A Survey of Asynchronous Finite-Difference Methods for Parabolic PDEs on Multiprocessors *

appeared in Applied Numerical Methods, Vol 12, pp. 27-45, May 1993

D. Amitai†, A. Averbuch†, M. Israeli‡, S. Itzikowitz†
E. Turkel†
† School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel
‡ Faculty of Computer Science, Technion, Haifa 32000,Israel

Abstract

A major obstacle to achieving significant speed-up on parallel machines is the overhead associated with synchronizing the concurrent processes. Removing the synchronization constraint has the potential of speeding up the computation. Recently developed asynchronous finite-difference schemes for parabolic PDEs designed for parallel computation are surveyed. Specifically, we consider the *asynchronous* (**AS**), *corrected-asynchronous* (**CA**), *time-stabilizing* (**TS**), *parametric* (**PAR**) and *hybrid* (**HYB**) schemes. The AS scheme is applicable only to steady-state problems. AS, CA and TS provide first-order spatial approximations. TS, however, minimizes the first order errors by maintaining the *time-stabilizing* property which also enables it to be implemented on parallel machines in which some processors have persistent speed differences. The PAR algorithm gives a second-order approximation is inefficient. The HYB algorithms are high-order schemes which have the potential to be applicable also to nonlinear problems.

1 Introduction

Although the use of parallel machines becomes more popular, most algorithms for the numerical solutions of PDEs on parallel machines still rely heavily on synchronizing concurrent processes[12]. There is an inherent inefficiency in the synchronization requirement, which is two-fold. First, faster processors are delayed until the slowest processor (or the one with the heaviest load) finishes its task, thus the pace of the algorithm is dictated by the slowest processor. There are various reasons why certain processors will be ahead of others, even when they are physically configured at the same speed. Listed below are some of them:

Random noise - Any unpredictable perturbation that can cause a momentary delay.

*This research was partially supported by a grant from The Basic Research Foundation administrated by The Israel Academy of Sciences and Humanities

Multi-user environment - In such an environment many tasks are simultaneously assigned to each processor. Since a single processor can be used for any of those tasks at any particular time, we cannot predict what part of its time will be devoted to a particular problem.

Master/Slave - In such an environment one processor, the *master*, has to perform additional tasks as compared to the other processors. For example, I/O operations, scheduling or load balancing tasks. Therefore, it may be slow in performing the problem under discussion.

Load Balancing - In many problems it may be convenient to divide the work load unequally among the processors due to different boundary conditions or approximation schemes that are used. In general the partition of nodes among processors is dictated by numerical and physical considerations rather than just by computer architecture considerations. Thus processors that are more heavily loaded will appear to be slower than the others.

Second, there is a delay period associated with the synchronization mechanism itself whether it is setting semaphores in a shared memory environment or waiting for a message to arrive in a message-passing environment, or any other possible implementation (for example, setting a time bound on the duration of an iteration, so that the next iteration starts only after this time bound is exhausted). Independently of which implementation is used, a slow communication channel slows down the progress of the entire computation. Moreover, in some situations synchronization may cause contentions over communication resources and memory access, that require careful implementation using additional mechanisms such as *locking*.

A particular situation where tasks are to be repeated is that of iterative computation. Numerical properties of iterative algorithms for PDEs usually depend on the assumption that the iterations are *synchronized*. This is equivalent to assuming that the algorithm is governed by a global clock so that the start of each iteration is simultaneous for all processors. The implementation of a synchronous algorithm in an inherently asynchronous architecture requires that a *synchronization mechanism* is used in order to guarantee correct execution. This may considerably degrade the efficiency of those algorithms. An asynchronous algorithm can potentially reduce the synchronization penalty since each processor can execute more iterations when it is not constrained to wait for the most recent results of the computation in other processors. In addition, asynchronous algorithms eliminate the programming efforts involved in setting up and debugging the synchronization mechanism and also simplifies the task management (see Fig. 1).

The approach presented here was triggered by [12] p.192: "An alternative that has special appeal in the case of the Jacobi or SOR iterations is to let the processors run asynchronously... In its simplest form, in the present context, we would simply not worry about synchronization at each iteration in carrying out the Jacobi sweeps of the multicolor SOR method; that is, at any given time, the Jacobi iteration would take as its data the most recently updated values of the variables. In general, this asynchronous iteration will deviate from the synchronized one but this need not diminish the rate of convergence."

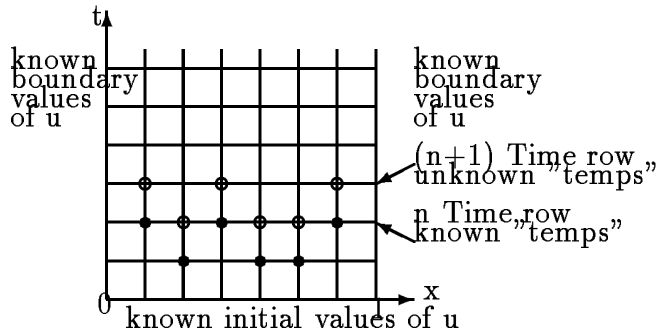

Figure 1: An asynchronous solution of a sample problem

The first usage of *asynchronous iterations* for the solution of systems of linear equations, is due to [6]. Asynchronous Iterative Methods for Multiprocessors are also discussed in [4], and are used for the solution of ordinary differential equations by [10].

Since very often the numerical solution of elliptic PDEs reduces to the solution of systems of linear equations, one may use asynchronous and semi-asynchronous algorithms for the solutions of such systems (see for example [5, 6, 9, 11]). Alternatively, asynchronous solution of a fixed point problem can be applied as for a 2D steady-state heat equation[7].

In this paper we survey *asynchronous finite-difference schemes* for the multidimensional heat equations in a spatially bounded domain. For simplicity, we consider

$$\frac{\partial u}{\partial t} = \sum_{s=1}^{d} a_s^2 \frac{\partial^2 u}{\partial x_s^2} + F(\vec{x}, t) \tag{1.1}$$

in a rectangular domain, with specified initial and boundary conditions of Dirichlet type. Here, a_s^2 $(1 \leq s \leq d)$ are constant positive coefficients. Note, however, that other bounded regions and other boundary conditions are equally applicable.

An asycnhronous (AS) finite difference scheme for the steady-state problem and the corrected asynchronous scheme (CA) of [1] are discussed in the next section. Section 3 presents the time-stabilizing (TS) scheme of [2]. High-order schemes of [3] are discussed in section 4. Performance tests for all the above schemes are presented in section 5. Finally, advantages and drawbacks of all of these schemes are summarized in section 6.

