}{(\Delta y)^2}\right\} = 0 \tag{5.6}$$

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.

5.2:1 ADI Scheme

In this section, we will derive the 2-D ADI scheme of the simple implicit Finite Difference Time Domain (FDTD) method by using a general ADI procedure (Peaceman & Rachford, 1955) applied on Eq. (5.4) can be rewritten as:

$$\left(I + \sum_{m=1}^{2} A_m\right) v_{i,j}^{n+1} - 2C_o v_{i,j}^{n} + C_1 v_{i,j}^{n-1} = 0 \tag{5.7}$$

where the operators of $I, A_m s,$ and the constants of C_o, C_1 are defined as:

$$\boldsymbol{I}\, v_{i,j}^{n} \equiv v_{i,j}^{n} \tag{5.8}$$

$$A_1 v_{i,j}^{n} \equiv -\rho_x \left(v_{i+1,j}^{n} - 2v_{i,j}^{n} + v_{i-1,j}^{n}\right) \tag{5.9}$$

$$A_2 v_{i,j}^n \equiv -\rho_y \left(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n \right) \tag{5.10}$$

$$C_o \equiv \frac{1}{(\Delta t)^2} \Bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.11}$$

$$C_1 \equiv \left(\frac{1}{(\Delta t)^2} - \frac{a}{2\Delta t} \right) \Bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.12}$$

the constant of ρ_x, and ρ_y are:

$$\rho_x = \frac{b}{(\Delta x)^2} \Bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.13}$$

$$\rho_y = \frac{b}{(\Delta y)^2} \Bigg/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.14}$$

sub-iteration 1 is given by:

$$\begin{aligned} -\rho_x v_{i+1,j}^{n+1(1)} + (1+2\rho_x) v_{i,j}^{n+1(1)} - \rho_x v_{i-1,j}^{n+1(1)} &= -(A_2) v_{i,j}^{n+1(*)} \\ &\quad + (2C_o v_{i,j}^n - C_1 v_{i,j}^{n-1}) \quad \forall i, j \end{aligned} \tag{5.15}$$

let $a = 1 + 2\rho_x$, $b = c = -\rho_x$. For various values of i and j, Eq. (5.15) can be written in a more compact matrix form at the $(k + 1/2)$ time level as:

$$A v_j^{(k+1/2)} = f_k, \quad j = 1,2,\ldots n. \tag{5.16}$$

where $v = (v_{1,j}, v_{2,j}, \ldots, v_{m,j})^T$, $f = (f_{1,j}, f_{2,j}, \ldots f_{m,j})^T$

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

$$-\rho_y v_{i,j+1}^{n+1(2)} + (1+2\rho_y) v_{i,j}^{n+1(2)} - \rho_y v_{i,j-1}^{n+1(2)} = v_{i,j}^{n+1(1)} + A_2 v_{i,j}^{n+1(*)}. \quad \forall\, i, j \tag{5.17}$$

let $a = 1 + 2\rho_y$, $b = c = -\rho_y$. For various values of i and j, Eq. (5.17) can be written in a more compact matrix form as:

$$B v_i^{(k+1)} = g_{k+1/2}, \qquad i = 1,2,\ldots m \tag{5.18}$$

where

$$v_i^{(k+1)} = (v_{i,1}, v_{i,2}, \ldots, v_{i,n})^T, \; g = (g_{i,1}, g_{i,2}, \ldots, g_{i,n})^T$$

and set

$$v_{i,j}^{n+1(*)} = 2v_{i,j}^{n} - v_{i,j}^{n-1} \tag{5.19}$$

this is a prediction of $v_{i,j}^{n+1}$ by the extrapolation method. Splitting by using an ADI procedure as in (Sahimi et. al., 2006), we get a set of recursion relations as follows:

$$(I + A_1)v_{i,j}^{n+1(1)} = -(A_2)v_{i,j}^{n+1(*)} + (2C_o v_{i,j}^{n} - C_1 v_{i,j}^{n-1}) \tag{5.20}$$

$$(I + A_2)v_{i,j}^{n+1(2)} = v_{i,j}^{n+1(1)} + A_2 v_{i,j}^{n+1(*)} \tag{5.21}$$

where $v_{i,j}^{n+1(1)}$ is the intermediate solution and the desired solution is $v_{i,j}^{n+1} = v_{i,j}^{n+1(2)}$. However, expanding A_1 and A_2 on the left side of Eq. (5.20) and Eq. (5.21), we get the 2-D ADI algorithm.

The 2-D ADI Algorithm

Input = $v_{i,j}^{n}, v_{i,j}^{n-1}\ \forall i, j$ **Output =** $v_{i,j}^{n+1}\ \forall i, j$

Begin

Sub-Iteration 1:

$$-\rho_x v_{i+1,j}^{n+1(1)} + (1+2\rho_x)v_{i,j}^{n+1(1)} - \rho_x v_{i-1,j}^{n+1(1)} = -(A_2)v_{i,j}^{n+1(*)} + (2C_o v_{i,j}^{n} - C_1 v_{i,j}^{n-1}) \quad \forall i, j$$

Sub-Iteration 2:

$$-\rho_y v_{i,j+1}^{n+1(2)} + (1+2\rho_y)v_{i,j}^{n+1(2)} - \rho_y v_{i,j-1}^{n+1(2)} = v_{i,j}^{n+1(1)} + A_2 v_{i,j}^{n+1(*)}. \quad \forall\ i, j$$

End

Fig. 5.1: The 2-D ADI Algorithm

5.2:2 IADE-DY Scheme

By applying the IADE-DY of (Sahimi et. al (2001)), the matrices derived from the discretization resulting to A in Eq. (5.16) and B in Eq. (5.18) are respectively tridiagonal of size $(m \times m)$ and $(n \times n)$. Hence, at each of the $(k+1/2)$ and $(k+1)$ time levels, these matrices can be decomposed into $G_1 + G_2 - \frac{1}{6}G_1 G_2$, where $G_1\ and\ G_2$ are lower and upper bidiagonal matrices given respectively by:

$$G_1 = [l_i, 1], \qquad and \quad G_2 = [e_i, u_i], \tag{5.22}$$

where

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

hence, by taking p as an iteration index, and for a fixed iteration parameter $r > 0$, where r is given by (Evans (2003)) as $r = \sqrt{u.v}$, where u and v in the below expressions are the minimum and maximum eigenvalues of the submatrices of G_1 and G_2 for the two-stage IADE-DY scheme of the form:

$$\begin{aligned}(rI + G_1)u^{(p+1/2)} &= (rI - gG_1)(rI - gG_2)u^{(p)} + hf \quad \textit{and}\\ (rI + G_2)u^{(p+1)} &= u^{(p+1/2)}\end{aligned} \tag{5.23}$$

can be applied on each of the sweeps Eq. (5.20) and Eq. (5.21). Based on the fractional splitting strategy of D'Yakonov, the iterative procedure is accurate and is found to be stable and convergent. By carrying out the relevant multiplications in Eq. (5.23), the following equations for computation at each of the intermediate levels are obtained:

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

$$\begin{aligned}u_1^{(p+1/2)} &= \frac{1}{\hat{d}}(s_1 \hat{s} u_1^{(p)} + w_1 \hat{s} u_2^{(p)} + hf_1)\\ u_i^{(p+1/2)} &= \frac{1}{\hat{d}}(-l_{i-1}u_{i-1}^{(p+1/2)} + v_{i-1}s_{i-1}u_{i-1}^{(p)} + (v_{i-1}w_{i-1} + s_i \hat{s})u_i^{(p)} + w_i \hat{s} u_{i+1}^{(p)} + hf_i),\\ &\quad i = 2,3,\ldots,m-1\\ u_m^{(p+1/2)} &= \frac{1}{\hat{d}}(-l_{m-1}u_{m-1}^{(p+1/2)} + v_{m-1}s_{m-1}u_{m-1}^{(p)} + (v_{m-1}w_{m-1} + s_m \hat{s})u_m^{(p)} + hf_m)\end{aligned} \tag{5.24}$$

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,

$$u_m^{(p+1)} = \frac{u_m^{(p+1/2)}}{d_m},$$
$$u_i^{(p+1)} = \frac{1}{d_i}(u_i^{(p+1/2)} - \hat{u}_i\, u_{i+1}^{(p+1)}),\ \textit{where}\ d_i = r + e_i,\ i = m-1, m-2, \ldots, 2, 1 \tag{5.25}$$

the two-stage iterative procedure in the IADE-DY algorithm corresponds to sweeping through the mesh involving at each iterates the solution of an explicit equation. This is continued until convergence is reached, that is when the convergence requirement $\left\| u^{(p+1)} - u^{(p)} \right\|_\infty \le \varepsilon$ is met, where ε is the convergence criterion.