2 The AS and CA algorithms

By approximating Eq. (1.1) at the grid point $(\vec{x}_{\mathbf{j}}, t_n)$ where $\mathbf{j} = (j_1, j_2, \ldots, j_d)$ and $t_n = n\Delta t$, with the forward Euler synchronous (lock-step) finite difference scheme, we obtain our SY scheme:

$$u_{\mathbf{j}}^{n+1} = u_{\mathbf{j}}^{n} + \sum_{l=1}^{d} r_l \delta_l^2 u_{\mathbf{j}}^{n} + (\Delta t) F_{\mathbf{j}}^{n} \qquad r_l = a_l^2 \frac{\Delta t}{(\Delta x_l)^2} \tag{2.1}$$

In Eq. (2.1), $u_{\mathbf{j}}^{n}$ and $F_{\mathbf{j}}^{n}$ denote the values of u and F at grid point $(\vec{x}_{\mathbf{j}}, t_n)$ respectively, $u_{\mathbf{j}}^{n+1}$ is the value of u at $(\vec{x}_{\mathbf{j}}, t_{n+1})$, and δ_l^2 denotes the central difference in the space coordinate l. Thus,

$$\delta_l^2 u_{\mathbf{j}}^{n} = u_{\mathbf{j}-\mathbf{1}_l}^{n} - 2u_{\mathbf{j}}^{n} + u_{\mathbf{j}+\mathbf{1}_l}^{n}$$

where $\mathbf{1}_l = (0, 0, \ldots, \underbrace{1}_{l-th\ component}, \ldots, 0, 0)$ while $\mathbf{j} - \mathbf{1}_l$ and $\mathbf{j} + \mathbf{1}_l$ are the neighbors of the point $\mathbf{j}$ in the positive and the negative directions along the l-axis, respectively.

Note that unlike the following approximations, the terms of the right hand side of Eq. (2.1) are all evaluated at the *same* time level $t = n\Delta t$.

In the synchronous scheme of Eq. (2.1), the $(n+1)^{th}$ time-step of the grid point $\mathbf{j}$ is being performed only after the n^{th} time-step of all other grid points had been completed. We now remove this synchronization restriction (and yet maintaining Eq. (2.1)), allowing the time-steps of any grid point $\mathbf{j}$ to be performed asynchronously by using the most recent available values at the neighboring grid points $\mathbf{j} \pm \mathbf{1}_l$ $(1 \le l \le d)$. Taking a **local** view of the behavior at the grid point $\mathbf{j}$ and assuming a constant time interval Δt at all grid points, its time level after completing $n_{\mathbf{j}}$ time-steps is $n_{\mathbf{j}}\Delta t$ (note that there is no global time step n, since there is no global synchronization). We denote the time-level difference of the grid point $\vec{\mathbf{j}}$ relative to its neighbors in the negative and positive l-direction, by $\alpha_{\mathbf{j}_l}^{n_{\mathbf{j}}}$ and $\beta_{\mathbf{j}_l}^{n_{\mathbf{j}}}$, respectively. Then we get the following asynchronous scheme of [1] which is denoted here by AS:

$$u_{\mathbf{j}}^{n_{\mathbf{j}}+1} = u_{\mathbf{j}}^{n_{\mathbf{j}}} + \sum_{l=1}^{d} r_l \left(u_{\mathbf{j}-\mathbf{1}_l}^{n_{\mathbf{j}}+\alpha_{\mathbf{j}_l}^{n_{\mathbf{j}}}} - 2u_{\mathbf{j}}^{n_{\mathbf{j}}} + u_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}+\beta_{\mathbf{j}_l}^{n_{\mathbf{j}}}}\right) + (\Delta t) F_{\mathbf{j}}^{n_{\mathbf{j}}}, \qquad r_l = a_l^2 \frac{\Delta t}{(\Delta x_l)^2} \tag{2.2}$$

In order to clarify our later discussions, we now introduce into Eq. (2.2) a slightly different notation. We consider two arrays of data, each with as many components as the number of grid points, (see Fig. 2). One array, denoted by $\vec{u}$, represents an array of the most recent solution values of the scheme as calculated by the processors. The other array, denoted by $\vec{t}$, contains the corresponding most recent time-levels as calculated for each grid point. A processor is allowed to change only its own values, and to read its neighbors' values. Therefore, the arrays are global in the sense that access is allowed and information is shared by all processors. In the shared memory machine model we have two global arrays whose pairs of items $u_{\mathbf{j}}$ and $t_{\mathbf{j}}$ are *atomized* (encapsulated) in such a way that the values must be accessed (read/write) together. In a message-passing type of architecture, we need the same mechanism of *atomizing* $\vec{u}$ and $\vec{t}$. However, in this case relevant array-components will reside in the local memories of each processor. In any case, a time-step counter, denoted by $n_{\mathbf{j}}$, is maintained for each grid point. Its value is incremented each time the processor had completed a new iteration (time-step) of this grid point.

Let $t_{\mathbf{j}\pm\mathbf{1}_l}^{n_{\mathbf{j}}}$ denote the most recent available time-values of the neighbors in the positive and negative l-direction, read by the processor evaluating u-value at the grid point $\mathbf{j}$ when its counter value was showing $n_{\mathbf{j}}$. Hence, if, for example, the neighbor of the grid point $\mathbf{j}$ in the positive l-direction is continuously updated faster than $\mathbf{j}$ itself, then $t_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}} > t_{\mathbf{j}}^{n_{\mathbf{j}}}$. If it is continuously updated slower than $\mathbf{j}$, then $t_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}} < t_{\mathbf{j}}^{n_{\mathbf{j}}}$.

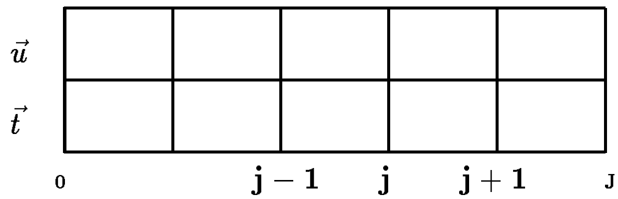

Figure 2: Solution array $\vec{u}$ and time array $\vec{t}$ for the one-dimensional case

We now rewrite Eq. (2.2) in the form

$$u_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}+1}) = u_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}}) + \sum_{l=1}^{d} r_l \bar{\delta}_l^2 u_{\mathbf{j}}^{n_{\mathbf{j}}} + (\Delta t) F_{\mathbf{j}}^{n_{\mathbf{j}}} \quad , \quad r_l = \frac{a_l^2 \Delta t}{(\Delta x_l)^2}. \tag{2.3}$$

In Eq. (2.3), $\bar{\delta}_l^2$ denotes a central difference in the direction of the space coordinate l, corresponding to the grid point $\mathbf{j}$, with the u-values of the neighbor points in the positive and negative l-direction evaluated not at the time-level $t_{\mathbf{j}}^{n_{\mathbf{j}}}$ of $\mathbf{j}$ but at their *own* most recent time levels: $t_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}}$ and $t_{\mathbf{j}-\mathbf{1}_l}^{n_{\mathbf{j}}}$, respectively. Thus,

$$\bar{\delta}_l^2 u_{\mathbf{j}}^{n_{\mathbf{j}}} = u_{\mathbf{j}-\mathbf{1}_l}(t_{\mathbf{j}-\mathbf{1}_l}^{n_{\mathbf{j}}}) - 2u_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}}) + u_{\mathbf{j}+\mathbf{1}_l}(t_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}})$$

In the one-dimensional case the asynchronous scheme (AS) becomes,

$$\begin{aligned} u_j(t_j^{n_j+1}) &= u_j(t_j^{n_j}) + r\bar{\delta}_s^2 u_j^{n_j} + (\Delta t) F_j^{n_j} = \\ &= r u_{j-1}(t_{j-1}^{n_j}) + (1-2r) u_j(t_j^{n_j}) + r u_{j+1}(t_{j+1}^{n_j}) + (\Delta t) F_j(t_j^{n_j}) \end{aligned}$$