5.2:3 MF-DS Scheme

According to (Mitchell & Fairweather (1964)) the numerical representative of Eq. (5.20) and Eq. (5.21) is as follows:

$$\left(1 - \frac{1}{2}\left(\rho_x - \frac{1}{6}\right)A_1\right)v_{i,j}^{n+1(1)} = \left(1 + \frac{1}{2}\left(\rho_y + \frac{1}{6}\right)A_2\right)v_{i,j}^{n+1(*)} + \left(2C_o v_{i,j}^{n} - C_1 v_{i,j}^{n-1}\right) \tag{5.26}$$

$$\left(1 - \frac{1}{2}\left(\rho_y - \frac{1}{6}\right)A_2\right)v_{i,j}^{n+1(2)} = \left(1 + \frac{1}{2}\left(\rho_x + \frac{1}{6}\right)A_1\right)v_{i,j}^{n+1(1)} + A_2 v_{i,j}^{n+1(*)} \tag{5.27}$$

the horizontal sweep Eq. (5.26) and the vertical sweep Eq. (5.27) 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, Eq. (5.26) can be written in a more compact matrix form at the $(k+1/2)$ time level as in Eq. (5.16). Similarly, let $a = \frac{5}{6} + \rho_y,\ b = c = \frac{1}{12} - \frac{\rho_y}{2}$ for various values of i and j, Eq. (5.27) can be written in a more compact matrix form at the $(k+1)$ time level as in Eq. (5.18). By defining $a = \frac{5}{6} + \rho_y,\ b = c = \frac{1}{12} - \frac{\rho_y}{2}$ the resulting tridiagonal system of equations are solved using similar iterative procedure as in the DS-PR, that is, the two-stage IADE-DY algorithm.

5.3 Introduction to 3-D Telegraph Equation

In this section, we present the implementation of an unconditionally stable three-dimensional (3-D) finite difference method on 3-D Telegraph Equation using 3-D ADI scheme. The principle of the alternating direction implicit (ADI) technique that has been used in formulating an unconditionally stable two-dimensional is applied. Unlike the conventional ADI algorithms, the alternation is performed in respect to coordinate direction. In reference to (Mohanty et. al. (1995)), the numerical solution of 3-D Alternating-Direction-Implicit (3-D ADI) method is obtained. The method is developed from the judicious splitting of the implicit equations derived from the finite difference discretization of the partial differential equations of the Telegraph Equation in 3-D. The method employs a splitting strategy which is applied alternately at each half time step. We examine the application and numerical performance of the 3-D ADI scheme.

We consider the second order Telegraph Equation:

$$\frac{\partial^2 v}{\partial t^2} + a\frac{\partial v}{\partial t} - \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) = 0 \tag{5.28}$$

where $a = RC + GL$, let $\Delta x, \Delta y \, and \, \Delta z$ be the grid spacing in the x, y, z and t directions, where $\Delta x = \Delta y = \Delta z = 1/m$, m is a positive integer. Hence, we can solve Eq. (5.28) by extending the 1-D simple implicit finite difference method by (Smith, 1978) to the above 3-D Telegraph Equation, Eq. (5.28) becomes:

$$\frac{v_{i,j,k}^{n+1} - 2v_{i,j,k}^{n} + v_{i,j,k}^{n-1}}{(\Delta t)^2} + a\frac{v_{i,j,k}^{n+1} - v_{i,j,k}^{n-1}}{2\Delta t} - \left\{\begin{array}{l} \dfrac{v_{i+1,j,k}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i-1,j,k}^{n+1}}{(\Delta x)^2} \\ + \dfrac{v_{i,j+1,k}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i,j-1,k}^{n+1}}{(\Delta y)^2} \\ + \dfrac{v_{i,j,k+1}^{n+1} - 2v_{i,j,k}^{n+1} + v_{i,j,k-1}^{n+1}}{(\Delta z)^2} \end{array}\right\} = 0 \tag{5.29}$$

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

5.4 ADI Scheme on 3-D Telegraph Equation (Mohanty, (2009))

In this section, we will derive the 3-D ADI method of the simple implicit finite difference method by using (Peaceman & Rachford, (1955)) extended to Eq. (5.28). The ADI method is a well-known method for solving PDE. The main feature of ADI is to sweep directions alternatively. In contrast to the standard finite-difference formulation with only one iteration to advance from the nth to $(n+1)th$ time step, the formulation of the ADI method requires multilevel intermediate steps to advance from the nth to $(n+1)th$ time step. Eq. (5.29) can be rewritten as:

$$\left(I+\sum_{m=1}^{3}A_m\right)v_{i,j,k}^{n+1}-2C_o v_{i,j,k}^{n}+C_1 v_{i,j,k}^{n-1}=0 \tag{5.30}$$

where the operators of $I, Ams,$ and the constants of C_o, C_1 are define as:

$$\boldsymbol{I}\, v_{i,j,k}^{n} \equiv v_{i,j,k}^{n} \tag{5.31}$$

$$A_1 v_{i,j,k}^{n} \equiv -\rho_x\left(v_{i+1,j,k}^{n}-2v_{i,j,k}^{n}+v_{i-1,j,k}^{n}\right) \tag{5.32}$$

$$A_2 v_{i,j,k}^{n} \equiv -\rho_y\left(v_{i,j+1,k}^{n}-2v_{i,j,k}^{n}+v_{i,j-1,k}^{n}\right) \tag{5.33}$$

$$A_3 v_{i,j,k}^{n} \equiv -\rho_z\left(v_{i,j,k+1}^{n}-2v_{i,j,k}^{n}+v_{i,j,k-1}^{n}\right) \tag{5.34}$$

$$C_o \equiv \frac{1}{(\Delta t)^2}\Bigg/\left(\frac{1}{(\Delta t)^2}+\frac{a}{2\Delta t}\right) \tag{5.35}$$

$$C_1 \equiv \left(\frac{1}{(\Delta t)^2}-\frac{a}{2\Delta t}\right)\Bigg/\left(\frac{1}{(\Delta t)^2}+\frac{a}{2\Delta t}\right) \tag{5.36}$$

The constant of ρ_x, ρ_y and ρ_z are:

$$\rho_x = \frac{b}{(\Delta x)^2} \Big/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.37}$$

$$\rho_y = \frac{b}{(\Delta y)^2} \Big/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.38}$$

$$\rho_y = \frac{b}{(\Delta y)^2} \Big/ \left(\frac{1}{(\Delta t)^2} + \frac{a}{2\Delta t} \right) \tag{5.39}$$

The 3-D ADI Algorithm

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

Begin

Sub-Iteration 1:

$$-\rho_x v^{n+1(1)}_{i+1,j,k} + (1+2\rho_x) v^{n+1(1)}_{i,j,k} - \rho_x v^{n+1(1)}_{i-1,j} = -(A_2 + A_3) v^{n+1(*)}_{i,j,k} + (2C_o v^{n}_{i,j,k} - C_1 v^{n-1}_{i,j,k}) \quad \forall i, j, k$$

Sub-Iteration 2:

$$-\rho_y v^{n+1(2)}_{i,j+1,k} + (1+2\rho_y) v^{n+1(2)}_{i,j,k} - \rho_y v^{n+1(2)}_{i,j-1,k} = v^{n+1(1)}_{i,j,k} + A_2 v^{n+1(*)}_{i,j,k}. \quad \forall i, j, k$$

Sub-Iteration 3:

$$-\rho_z v^{n+1(3)}_{i,j,k+1} + (1+2\rho_z) v^{n+1(3)}_{i,j,k} - \rho_z v^{n+1(3)}_{i,j,k-1} = v^{n+1(2)}_{i,j,k} + A_3 v^{n+1(*)}_{i,j,k}. \quad \forall i, j, k$$

End

Fig. 5.2: The 3-D ADI Algorithm

and set

$$v^{n+1(*)}_{i,j,k} = 2v^{n}_{i,j,k} - v^{n-1}_{i,j,k} \tag{5.40}$$

which is a prediction of $v^{n+1}_{i,j,k}$ by the extrapolation method.

Then splitting Eq. (5.30) by using an ADI procedure, we get a set of recursion relations as follows:

$$(I + A_1) v^{n+1(1)}_{i,j,k} = -(A_2 + A_3) v^{n+1(*)}_{i,j,k} + (2C_0 v^{n}_{i,j,k} - C_1 v^{n-1}_{i,j,k}) \tag{5.41}$$

$$(I + A_2) v^{n+1(2)}_{i,j,k} = v^{n+1(1)}_{i,j,k} + A_2 v^{n+1(*)}_{i,j,k} \tag{5.42}$$

$$(I + A_3)v_{i,j,k}^{n+1(3)} = v_{i,j,k}^{n+1(2)} + A_3 v_{i,j,k}^{n+1(*)} \tag{5.43}$$

where $v_{i,j,k}^{n+1(1)}, v_{i,j,k}^{n+1(2)}$ are the intermediate solutions and the desired solution is $v_{i,j,k}^{n+1} = v_{i,j,k}^{n+1(3)}$. Finally, expanding A_1, A_2 and A_3 on the left side of Eq. (5.41) and Eq. (5.43), we get the 3-D ADI algorithm as in Fig. 5.2.

5.5 Parallel Performance Analysis of the Algorithms

The following authors gave good illustrations on measures of performance for parallel programs; Anderson et al., (1998); Burns (1988), Flynn (1972), Foster (1995), Barry (2003), Ananth et al., (2003), Amdahl (1967), Sun (1991), Akl (1997) and Pacheco (1997). Parallel computing has always had its skeptics. First, Grosch's Law had to be overturned by making computers four times as fast in order to sell them for two times as much. Grosch's Law was repeated by large scale integration of electronics and then Von-Neumann's bottleneck had to be dealt with. A Von-Neumann computer is limited in performance by the narrow connection between the processor and its memory. But Amdahl's Law (Amdahl, (1967)) predicted very limited improvement in performance because the speed of a computer was limited by its slowest (sequential) part. Regardless of the number of processors, the problem could never be solved faster than naturally occurring serial part would permit. Amdahl's Law was shown to be invalid in certain very interesting cases – cases where the problem size could be increased and the regularity of the problem could be used to feed as many parallel processors as the problem needed. Thus, large matrix calculation could grow even larger without sacrificing speed, if more and more processor were "thrown at the problem". The Gustafson – Basis Law stimulated great interest once again in parallelism. This brings up the issue of performance measurement. Parallel computing redefines traditional measures such that Million Instructions Per Second (MIPS) and Million Floating Point

Operations per Seconds (MFLOPS). A new measure of performance is needed to relate parallel computing to performance. Recognizing the performance limitation of using commodity interfaces in workstation clusters has lead several researchers to design High Performance network interface cards (NICS), examples includes Blumrich et al., (1995), Bodon et al., (1995), Gillett & Kaufmann (1997), Burns (1988). Martin et al., (1997) made a detailed study of the effects of communication latency, overheads, and bandwidth in cluster architecture. They improved the communication performance of the communication system rather than invest in doubling the machine performance. The most often quoted measure of parallel performance is the speed-up curve. This is computed by dividing the time to compute a solution to a certain problem using one processor by the solution time using N processors in parallel. According to Barry & Michael (2003) and Lewis (2001), measures of performance were discussed explicitly.

5.5:1 Parallel Strategies

We essentially focus on the parallelization of the methods stated earlier in the abstract. Striped partitioning and Cell partitioning are two main decomposition approaches for solving on parallel computers. For ADI method striped partitioning turns out to be both numerically uneconomical for implementation and non-scalable. An individual tri-diagonal system of equations is most easy to solve if all the data is residing in one processing element (PE). In the (*y*) striped partition approach for each subset in *y*-direction, this can be achieved by assigning n/p equation systems to each PE. But in the *x*-direction the *(y)* stripped partition implies that for each tri-diagonal system of equations each holds only n/p equations. Hence, this demand a transpose of $n \times n$ grid information which effectively turns out to be a costly communication process i.e. $O(n^2/p)$ which is of the order of the computation and hence not desirable.

In the cell partitioning approach we do away with the need for such global communication by suitably re-grouping the PEs under column communicators to row communicators and thereby requiring only local communications. The details of the parallel implementation under cell partitioning strategy are as follows: Assume that there are k^2 processors arranged in $(k \times k)$ grid. The domain in which solutions are sought is treated as $n \times n$ grid. Say, $k = 4$ for the parallel execution of the ADI method k row communicators and k column communicators are defined. As in Fig. 5.3, during the first sweep the processors are grouped under row communicators and in the second sweep they are re-grouped under the column communicators. In each of the two steps, the set of tri-diagonal is also divided into k parts. Each of these k sub-systems of equations is further divided into k-sub-matrices and is assigned to different processors. The distribution of the matrix systems among the various processors under row and column communicators during the two steps is shown in Fig. 5.4.

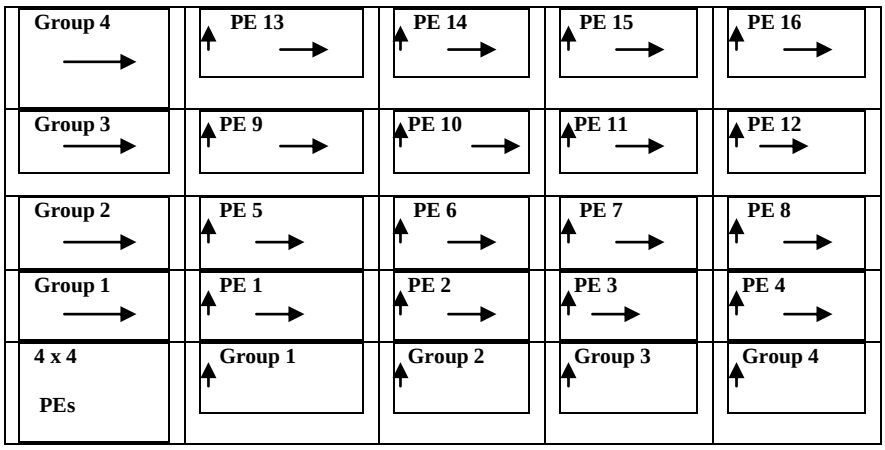

Fig.5.3. Grouping of the processors under the row and column communicators

(Row: ; column:).

Group 4 (row)	(NJ/4) no. (NI/4 x NI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices
Group 3 (row)	(NJ/4) no. (NI/4 x NI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices
Group 2 (row)	(NJ/4) no. (NI/4 x NI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices
Group 1 (row)	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices	(NJ/4) no. (NI/4xNI/4) Matrices
	Group 1 (column)	Group 2 (column)	Group 3 (column)	Group 4 (column)

Fig.5.4 Distribution of the matrix systems among the $(k \times k)$ processors under the row and column communication groups.