However, as shown in [1], this scheme is no longer consistent with Eq. (1.1). Yet, it can be modified to produce a consistent scheme by using the time-level values at the neighboring grid points. We thereby obtain the Corrected-Asynchronous (CA) scheme of [1] as follows. For the calculation of the updated solution $u_{\mathbf{j}}^{n_{\mathbf{j}}+1}$, the CA scheme requires the components' values of $\vec{u}$ as before, but also the components' values of $\vec{t}$ corresponding to the grid point $\mathbf{j}$ and to the relevant neighboring points. With our notations, the finite difference CA scheme becomes:

$$u_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}+1}) = u_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}}) + \frac{\Delta t}{K} \sum_{l=1}^{d} \frac{a_l^2 \bar{\delta}_l^2 u_{\mathbf{j}}^{n_{\mathbf{j}}}}{(\Delta x_l)^2} + \frac{\Delta t}{K} F_{\mathbf{j}}(t_{\mathbf{j}}^{n_{\mathbf{j}}}) \tag{2.4a}$$

where,

$$K = 1 + \sum_{l=1}^{d} \frac{a_l^2 \delta_l^2 t_{\mathbf{j}}^{n_{\mathbf{j}}}}{(\Delta x_l)^2} \tag{2.4b}$$

and

$$\delta_l^2 t_{\mathbf{j}}^{n_{\mathbf{j}}} = t_{\mathbf{j}+\mathbf{1}_l}^{n_{\mathbf{j}}} - 2t_{\mathbf{j}}^{n_{\mathbf{j}}} + t_{\mathbf{j}-\mathbf{1}_l}^{n_{\mathbf{j}}} \tag{2.4c}$$

As shown in [1], this scheme is stable (and convergent) if

$$0 < \frac{\sum_s r_s}{1 + \sum_s r_s (\alpha_{\mathbf{j}_s}^{n_{\mathbf{j}}} + \beta_{\mathbf{j}_s}^{n_{\mathbf{j}}})} \leq \frac{1}{2} \tag{2.5}$$

In order to simplify our analysis to follow, we will use the following short notations:

$$t \triangleq t_{\mathbf{j}}^{n_{\mathbf{j}}}, \quad t^+ \triangleq t_{\mathbf{j}}^{n_{\mathbf{j}}+1}, \quad t_{\pm l} \triangleq t_{\mathbf{j}\pm\mathbf{1}_l}^{n_{\mathbf{j}}}, \quad n \triangleq n_{\mathbf{j}}, \quad n_{\pm l} \triangleq n_{\mathbf{j}\pm\mathbf{1}_l}$$

where $n_{\mathbf{j}\pm\mathbf{1}_l}$ are the most recent counters' values of the neighbors in the positive and negative l-direction, examined when the local counter of grid point $\mathbf{j}$ was $n_{\mathbf{j}}$,

$$u \triangleq u_{\mathbf{j}}(t), \quad \frac{\partial u}{\partial t} \triangleq (\frac{\partial u}{\partial t})_{\mathbf{j}}(t), \quad u^+ \triangleq u_{\mathbf{j}}(t^+),$$

$$\frac{\partial u}{\partial x_l} \triangleq (\frac{\partial u}{\partial x_l})_{\mathbf{j}}(t), \quad \frac{\partial^2 u}{\partial t \partial x_l} \triangleq (\frac{\partial^2 u}{\partial t \partial x_l})_{\mathbf{j}}(t), \quad u_{\pm l} \triangleq u_{\mathbf{j}\pm\mathbf{1}_l}(t), \text{ and } \bar{u}_{\pm l} \triangleq u_{\mathbf{j}\pm\mathbf{1}_l}(t_{\pm l}).$$

Note that while $\bar{u}_{\pm l}$ denote the neighbors' values at their *own* time-levels when the counter of the grid point $\mathbf{j}$ is showing $n_{\mathbf{j}}$, $u_{\pm l}$ denote their (estimated) values at the same time-level as of the grid point $\mathbf{j}$ (i.e. $t_{\mathbf{j}}^{n_{\mathbf{j}}}$). With these notations we rewrite Eq. (2.4a) in the following form:

$$u^+ = u + \frac{\Delta t}{K} \sum_{l=1}^{d} \frac{a_l^2 \bar{\delta}_l^2 u}{(\Delta x_l)^2} + \frac{\Delta t}{K} F \tag{2.6}$$

In one dimension it becomes:

$$u^+ = u + \frac{a^2 \Delta t}{(\Delta x)^2 + a^2 \delta^2 t} \bar{\delta}^2 u + \frac{(\Delta t)(\Delta x)^2}{(\Delta x)^2 + a^2 \delta^2 t} F$$

Equation (2.6) along with the stability restriction (2.5) represent the CA algorithm. Note that the time increment Δt is constant, and when implementing this scheme, one must ensure that (2.5) holds, before proceeding to the next time-level.

3 TS algorithm

We now generalize Eq. (2.6) and allow Δt to vary and to depend upon the grid point $\mathbf{j}$ and upon the current time-step counter-value of that point, $n_{\mathbf{j}}$. With the above notations our new scheme becomes:

$$u^+ = u + \frac{\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}}}{K} \sum_{l=1}^{d} \frac{a_l^2 \bar{\delta}_l^2 u}{(\Delta x_l)^2} + \frac{\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}}}{K} F \tag{3.1}$$

where K is given by Eqs. (2.4b) and (2.4c). Equation (3.1), along with the appropriate expression for $\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}}$ as determined by a stability restriction (see below) is our *Time-Stabilizing* (TS) scheme.

In one dimension we obtain

$$u^+ = u + \frac{a^2 \Delta t_j^{n_j}}{(\Delta x)^2 + a^2 \delta^2 t} \bar{\delta}^2 u + \frac{(\Delta t_j^{n_j})(\Delta x)^2}{(\Delta x)^2 + a^2 \delta^2 t} F$$

As discussed in [1], CA scheme can be applied only when the differences between the time levels of a grid point and of its relevant neighbors (i.e $\alpha_{\mathbf{j}}^{n_{\mathbf{j}}}$ and $\beta_{\mathbf{j}}^{n_{\mathbf{j}}}$ of Eq. (2.2)) are bounded. Therefore, the CA algorithm cannot be implemented when, for example, one or more processors are continously slower than the others, since then $\alpha_{\mathbf{j}}^{n_{\mathbf{j}}}$ and $\beta_{\mathbf{j}}^{n_{\mathbf{j}}} \to \infty$ as $n_{\mathbf{j}} \to \infty$ so that the gap between the time-levels becomes unbounded. We call this the "gapping" effect. In addition, a stability condition of the CA scheme (expression (2.5)), which has to be examined before each time-step, reduces its efficiency substantially.

As shall be described later, stability of the TS algorithm dictates the time increment $\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}}$ of point $\mathbf{j}$ to be determined by an average of its neighbors time-levels. This averaging avoids the need of examining the stability condition before each time-step (as in the CA case) on one hand, and "smears" the time-levels of the grid points on the other hand. Therefore, the TS algorithm can be implemented efficiently on a multiprocessor in which the processors are not necessarily identical, as long as their speeds do not differ too much, so that the "smearing" effect dominates the "gapping" effect. Note that speed differences can occur not only due to differences in the hardware configuration, but also due to unbalanced workload, which may be caused by the geometry or the specification of the problem.