As has been stated above, the effective parallel implementation of the algorithms considered in this work should lead to a substantial increase in the computational count per data exchange, major reduction in synchronization frequency and subsequent decrease in communication sessions. A typical parallel implementation involves the assignment of a block of grids to each task to a surface so that each task only communicates with its limited nearest neighbors. Only the top, bottom, left and right surfaces of the block need to be exchanged between neighboring tasks. Hence, it is important to maintain load balancing in the distribution of *n* grids to tasks $P_1, P_2 \ldots, P_p$. The data decomposition of the algorithms is run simultaneously at every time level, where each task is allocated n/p grids. It proceeds for every task at each time level until the local error and the approximate solution are computed at the last time level.

These tasks then send the local errors to the master, which in turn processes the global error.

However, on the MPI and PVM parallel implementation of the various schemes are based on one master and many tasks. The master program is responsible in constructing the n grid sizes, computing the initial values, partitioning the grid into blocks of surfaces, assigning these blocks to the p task modules, distributing the task to different processors and receiving local errors from the tasks. Each block that is assigned to a task module is composed of n/p blocks. A task process starts computations after it receives a work assignment. A task module $f\,(f < p)$ performs the algorithm iterations on the grid points of the assigned block which is composed of surfaces with indices below:

$$SF_f\,(start) = \frac{n(f-1)}{p} \text{ and } SF_f\,(end) = \frac{nf}{p} - 1,$$

where SF refers to the surface. The task f will transmit to its upper neighbor $(task\ f-1)$, $SF_f\,(start)$ and receive from $SF_f\,(start) - 1 = SF_{f-1}\,(end)$ and to it lower neighbors $(task\ f+1)$, $SF_f\,(end)$ and receive from it $SF_f\,(end) + 1 = SF_{f+1}\,(start)$.

Since multiple copies of the same task code run simultaneously, the tasks will exchange data with their neighbors at different times. At this point, the barrier function is called by the MPI and PVM library routines for synchronization. The tasks will repeat the above procedure, until the local convergence criterion is met. The criterion requirement computed in the task f is as follows,

$$h[i][j] = \max\left\{\left|u_{i,j}^{(p+1)} - u_{i,j}^{(p)}\right|,\ (i,j) \in f\right\},$$

The tasks will return all its local errors to the master module. After receiving the locally converged blocks from the tasks, the master module checks whether the global convergence is satisfied.

$$h[i][j] \le \varepsilon,\ \forall\ i, j \in [0, n],$$

where $h[i][j] = \max\left\{\left|u_{i,j}^{(p+1)} - u_{i,j}^{(p)}\right|, (i, j) \in f\right\}$.

This procedure is repeated and the system terminates if a global convergence is reached. Otherwise, the master repartitions the blocks and reassigns them to the tasks. More of this parallel implementation can be found in (Barry & Michael (2003), Quinn (2001) and Sahimi et al., (2001)).

5.5:2 Parallel Computation of the Algorithms

To correlate the communication activity with computation, we counted events between significant PVM/MPI call sites. 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 exchanged between the neighboring processors using the libraries and computation for the next phase proceeds in parallel in each processor, see (Bin Jia, (2009)) . 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:

$$T_{2p}(n) = T_{comp}(n) + T_{send}(n) + T_{wait}(n) + T_{recv}(n) \tag{5.44}$$

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:

$$\begin{aligned} T_{best}(n) &= T_{postcompute}(n) + T_{send}(n) + T_{precompute}(n) + T_{recv}(n) \\ &= T_{comp}(n) + T_{send}(n) + T_{recv}(n) \end{aligned} \tag{5.45}$$

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:

$$T_{best} = T_{send}(n) + T_{wait}(n) + T_{recv}(n) + T_{postcomput}(n) \tag{5.46}$$

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.

5.6 Parallel Performance Measurement of the Algorithms

In this section, we measure the effectiveness and performance between the numerical methods under considerations. More on parallel performance can be found in Rocco et al., (2005).

5.6:1 Speedup, Efficiency and Effectiveness

In this work, speedup, efficiency and effectiveness are terms necessary to explain the performance of our numerical algorithms on distributed platforms. 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 (Rajamony & Cox (1997) and Womble (1990)). Speedup is the ratio of the serial time to the parallel version run on N processors. Efficiency is the ability to judge how effective the parallel algorithm is expressed as the ratio of the speedup to N processors. The concept of speedup has yet to find a widely accepted definition. In traditional parallel systems it is widely define as:

$$S(n) = {T(s)}/{T(n)}, \qquad E(n) = {S(n)}/{n} \tag{5.47}$$

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 together with their advantages and disadvantages is presented by Sahni (1996) and observed that speedup is normally defined as the execution time of the best sequential algorithm also known as absolute speedup, therefore implying that the sequential and parallel might be different. A different approach known as relative speedup, considers the parallel and sequential algorithm to be the same. While the absolute speedup calculates the performance gain for a particular problem using any algorithms, relative speedup focuses on the performance gain for a specific algorithm that solves the problem. The total efficiency according to (Dou & Phan-Thien (1997)) is usually decomposed into the following equations

$$E(n) = E_{num}(n)E_{par}(n)E_{load}(n), \tag{5.48}$$

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

$$S_{par}(n) = {T(1)}/{T(n)}, \quad E_{par}(n) = {S_{par}(n)}/{n} \tag{5.49}$$

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

$$T(n) = T_m(n) + T_{sd}(n) + T_{sc}(n) \tag{5.50}$$

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

$$T_m(n) = T_m(1), \quad T_{sd}(n) = T_{sd}(1), \quad T_{sc}(n) = T_{sc}(1)/n. \tag{5.51}$$

Therefore, the speedup can be written as:

$$S_{par}(n) = \frac{T(1)}{T(n)} = \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1) + T_{sc}(1)/n} < \frac{T_{ser}(1) + T_{sc}(1)}{T_{ser}(1)} \tag{5.52}$$

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

$$Epar(n) = \frac{T(1)}{nT(n)} = \frac{Tser(1) + Tsc(1)}{nTser(1) + Tsc(1)} < \frac{Tser(1) + Tsc(1)}{nTser(1)} \tag{5.53}$$

the parallel efficiency represents the effectiveness of the parallel program running on N processors relative to a single processor. However, it is the total efficiency that is of real significance when comparing the performance of a parallel program to the corresponding serial version. Let $T_s^{No}(1)$ denotes the CPU time of the corresponding serial program to reach a prescribed accuracy with *No* iterations, and $T_{B=B}^{N_1 L}(n)$ 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 can be decomposed as follows:

$$E(n) = \frac{T_s^{N_o}(1)}{n.T_{B=B}^{N_1 L}(n)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)} \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \frac{T_{B=B}^{N_0}(1)}{T_{B=B}^{N_1}(1)} \frac{T_{B=B}^{N_1}(1)}{T_{B=B}^{N_1}(n)} \frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1 L}(n)}, \tag{5.54}$$

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 Eq. (5.51) and Eq. (5.48), we obtain:

$$E_{load}(n) = \frac{T_{B=B}^{N_1}(n)}{T_{B=B}^{N_1 L}}, E_{par}(n) = \frac{T_{B=B}^{N_1}(1)}{n.T_{B=B}^{N_1}(n)},$$
$$Enum(n) = \frac{T_s^{N_o}(1)}{T_{B=B}^{N_1}(1)} = \frac{T_s^{N_o}(1)}{T_{B=1}^{N_o}(1)} \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)} \frac{T_{B=B}^{N_o}(1)}{T_{B=B}^{N_1}(1)}, \quad (5.55)$$

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,

$$E_{num}(n) = E_{dd} \frac{N_o}{N_1}, \quad E_{dd} = \frac{T_{B=1}^{N_o}(1)}{T_{B=B}^{N_o}(1)}. \quad (5.56)$$

We call Eq. (5.56) domain decomposition efficiency (DDE), which includes the increase of CPU time induced by grid overlap at interfaces and the CPU time variation generated by DD techniques.