It can be shown[2] that the TS algorithm of Eq. (3.1) is consistent with Eq. (1.1) if $\forall \mathbf{j}, n_{\mathbf{j}}, l \ \sum_{l=1}^{d} \frac{\delta_l^2 t}{(\Delta x_l)^2} \neq -1$ and $t_{\pm l} - t = O(\Delta {x_l}^2)$ so that $\frac{1}{K} = O(1)$ as $\Delta x_l \to 0 \ \forall l$.

Furthermore, it is stable if

$$\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}} = \frac{c}{2}\left(\frac{K}{\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}}\right) \qquad 0 < c \leq 1 \tag{3.2}$$

By the choice of $\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}}$ according to Eq. (3.2), our TS scheme becomes,

$$u^+ = (1-c)u + \frac{c}{\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}} \sum_{l=1}^{d} \frac{a_l^2 \bar{\mu}_l u}{(\Delta x_l)^2} + \frac{c}{2\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}} F \tag{3.3a}$$

$$t^+ = (1-c)t + \frac{c}{2\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}} + \frac{c}{\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}} \sum_{l=1}^{d} \frac{a_l^2 \bar{\mu}_l t}{(\Delta x_l)^2} \qquad 0 < c \leq 1 \tag{3.3b}$$

when

$$\bar{\mu}_l v = \frac{\bar{v}_{+l} + \bar{v}_{-l}}{2} \tag{3.3c}$$

This scheme converges to the solution of (1.1) under the above consistency assumption. However, it yields only first-order approximation [2].

In one dimension Eqs. (3.3a) and (3.4b) become,

$$u^+ = (1-c)u + c\frac{\bar{u}_+ + \bar{u}_-}{2} + \frac{c}{2}\frac{(\Delta x)^2}{a^2} F \tag{3.4a}$$

$$t^+ = (1-c)t + \frac{c(\Delta x)^2}{2a^2} + c\frac{t_+ + t_-}{2} \tag{3.4b}$$

As shown in [2] the optimal parameter value is $c = 1$, so that u^+ and t^+ are determined by the weighted averages of the corresponding neighbors values along each spatial axis.

We shall now examine the *Time-Stabilizing* property of the last scheme: $|\frac{\partial t(\vec{x})}{\partial x_l}|$ is bounded as the time-step number increases and $\frac{\partial t(\vec{x})}{\partial x_l} \to 0$ as $\Delta x_l \to 0$, where $\vec{x}$ is the position vector and $t(\vec{x})$ is a surface of the problem time for a fixed spatial mesh-size Δx_l $(1 \leq l \leq d)$. This is illustrated clearly by Figs. 2 and 3 which plot the relative time-level $t - t_s$ versus x-coordinate in the one-dimensional case where t_s is the time-level of the "slowest" grid point, and for $\Delta x = 0.05263$ and 0.02564, respectively. In these cases, each processor is assigned a single grid point. In these simulations, we scan all grid points iteratively, and determine which can be updated. All grid points are updated simultaneously at the same rate (every 9^{th} iteration), except the middle point ($x = 0.5$), which is updated at 10% "slower" rate (every 10^{th} iteration). This iterative scanning is considered as ticks of some outside global clock that determine the behavior of the system. We observe that as Δx decreases, the time-level flattens and the time-level differences between the grid points diminish. Furthermore, for a fixed mesh-size, a "steady-state" is reached in the sense that $|t_{\mathbf{i}} - t_{\mathbf{j}}|$ for $\mathbf{i} \neq \mathbf{j}$ is bounded as $n_{\mathbf{j}} \to \infty$. We emphasize, however, that for relatively large speed-or load-differences, the *Time-Stabilizing* property will disappear.

In the case of one grid point per processor, when the "slow" grid point is updated, its time-level becomes higher than that of all the others (since it is the average of neighboring time-levels incremented by $\frac{1}{2}(\Delta x)^2$) and a global maximum is attained at the "slow" grid point (see Figs. 2,3). Note, however, that if the snapshot is taken immediately after the fast processor updating, then a local minimum is obtained at the "slow" grid point and global maxima at its left and right neighbors.

In the case of several grid points per processor, there is an interaction between two opposing mechanisms. The inner grid points of the "slow" processor tend to lower the problem time-level while its edge points (which are the most affected by the fast grid points of adjacent processors) tend to raise it. If there are a relatively small number of grid points per processor, the effect of the edge points is dominant. Then the time-level attains a global maxima at the edge points of the "slow" processor or at the closest edge points of the fast processors (depending on whether the snapshot is taken immediately after an updating of the "slow" processor or immediately after an updating of the fast processors). In addition, a local minimum is attained at the midpoint of the "slow" processor (see Fig. 4). If there are a relatively large number of grid points per processor, the effect of the inner "slow" grid points becomes dominant and the problem time-level attains a funnel shape as illustrated by Fig. 5.

As indicated in [1], the CA algorithm can be applied only when the differences between time-levels of adjacent grid points are bounded. This can never happen, for example, when implemented on a parallel machine with processors of different speeds or different workload as in the example of Fig. 2 through Fig. 5. However, the *Time-Stabilizing* property enables it to be implemented in such circumstances, provided that the speed/workload differences are not too large to destroy this stabilizing effect.

In conjunction with the consistency assumptions of TS scheme, numerical simulations have shown that for large $n_{\mathbf{j}}$

$$\sum_{l=1}^{d} \frac{a_l^2 \delta_l^2 t}{(\Delta x_l)^2} = O(1)$$

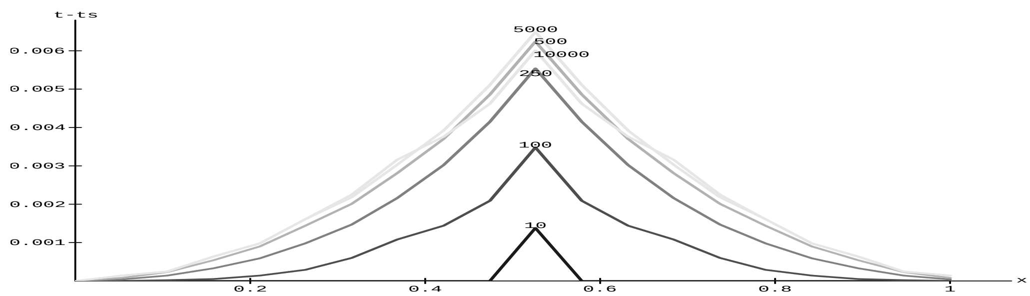

Figure 3: Dependence of t relative to the time-level t_s of the "slowest" grid point on the spatial coordinate for the 1-D case. Each processor is assigned one grid point, the middle grid point ($x = 0.5$) is updated at 10% "slower" rate than the others, and $\Delta x = 0.05263$. The numbers indicate ordinal numbers displayed by a global counter of external equal time-intervals.