To obtain a high efficiency, the slave computational time $T_{sc}(1)$ should be significantly larger than the serial time T_{ser}. 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 speedup and efficiency obtained for various grid sizes for the PVM and the MPI implementations are listed in section 6, showing different meshes for different problems. The figures show graphical representation of the lower mesh sizes. In the tables, we also listed the elapsed time for the master task, T_w, the master CPU time, T_m, and the slave data communication time, T_{sd}, all in seconds.

The effectiveness is given by:

$$L_n = S_n / C_n \quad (5.57)$$

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. We can observe from the figures in section 6 that high speedups for the algorithms are obtained from larger problems. This is due to the relatively high communication time for passing data between the master and slaves;

wasteful idle time at a barrier synchronization point before proceeding with the next iteration. Hence, effectiveness is given as

$$L_n = S_n/(nT_n) = E_n/T_n = E_n S_n/T_1 \tag{5.58}$$

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 $L_n T_1 = S_n E_n$.

5.7 General Parallel Implementation of the Schemes

Implementation of the algorithms is straight forward as follows:

- Division of spatial computational domain to desired number of sub-domains, based on load balancing constraint.
- Indication of sweeping direction for each sub-domain. Sweeping direction of each sub-domain must be in opposite direction of its neighbors. For example we use LR(left to right) direction for odd sub-domains and RL (right to left) direction for even sub-domains. This sweeping direction is inverted after each time step.
- Updating start node of each sub-domain and remained nodes. If the start node is located at physical boundaries, the prescribed boundary values is used.

The first aspect of parallelizing a finite difference method by domain decomposition is to divide the grid into parts. This is how the full computational task is divided among the various processors of the parallel machine. Each processor works only on its specific portion of the grid and anytime a processor needs information from another processor a message is passed. For the best parallel performance, one would like to have optimal load balancing and as little communication between processors as possible.

Consider load balancing first. Ideally, one would like each processor to do exactly the same amount of work (suppose we have same processors). That way, each processor is

always working and not sitting idle. For a finite difference code, the basic computational element usually is the node. It makes sense to partition the grid such that each processor gets an equal (or nearly equal) number of nodes to work on.

The second criterion is that the amount of communication between processors be made as small as possible. To minimize communication, the program must divide the domain in a way that minimizes the length of the touching faces in the different sub-domains. The number of processors that once given has to communicate with also contributes to additional communication time. This is because each time a processor wants to communicate with a new processor, a latency penalty for starting the new message is incurred. This leads to the full statement of the second criterion for a good grid partitioning. Consideration of system latency in domain decomposition procedure is not a simple task because it is severely a function of network specifications and needs some experiments with network. In this thesis work we neglect this effect for simplicity of domain decomposition (DD).

Therefore the DD procedure is converted to a simple constrained minimization problem that its cost function is the total length of interfaces between sub-domains and its constraint is equal to the number of sub-domains nodes. Therefore at first step we divide the spatial computational domain to $P = P_1 \times P_2$ sub-domains without any overlapping and define two-dimensional Cartesian topology with $P = P_1 \times P_2$ processors. As an example Fig. 5.5 shows a partition square domain with $P_1 = P_2 = 3$. After DD, we need to indicate sweeping directions in x and y directions for each sub-domain. Sweeping directions of each sub-domain must be in opposite directions of its neighbors. For example if (P_x, P_y) are the coordinates of each processor in defined Cartesian topology, we use LR direction for odd value of P_x and RL direction for even value of P_x and in the same manner, we use DT (down to top) direction for odd value of

P_y and TD (top to down) direction for even value of P_y. These sweeping directions are inverted after each time step. Fig. 5.6 shows schematically this procedure for $P_1 = P_2 = 3$.

The next stage is updating starting nodes of each sub-domain with their respective equations. We need two nodal values of neighbor sub-domain; we add two additional columns (or rows) of grids to local mesh of each processor at its interfaces with other processors (see Fig. 5.7). Therefore before performing the differential step in x direction each processor communicate with its left and right neighbors and attains the new values of its left and right additional columns. In the same manner before performing differential step y direction each processor has communication with it's down and up neighbors. Consider the differential step in x direction and suppose the sweeping direction is LR. At start of updating procedure the processor use the equation in the $(n+1)th$ or $(p+1)th$ time level therefore it needs previous values of its two additional columns that are related to its left neighbor processor. This needs one communication stage with its left neighbor processor. When calculation proceeds to right boundary for updating last column of nodes the previous values of one column of right neighbor processor is also needed. This needs also one communication stage that must be completed before updating last column. For achieving better performance we use the overlapping communication for hiding network latency in this communication stage. For example, we use the non-blocking message passing for this communication stage.

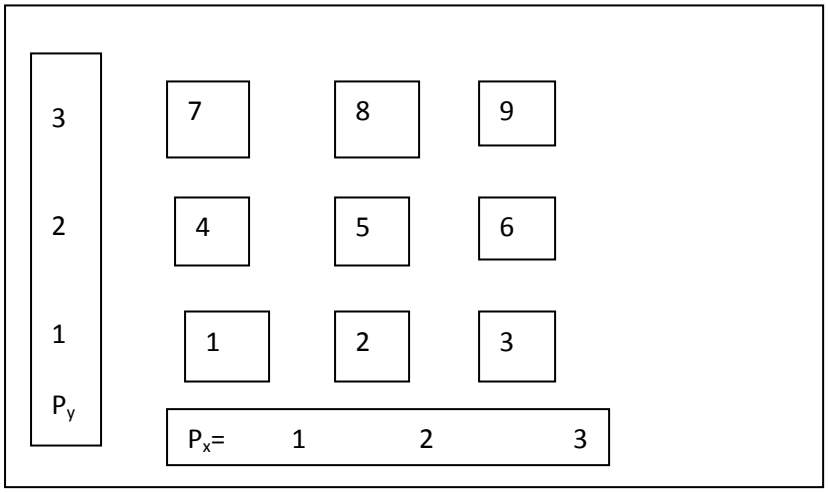

Fig. 5.5. Schematic of 3 x 3 array of processors in two-dimension Cartesian topology.

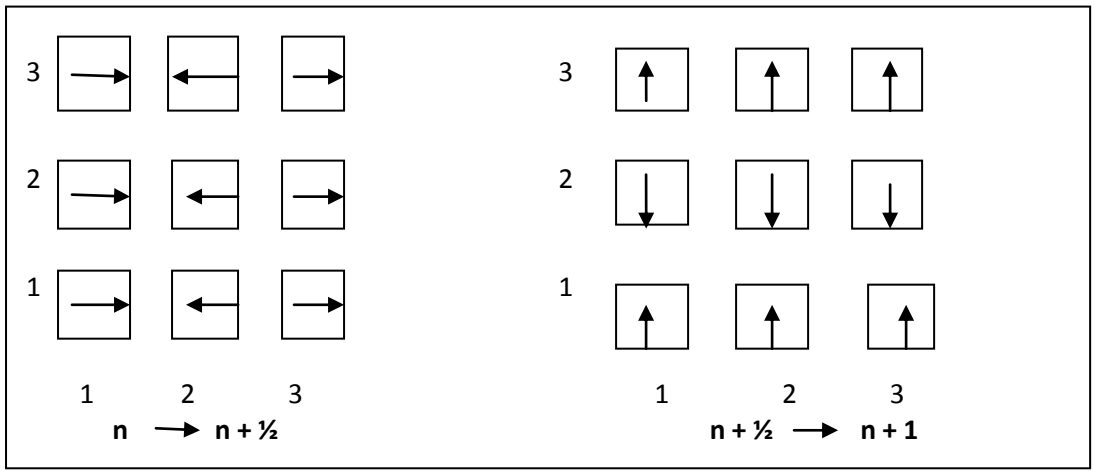

Fig. 5.6. Schematic of sweeping directions of 3 x 3 array of processors two time steps

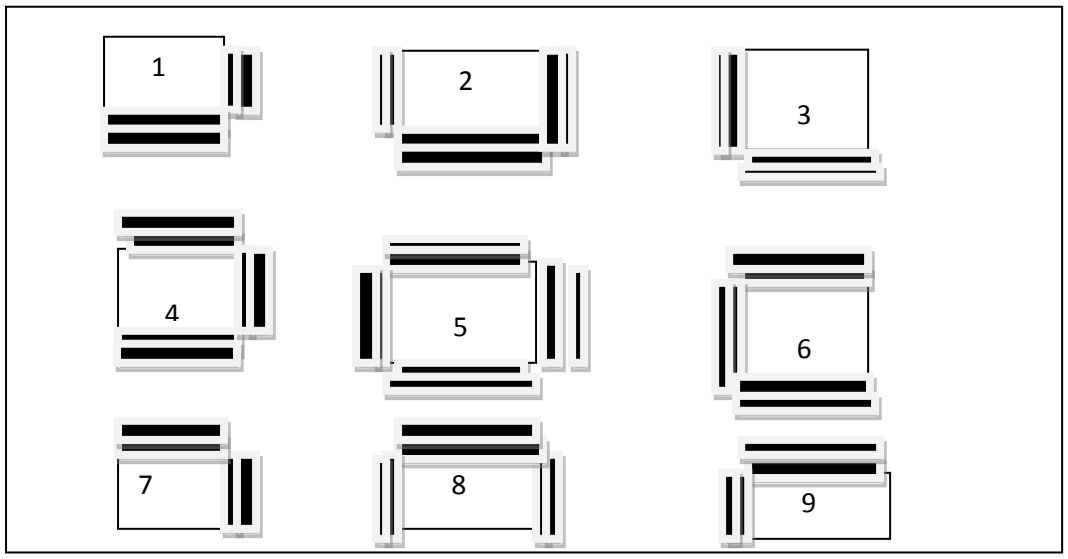

Fig. 5.7 Schematic of 3x3 array processors with local grid and additional columns

5.8 Other Parallel Implementation of the Algorithms

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 and m is the number of grid points along the y-axis. Suppose we are implementing this method on $R \times S$ mesh connected computer. Denote the processors by

$$P_{i1,j1}: i1 = 1,2,\ldots,R \text{ and } R < l,\ j1 = 1,2,\ldots,S \text{ and } S < M .$$

The processors P_{i1j1}, are connected as shown in Fig. 5.8. 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

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

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

Fig. 5.8 Connected processors

$$\lfloor P_{1j1} \rfloor \leftrightarrow \lfloor P_{2j1} \rfloor \leftrightarrow \lfloor P_{3j1} \rfloor \leftrightarrow \cdots \leftrightarrow \lfloor P_{Rj1} \rfloor$$

Fig. 5.9 Row-wise processor communication

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

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

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. 5.9. 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. 5.10. Then each processor computes the values $u_{i,j}^{(p+1)th}$ of its assigned group.

$[P_{i1,1}]$

$\updownarrow$

$[P_{i1,2}]$

$\updownarrow$

$\vdots$

$\updownarrow$

$[P_{i1S}]$

Fig. 5.10 Column-wise processor communication

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.

5.9 Parallel Domain Decomposition Algorithms

The domain partition for the parallel algorithm described in this section is based on the availability of processors, i.e. if there are $p \times q$ processors available, then the discrete domain is divided $p \times q$ sub-domains and each sub-domain $\Omega_{i,j}$ is of equal size as shown in Fig. 5.11. The values of p and q are chosen to be close to each other.

$\Omega_{1,q}$	$\Omega_{2,q}$	. . . . .	$\Omega_{p,q}$
. . . . .	. . . .	. . . . .	. . . . .
$\Omega_{1,2}$	$\Omega_{2,2}$	. . . . .	$\Omega_{p,2}$
$\Omega_{1,1}$	$\Omega_{2,1}$	. . . . .	$\Omega_{p,1}$

Fig. 5.11 The domain divided into $p \times q$ sub-domains

Before more discussion of the parallel algorithm, description on the allocation of data associated with the sampling points on the sub-domains, interface boundaries and the boundary intersection points to the processors is discussed first.

We illustrate the data distribution (to different processors) using Fig. (5.12), which displays the sub-domain $\Omega_{i,j}$ with its interface boundaries and some of its neighboring sub-domains. The four interface boundaries of the sub-domain are identified by b_1, b_2, b_3, and b_4 (see Fig. (5.12), b_1, b_2, b_3, and b_4 are part of the set of all interface boundary points excluding the intersection points of the interface boundaries $(B = \Omega_1, \Omega_2, \ldots, \Omega_n)$. The four points of the sub-domain are identified by ip_1, ip_2, ip_3 and ip_4, and the sub-domain $\Omega_{i,j}$ are part of B^c, where B^c is the sampling points excluding the interface boundaries but including the intersection points. With this notations explained, the data distribution for the parallel implementation is the following:

5.10 Data Allocation in the Algorithms

Assign the sub-domain $\Omega_{i,j}$ to processor $P_{i,j}$ for $i = 1,2\ldots, p$ and $j = 1,2,\ldots, q$. Assign the interface boundaries b_3, b_4 and the intersection point ip_3 to processor $P_{i,j}$. According to the above allocation scheme, the rest of the interface boundaries and intersection points in Fig. (5.12) (i.e. b_1, b_2, ip_1, ip_2 and ip_4) are assigned to the processors processing neighboring sub-domains, that is, the interface boundary b_1 and the intersection point ip_4 is assigned to processor $P_{i-1,j}$. The intersection point ip_1 is assigned to processor $P_{i-1,j-1}$. The interface boundary b_2 and the intersection point ip_2 are assigned to processor $P_{i,j-1}$.

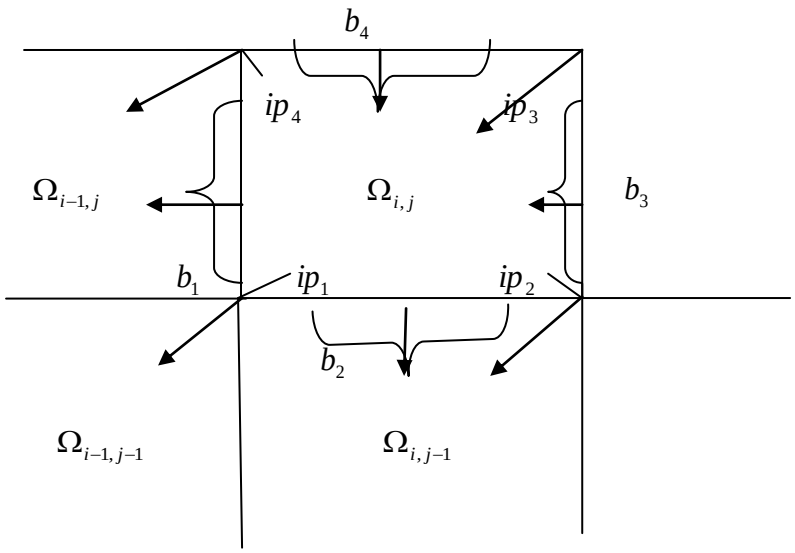

Fig. (5.12) The assignment of boundaries and intersection points of a sub-domain to processors

With the data allocation scheme given, our various iterative schemes for computing the solution at $(k+1)-$th time-step from the current $(k)-th$ time-step is the following:

1) Compute the solution $u^{k+1/2}$ on the interface boundary (i.e. on B). Then send the computed solution $u^{k+1/2}$ on interface boundaries to its neighbor (as shown in Fig. (4.6)) as follows.

a) *For processor $P_{i,j}$ as shown in Fig. (5.8), if a right-neighbor is present (i.e. processor $P_{i+1,j}$), then processor $P_{i,j}$ sends the interior vertical interface boundary b_3 (refer to Fig. 5.8)) 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 b_1 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 b_4 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 b_2 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 $k+\frac{1}{2}$ at the interface boundary conditions.