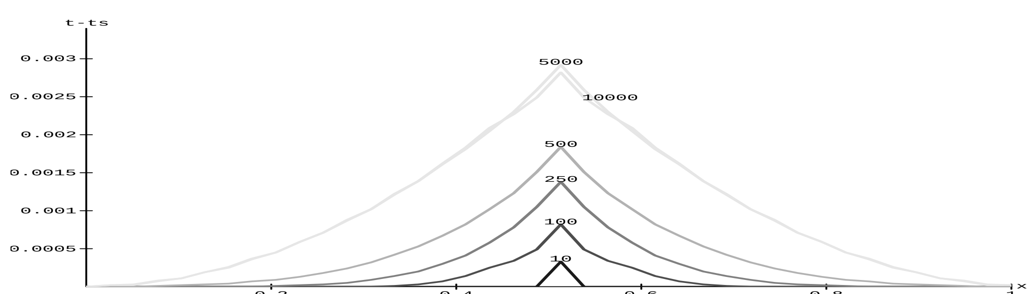

Figure 4: Dependence of t relative to the time-level t_s of the "slowest" grid point on the spatial coordinate for the 1-D case, when $\Delta x = 0.02564$. Other parameters are as in Fig.2.

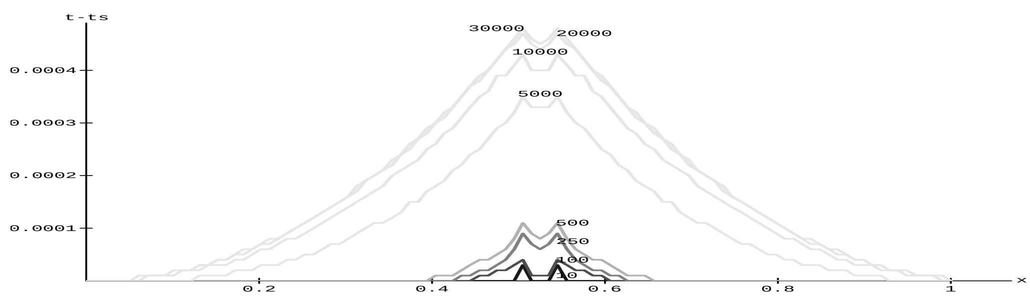

Figure 5: Simulation of 1-D problem by 20 processors, 5 points per processor, where the 11^{th} processor is 10% "slower" than the rest, and $\Delta x = 0.01010$. Other parameters are as in Fig. 2.

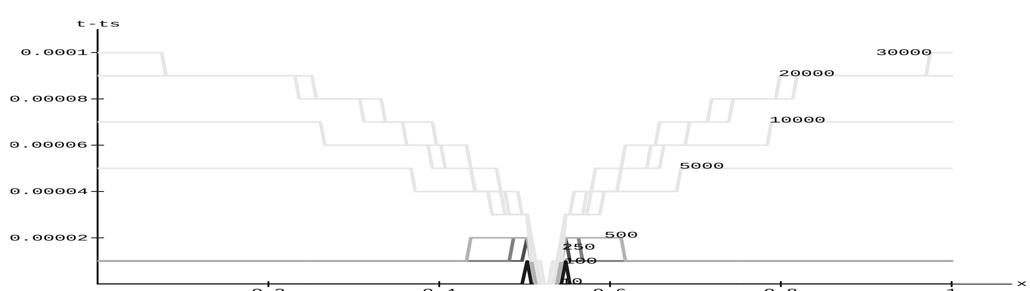

Figure 6: Simulation of 1-D prob. by 20 processors, 10 points per processor, where the 11^{th} processor is 10% "slower" than the rest, and $\Delta x = 0.00502$. Other parameters are as in Fig. 2.

In particular,

$$\Delta t_{\mathbf{j}}^{n_{\mathbf{j}}} \approx \frac{c}{\sum_{l=1}^{d} \frac{a_l^2}{(\Delta x_l)^2}}$$

This behavior is depicted clearly by Tables 1 and 2 which present the maximal number of time steps within each processor needed to reach the first time $T = 0.1$, in one-dimensional and two-dimensional problems[2], respectively. In both cases $c = 1$. n_{TS} and n_{SY} are the number of iterations need to reach $T = 0.1$ for the TS scheme and the SY scheme, respectively.

We note that very small variations of n_{TS} occured within each processor among its own grid points. In either the one dimensional case or in the two-dimensional case $n_{\mathrm{TS}} \approx \frac{1}{2} n_{\mathrm{SY}}$.

processor ID	0	1	2	3	4	5	6	7	8	9
Max n_{TS}	258	248	246	257	249	246	252	242	243	261

Table 1: Maximal Iteration number for 1-D prob. with 10×5 grid points, $T = 0.1$ $(n_{\mathrm{SY}} = 480)$

processor ID	0	1	2	3	4	5	6	7
Max n_{TS}	57	52	51	53	51	52	53	53

Table 2: Maximal Iteration number for 2-D prob. with $8 \times 2 \times 16$ grid points, $T = 0.1$ $(n_{\mathrm{SY}} = 90)$

4 High-order asynchronous schemes

As indicated earlier, CA and TS algorithms are first-order spatial approximations to the solution of (1.1). In order to obtain high-order asynchronous solutions, one may use the TS approach, but apply parametric multilevel schemes. In [3], the following parametric (PAR) scheme is investigated (see Fig. 6).

$$t_{\mathbf{j}}^{+} = \frac{c}{2\sum(\frac{a_l}{\Delta x_l})^2} + (1-c)t_{\mathbf{j}} + \frac{c}{\sum_s(\frac{a_s}{\Delta x_s})^2}\sum(\frac{a_l}{\Delta x_l})^2 t_{I_l} \tag{4.1a}$$

where

$$t_{I_l} = \frac{1}{2}(\theta_{1_l} t_{+l}^{+} + \theta'_{1_l} t_{+l} + \theta_{2_l} t_{-l}^{+} + \theta'_{2_l} t_{-l}) \quad , \tag{4.1b}$$

and

$$u_{\mathbf{j}}^{+} = (1-c)u_{\mathbf{j}} + \frac{c}{\sum(\frac{a_s}{\Delta x_s})^2}\sum_l(\frac{a_l}{\Delta x_l})^2 u_{I_l} + \frac{c}{2\sum(\frac{a_l}{\Delta x_l})^2}F_{\mathbf{j}} \tag{4.1c}$$

where

$$u_{I_l} = \frac{1}{2}(\theta_{1_l} u_{+l}^{+} + \theta'_{1_l} u_{+l} + \theta_{2_l} u_{-l}^{+} + \theta'_{2_l} u_{-l}) \tag{4.1d}$$

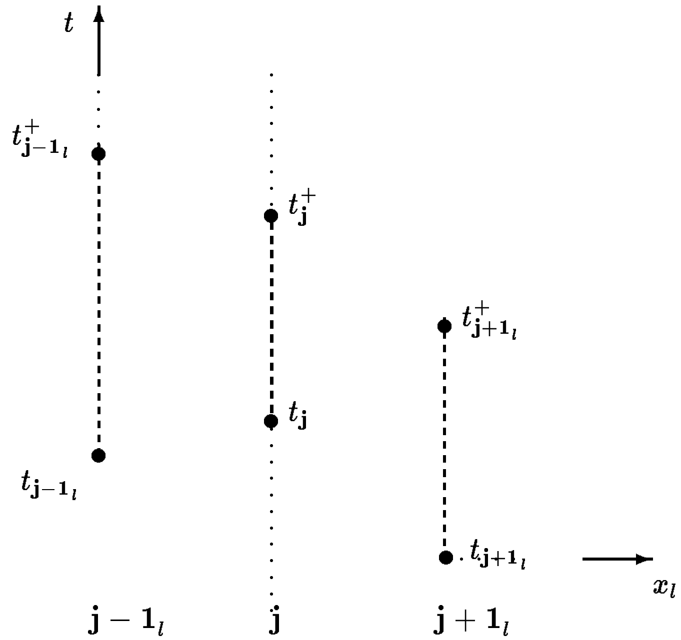

Figure 7: Six points asynchronous scheme when $t^+_{\mathbf{j+1}_l} > t_{\mathbf{j+1}_l}$ and $t^+_{\mathbf{j-1}_l} > t_{\mathbf{j-1}_l}$.