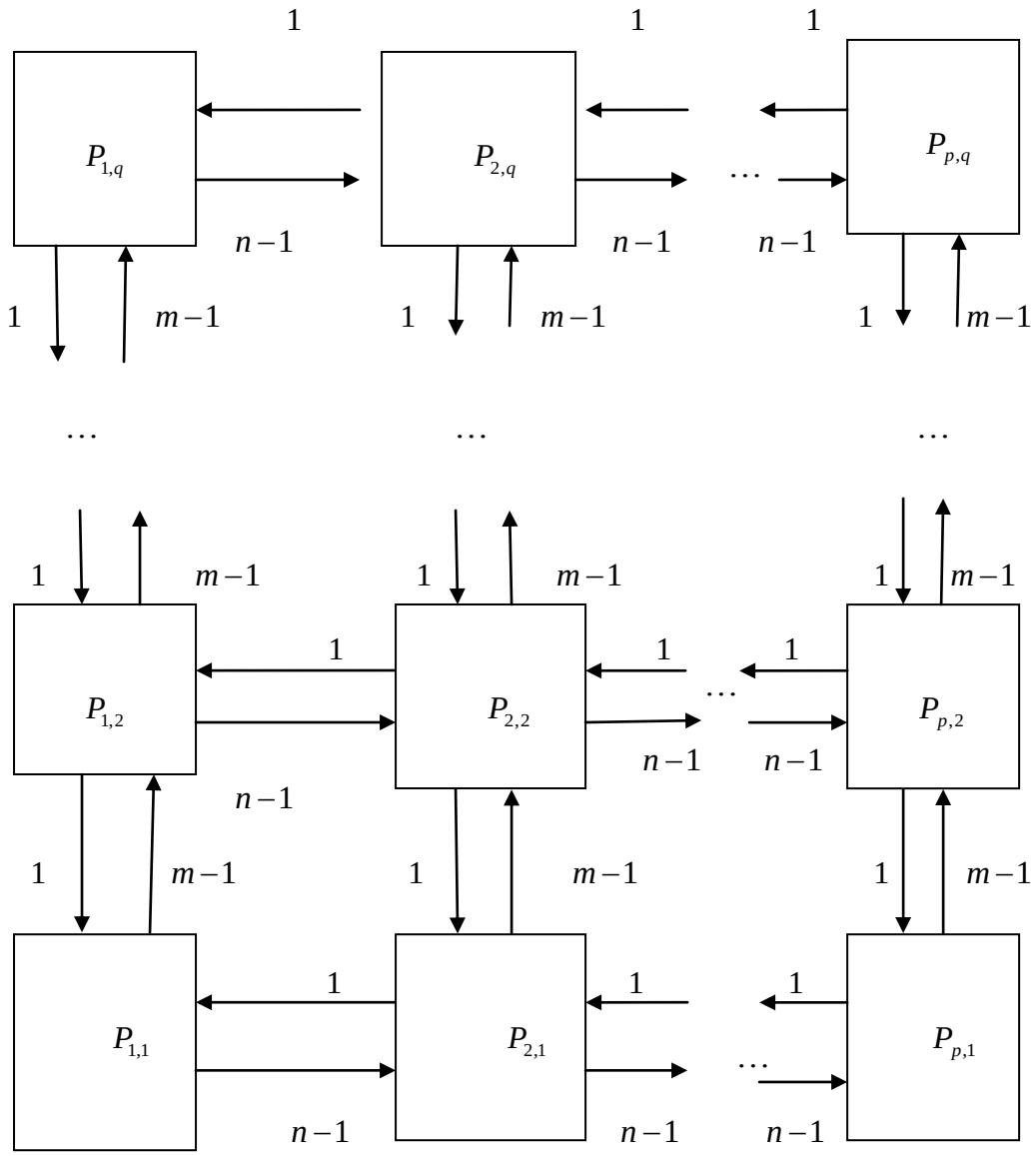

Fig. 5.13: Inter-processor communication patterns and amount of transferred data of step 1, assuming each processor is assigned a sub-domain of grid size $\frac{M}{p} \times \frac{N}{q}$, where

$$m = \frac{M}{p} \text{ and } n = \frac{N}{q}$$

2) Compute solution $u^{k+1/2}$ on B^c, i.e. the sampling points within the domain and the boundary intersections. The computation of this step can be solved mutually independently on the sub-domains and thus in parallel. No communication is necessary in this step.

3) Compute solution u^{k+1} on B^c. *Then part of the computed solution is transferred for the computation of* u^{k+1} on *B. The data communicated in this step is the following (shown in Fig. 4.7)).*

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* ip_3 *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 at* ip_4 *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 at* ip_2 *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* b_2 *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* b_1 *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 $P_{i+1,j}$.

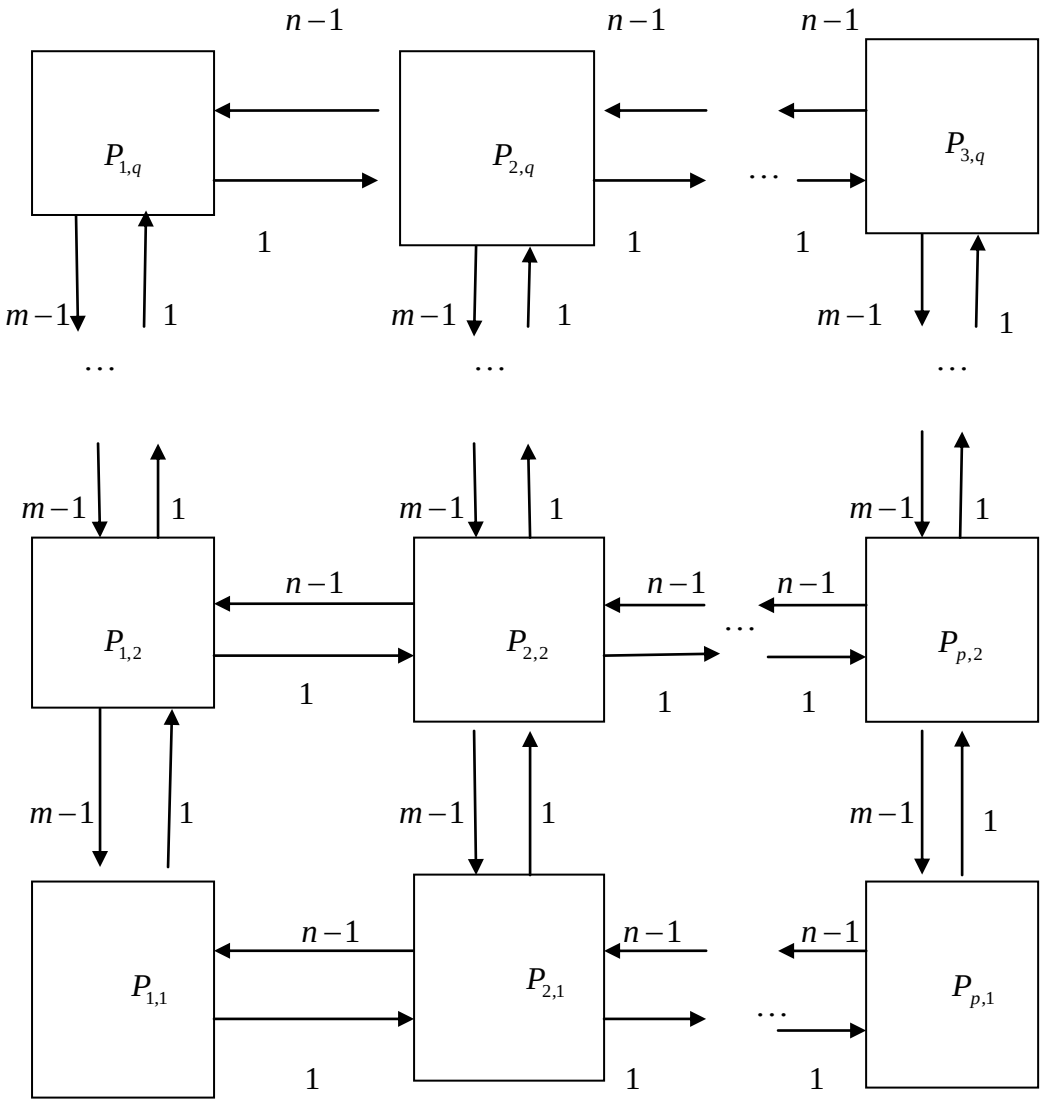

Fig. 5.14: Inter-processor communication patterns and amounts of transferred data of