$$\begin{array}{ll} u_{\mathbf{j}} \triangleq u(\vec{x}_{\mathbf{j}}, t_{\mathbf{j}}) & u^+_{\mathbf{j}} \triangleq u(\vec{x}_{\mathbf{j}}, t^+_{\mathbf{j}}) \\ u_{-l} \triangleq u(\vec{x}_{\mathbf{j-1}_l}, t_{\mathbf{j-1}_l}) & u^+_{-l} \triangleq u(\vec{x}_{\mathbf{j-1}_l}, t^+_{\mathbf{j-1}_l}) \\ u_{+l} \triangleq u(\vec{x}_{\mathbf{j+1}_l}, t_{\mathbf{j+1}_l}) & u^+_{+l} \triangleq u(\vec{x}_{\mathbf{j+1}_l}, t^+_{\mathbf{j+1}_l}) \end{array}$$

This scheme is stable and converges to the solution of (1.1) and is of $O[\max_{1\le l\le d}(\Delta x_l)^2]$, under the following assumptions for $1 \le l \le d$:

1. $\theta_{j_l},\ \theta'_{j_l} \ge 0\ \ j = 1, 2$
2. $\theta_{1_l} + \theta_{2_l} + \theta'_{1_l} + \theta'_{2_l} = 2$
3. $\theta_{1_l} - \theta_{2_l} + \theta'_{1_l} - \theta'_{2_l} = 0$
4. $t^+_{\pm l} - t_{\mathbf{j}} = O(\Delta x_l^2) = t_{\pm l} - t_{\mathbf{j}}$
5. $\theta_{1_l} t^+_{+l} + \theta'_{1_l} t_{+l} = \theta_{2_l} t^+_{-l} + \theta'_{2_l} t_{-l}$
6. $\frac{1}{K}$ and K are both $O(1)$

Note that t_{I_l} is an intersection point of $[t_{-l}, t^+_{-l}]$ (or $[t^+_{-l}, t_{-l}]$ if $t_{-l} > t^+_{-l}$) and $[t_{+l}, t^+_{+l}]$ (or $[t^+_{+l}, t_{+l}]$ if $t_{+l} > t^+_{+l}$), and is taken as the highest one (see Fig. 7). However, evaluating t_{I_l} turns out to be time-consuming so that the corresponding synchronous scheme is preferable[3]. Instead, high-order asynchronous hybrid (HYB) schemes that are proposed in [3] turned out to be very efficient. These schemes are based on domain decomposition such that each processor is assigned the task of updating a single subdomain. Within each subdomain we use high-order finite difference scheme. The solution values on the subdomains' boundaries are determined by using the *Green's function* based solution as shall be explained below. For simplicity we shall consider here only the 1-D case. For more details for the multidimensional case see [3].

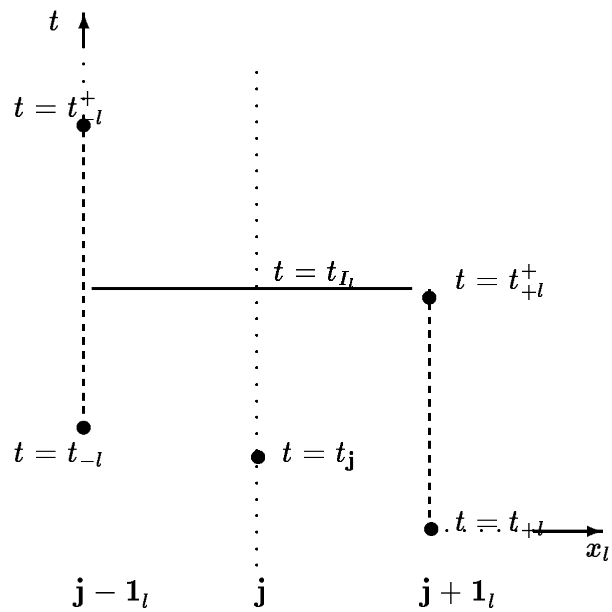

Figure 8: For evaluating the solution at $t = t^+_{\mathbf{j}}$, $t = t_{I_l}$ is taken as the highest intersection point

In this case we wish to solve the equation

$$L(u) = F(x,t) \tag{4.2}$$

where

$$L(u) = \frac{\partial u}{\partial t} - a_1^2 \frac{\partial^2 u}{\partial x^2} \tag{4.3}$$

along with the initial and boundary conditions.

Suppose that $x = \xi$ is the boundary between two adjacent subdomains and the current time levels of the left and right subdomains are $t = t_L$ and $t = t_R$, respectively (see Fig. 8).

In order to update the inner grid points of the faster processor we would need to know the value of the solution on the boundary, $u(\xi, \tau)$, where $\tau = t_L + \Delta t$ if $t_L > t_R$ and $\tau = t_R + \Delta t$ if $t_R > t_L$. In this regard we note that the adjoint operator of Eq. (4.3) is given by

$$M(v) = -\frac{\partial v}{\partial t} - a_1^2 \frac{\partial^2 v}{\partial x^2} \tag{4.4}$$

A solution of $M(v) = 0$ is given by the 1-D Green's function

$$v = \frac{1}{2a_1\sqrt{\pi(\tau - t)}} e^{-\frac{(x-\xi)^2}{4a_1^2(\tau-t)}} \tag{4.5}$$

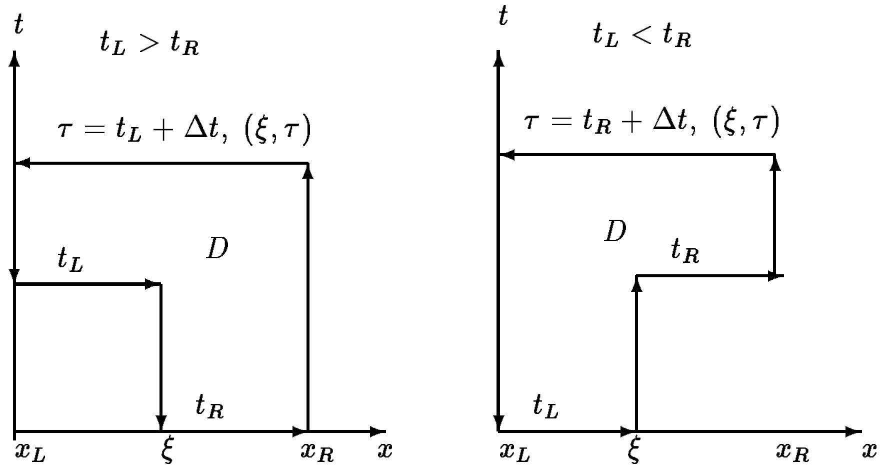

Figure 9: 1-D Asynchronous case: $x = \xi$ is a boundary between two adjacent subdomains and $x = x_L, x_R$ are some inner grid points of the left and right subdomains, respectively.

where ξ and $\tau > t$ are as defined above. It can be verifed easily that

$$vL(u) - uM(v) = \frac{\partial X}{\partial x} + \frac{\partial (uv)}{\partial t} = vF(x,t)$$

where

$$X = -a_1^2(vu_x - uv_x)$$

Then with v as in Eq. (4.5) and u as in Eq. (4.2) we get by Gauss theorem that (see Fig. 8)

$$\begin{aligned} \int_D vF(x,t)dxdt \; &= -\int_{t=t_L} uvds - \int_{t=t_R} uvds + \int_{t=\tau} uvds - \\ &-a_1^2 \int_{x=x_L} (v\tfrac{\partial u}{\partial n} - u\tfrac{\partial v}{\partial n})dt \\ &-a_1^2 \int_{x=\xi} (v\tfrac{\partial u}{\partial n} - u\tfrac{\partial v}{\partial n})dt \\ &-a_1^2 \int_{x=x_R} (v\tfrac{\partial u}{\partial n} - u\tfrac{\partial v}{\partial n})dt \end{aligned} \tag{4.6}$$

where the integration on the right hand side is in the counterclockwise direction and $\frac{\partial}{\partial n}$ represents the derivative in the normal direction along the vertical lines $x = x_L, x = \xi, x = x_R$ as the case may be. Since $v \to \delta(x - \xi)$ as $t \to \tau$, where "δ" is the Dirac delta, we have that $lim_{t \to \tau^-} \int_{t=\tau} uvds = -u(\xi, \tau)$. Since $\frac{\partial v}{\partial x}(\xi, \tau) = 0$, then by using the Trapezoidal approximation for the time integration in Eq. (4.6) we obtain

$$\begin{aligned} u(\xi,\tau) \; &= -\tfrac{2\tau-(t_L+t_R)}{4} F(\xi,\tau) - (\tfrac{\tau-t_L}{2}) \int_{x=x_L}^{\xi} (vF)(x,t_L)dx + \\ &-(\tfrac{\tau-t_R}{2}) \int_{x=\xi}^{x_R} (vF)(x,t_R)dx - \int_{x_L}^{\xi} (uv)(x,t_L)dx - \int_{\xi}^{x_R} (uv)(x,t_R)dx + \\ &-a_1^2 \left|\tfrac{t_R-t_L}{2}\right| [(vu_x)(\xi,t_R) + (vu_x)(\xi,t_L)] + \\ &+a_1^2 \left(\tfrac{t_L-\tau}{2}\right) [(vu_x - uv_x)(x_L,t_L)] + a_1^2 \left(\tfrac{t_R-\tau}{2}\right) [(vu_x - uv_x)(x_R,t_R)] + \\ &+O(\tau - t_L)^3 + O(\tau - t_R)^3 + O(t_L - t_R)^3 \end{aligned} \tag{4.7}$$

Each of the integrals in (4.7) is approximated by some numerical quadrature, using known values of u at the grid points of the left subdomain (at $t = t_L$) and of the right subdomain (at $t = t_R$), consistently with the accuracy of the numerical approximation used within each subdomain. In general, the higher the desired accuracy, the further the inner grid points x_L and x_R are to be taken in (4.7). For the derivatives in (4.7), high-order one-sided finite difference approximation are to be used.

Practically, the asynchronous hybrid scheme works as follows:

1. We divide the domain into subomains and assign each processor the task of updating one such subdomain.

2. Within each subdomain we use a high-order finite-differences scheme

3. The solution at grid points of subdomains' boundaries are determined by Eq. (4.7) in the 1-D case, or by its analogue in the multidimensional case. Each of the spatial integrals in Eq. (4.7) (or its analogue in the multidimensional case) is approximated by some high-order numerical quadrature, along with high-order one-sided finite-difference approximation for the spatial partial derivative.

For the synchronous case $t_R = t_L$, and symmetric finite-difference schemes are used to approximate the needed derivatives along subdomains' boundaries.

5 Efficiency and accuracy tests

As an example, Table 3 [1] presents efficiency ($=\frac{\text{serial run time}}{(\text{no. of processors})\times(\text{parallel run time})}$) and relative run time results, when implementing CA, AS and its synchronous counterpart (SY) on the shared-memory multi-user Sequent Balance multiprocessor, for the following 2-D steady-state problem:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + F \quad 0 < x, y < 1$$

where

$$F = 5\pi^2 \sin(\pi x) \sin(2\pi y)$$

with the initial condition

$$u(x, y, 0) = 0$$

and the boundary conditions

$$u(0, y, t) = 0 \quad u(1, y, t) = 0 \quad u(x, 0, t) = 0 \quad u(x, 1, t) = 0$$

The calculated results are compared with the analytic result given, for the steady-state, by

$$u_\infty = \sin(\pi x) \sin(2\pi y).$$

L is the number of grid points and p is the number of processors. We notice that AS is much faster than the SY and CA is at best as fast as SY.

p	pts in blk	Efficiency			Rel. Run Time	
		SY	AS	CA	AS	CA
2	8×16	0.66	1.0	0.60	0.59	1.11
4	4×16	0.38	0.645	0.35	0.58	1.09
8	2×16	0.22	0.38	0.205	0.56	1.07
16	1×16	0.11	0.20	0.10	0.55	1.03

Table 3: Efficiency and run time relative to SY for the 2-dim. steady-state non-homogeneous problem; $L = 256, \Delta x = \Delta y = 0.0667$, $\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2} = 0.5$; Calculated until $\sqrt{\frac{\sum (u_{ij} - u_\infty)^2}{L}} < 10^{-2}$

Table 4 [2] presents efficiency results for the following non steady-state problem

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \qquad 0 < x, y < 1, \quad t > 0$$

with the initial condition,

$$u(x, y, 0) = \sin(\pi x)\sin(\pi y)$$

and the following Dirichlet boundary conditions:

$$u(0, y, t) = 0 \quad , \quad u(1, y, t) = 0 \quad , \quad u(x, 0, t) = 0 \quad , \quad u(x, 1, t) = 0$$

The calculated results are compared with the analytic solution given by

$$u(x, y, t) = \sin(\pi x)\sin(\pi y)e^{-2\pi^2 t}$$

p	pts in blk	Efficiency			
		SY	AS	CA	TS
2	8×16	0.92	1.14	0.53	1.09
4	4×16	0.87	0.98	0.47	1.04
8	2×16	0.75	0.87	0.34	0.92
16	1×16	0.36	0.5	0.3	0.63

Table 4: Efficiency for the 2-dim. problem with p processors; $L = 256$, $\Delta x = \Delta y = 0.0667$, $\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2} = 0.5$, $T = 0.5$ and $c = 1$.

Table 5 [2] presents accuracy results for the above 2-D non-steady state problem, in which e denotes the maximal error magnitude and E denotes the maximal absolute relative error of the numerical result.

p	pts in blk	E				e			
		SY	AS	CA	TS	SY	AS	CA	TS
2	8 × 16	0.07	0.99	0.17	0.5	0.0000036	0.00005	0.000003	0.00002
4	4 × 16	0.07	0.99	0.5	0.21	0.0000036	0.00005	0.000003	0.000008
8	2 × 16	0.07	0.97	0.17	0.12	0.0000036	0.000049	0.0000021	0.0000022
16	1 × 16	0.07	0.99	0.17	0.18	0.0000036	0.00005	0.000003	0.000002

Table 5: Accuracy for the two-dimensional problem with p processors; $L = 256$, $\Delta x = \Delta y = 0.0667$, $\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2} = 0.5$, $T = 0.5$ and $c = 1$.