step 3. Assuming each processor is assigned a sub-domain of grid size $\frac{M}{p} \times \frac{N}{q}$, where

$$m = \frac{M}{p} \text{ and } n = \frac{N}{q}$$

4) Compute the solution u^{k+1} on the interface boundaries B with the solution u^{k+1} on nearby sub-domains which it received in previous step (step 3) as boundary conditions. This step does not involve any communication with neighboring sub-domains. It involves the following:

a) *If processor* $P_{i,j}$ *has a right-neighbor (i.e. processor* $P_{i+1,j}$*), compute* u^{k+1} *on interface boundary* b_3.

b) *If processor* $P_{i,j}$ *has an upper-neighbor (i.e. processor* $P_{i,j+1}$*), compute* u^{k+1} *on interface boundary* b_4.

5.11 Parallel Implementation of the 3-D Algorithm

The realistic simulation of complex 3D structures may require a large number of grid nodes and can quickly push the limits of a single processor computer in terms of memory needs and processor speed. To achieve acceptable computation times, our finite difference scheme has been design to run on parallel computers using MPI and PVM. Thus, our parallel approach is optimized for distributed memory architecture.

Given the number of mesh nodes in each dimension and the number of available processors, we divide the mesh into 3D subsets representing the different processor domains. It is important that the mesh subsets are as equal in size as possible. Otherwise, long idle times result from an unbalanced load and thus deteriorate the parallel performance. For simplicity, suppose that each processor is assigned to only one grid node in a 3D mesh. The time-stepping scheme requires message passing across processor boundaries, which alternates with the fields updating steps. The fields which are needed in order to complete the field update at a given node (i, j, k), yet are calculated by an adjacent processor, are called "ghost" values.

One must ensure that an effective parallel implementation of the algorithm leads to a substantial increase in the computational count per data exchange, major reduction in synchronization frequency and subsequent decrease in communication sessions. Here, we involve the assignment of block of grids to each task to a surface so that each task only communicates with its limited nearest neighbors. Only the top, bottom, left

and right surfaces of the block need to be exchanged between neighbouring tasks. As an example, Fig. 5.1 illustrates the pattern of communication with 4 tasks $(p = 4)$. It is important to maintain load balancing in the distribution of m grids to tasks $P_1, P_2, \ldots, P_p$. The data decomposition of the 3-D ADI algorithm is implemented and run simultaneously at every time level, where each task is allocated m/p grids. It proceeds for every task at each time level until the approximate solutions are computed at the last time level. These tasks then send the approximate solutions to the master, which in turn processes the global solutions.

On the MPI and PVM parallel implementation of the 3-D ADI is based on one master and many tasks. The master program is responsible in constructing the m grids sizes, computing the initial values, partitioning the grid into blocks of surface, assigning these blocks to the p task modules, distributing the task to different processors and receiving approximate solutions from the tasks. Each block that is assigned to a task module is composed of m/p blocks.

A task process starts computations after it receives a work assignment. A task module $q(q < P)$ performs the 3-D ADI iterations on the grid points of the assigned block which is composed of surfaces with indices between

$$SUR_q(start) = \frac{m(q-1)}{p} \quad \text{and} \quad SUR_q(end) = \frac{m}{p} - 1,$$

where SUR refers to the surface. The task q will transmit

i. to its upper neighbor $(task\ q-1)$, $SUR_q(start)$ and receive from it

$$SUR_q(start) - 1 = SUR_{q-1}(end)$$

ii. to its lower neighbor $(task\ q+1)$, $SUR_q(end)$ and receive from it

$$SUR_q(end) + 1 = SUR_{q+1}(start).$$

Since multiple copies of the same task code run simultaneously, the tasks will exchange data with their neighbors at different times. At this point, the barrier function is called by the MPI and PVM library routine for synchronization.

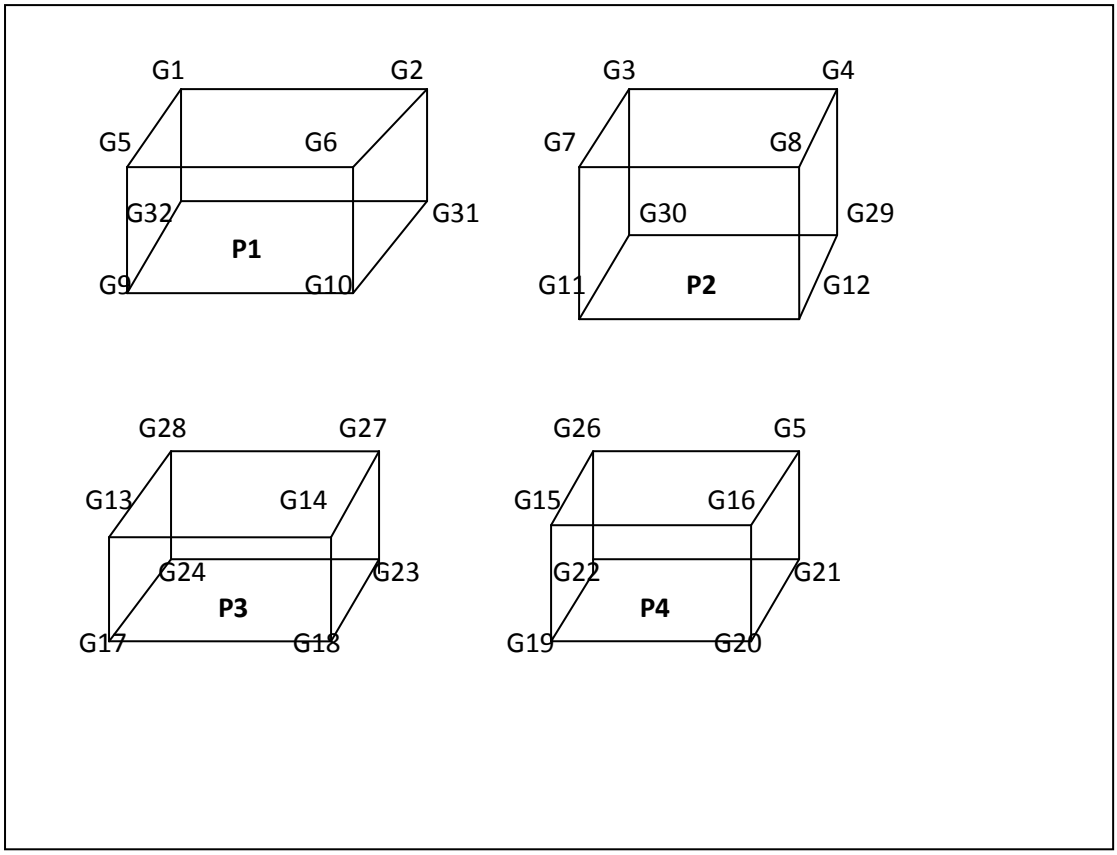

Fig. 5.15. Communication of data exchange between 4 tasks