As observed, the efficiency of our time-stabilizing scheme is about the same as that of the asynchronous scheme in certain cases, and it is much higher in other cases. Furthermore, in all cases, the TS scheme efficiency is significantly higher than that of the corrected-asynchronous scheme; in certain cases it is even tripled. This is in spite of the fact that extra interpolation is needed to evaluate the solution at a specific time value, in the TS scheme. It should be noted here that since we are using a multi-user system, the corresponding delays of the serial and the parallel schemes may not be the same. For this reason, the efficiency might be in certain cases slightly greater than one (see the AS scheme in Table 4). However, in the TS case, as explained above, for a given T the number of time-steps is reduced by about 2, as compared to the SY case. Therefore, in this case, the efficiency may be significantly better. This is even more evident in cases where the parallel algorithms exploit the cache better than the serial algorithm in which all the data is referenced by the same processor. In such paralll architectures where a local memory cache is used by each processor. As a result the efficiency may become much higher than one.

As was already indicated AS scheme is not consistent with the heat equation. It is accurate only for steady-state results, while the corrected asynchronous scheme provides accurate results also for intermediate (non steady-state) values. As noticed from table 5 TS and CA both provide about the same accuracy. Due to the randomness of the phase delays, the shown accuracies of the different approximations were not obtained under the same circumstances. Different runs of the same approximation provided slightly different results and in other cases (not shown here) the TS was slightly more accurate than the CA scheme. However, one can say that in general the CA and the TS schemes provide about the same accuracy. Both the TS and the CA schemes are less accurate than the synchronous scheme, because of the lower order of their truncation error ($O(\Delta x_l)$ instead of $O(\Delta x_l^2)$). Due to the dimensional additivity of the truncation error, these schemes are even less accurate for multidimensional problems. For such problems, if high accuracy is sought, in general one should use higher order schemes [3] since for the TS one would then need a very fine grid that would increase its run-time substantially and make the SY approximation preferable. Nevertheless, under certain circumstances, when the spatial increments are relatively large, the accuracy of the TS may become about the same as that of the SY scheme and yet providing higher efficiency [2].

The PAR scheme is constructed to improve the accuracy of time-stabilizing schemes

[3]. Performing PAR at each grid point is almost as accurate as SY, but, at the cost of additional memory (twice longer arrays as compared with the TS scheme) and additional processing-time (even more than twice as slow than SY), needed for evaluating t_{I_l}.

We note that the higher-order accuracy of our PAR scheme was even more evident in tests we have done (not shown here) on a serial computer simulating a multiprocessors machine. In cases where all simulated processors run at the same speed, except one which was slower than the others (10% up to 35% slower), the high-order accuracy of the PAR scheme was even more evident.

The PAR scheme is slow mainly due to the complicated search for the intersection point t_{I_l} performed for each grid point and for $1 \leq l \leq d$. Nevertheless, this search for each single grid point is not necessary since we may assign several grid points per processor such that all points within a single processor's subdomain are synchronized (i.e. are guaranteed to be in the same time-level at any iteration), while the PAR scheme can be used only at grid points of subdomains' boundaries.

However, using explicit scheme within the subdomain restricts the time-step to that of the explicit scheme even in cases where the PAR allows us to maintain larger time-steps on the subdomains' boundaries. Using implicit scheme within a subdomain eases this restriction, but then the time-step is restricted by the PAR which in an explicit scheme in nature. In our tests, run-time of implicit scheme equals the run-time of an explicit scheme with time-step three times as much.

We conclude that using the higher-order Parametric Asynchronous finite difference scheme seems impractical. It requires too much processing per grid point or on subdomains' boundaries, and yet is explicit in nature and thereby limits the allowed time-step severly.

The performance of the HYB scheme is illustrated by Tables 6 and 7, for the above two-dimensional non-steady state problem. In these tests, the problem domain is divided along the x-axis into the subdomains thereby forming strips parallel to the y-axis. Within each subdomain we used an ADI implicit scheme. Along the subdomains' boundaries we used the Trapezoidal Rule for the spatial integrals in either the synchronous or asynchronous case. When calculating the integrand we use a pre-prepared table to avoid a repetition of the same calculation. It is calculated to the second order in space.

Among the two, HYB-AS performance is better and in our experiments it was about 15-25% faster than HYB-SY. The efficiency decreases as p increases since the computational load of each processor, relative to communication and synchronization (for HYB-SY only) overhead is smaller. The HYB-AS scheme acheives an excellant efficiency not only due to the elimination of synchronization delays but also due to overlapping the boundaries. Thus a slower processor can used the values calculated by a fast one.

When the approximations on the subdomains' boundaries are more accurate than the scheme used for the internal grid points, the final result is more accurate. In this case accuracy improves as the number of processors increases. The synchronization overhead is more significant as the number of processors to be synchronized is increased. We note that as the number of iterations increases the efficiency of our hybrid scheme becomes better, since the initial overhead of splitting the domain is less evident relative to the workload.

p	pts in blk	Efficiency	
		HYB-SY	HYB-AS
2	128 $\times$ 256	0.925	0.99
4	64 $\times$ 256	0.79	0.88
8	32 $\times$ 256	0.68	0.82
16	16 $\times$ 256	0.61	0.78

Table 6: HYB's efficiency for the 2-D problem with p processors; with 256×256 grid points, $\Delta x = \Delta y = 0.00392$, $\Delta t = 0.00111$ and $T = 0.5$

.

p	pts in blk	E		e	
		HYB-SY	HYB-AS	HYB-SY	HYB-AS
2	128 $\times$ 256	0.00002527	0.00002527	0.1 $\times 10^{-8}$	0.1 $\times 10^{-8}$
4	64 $\times$ 256	0.00002527	0.00002527	0.1 $\times 10^{-8}$	0.1 $\times 10^{-8}$
8	32 $\times$ 256	0.00002527	0.00002527	0.1 $\times 10^{-8}$	0.1 $\times 10^{-8}$
16	16 $\times$ 256	0.00002527	0.00002483	0.1 $\times 10^{-8}$	0.1 $\times 10^{-8}$

Table 7: HYB's accuracy for the two-dimensional problem with p processors; $L = 256 \times 256$, $\Delta x = \Delta y = 0.00932$, $\Delta t = 0.00111$ and $T = 0.5$

6 Summary

This paper surveys recently developed asynchronous finite differences schemes for linear parabolic PDEs, to be implemented on parallel computers. These schemes are based on domain decomposition such that each subdomain is assigned the task of updating a single subdomain. The asynchronous scheme AS is applicable only to steady-state problems. The CA algorithm is applicable also to non steady-state problems, but it provides only first-order spatial approximation and its efficiency is poor. The TS scheme is still first-order approximation but it provides excellent efficieny. Furthermore,, unlike other schemes, due to its *time-stabilizing* property it is suitable for parallel machines in which some processors are continously slower than others, provided that the speed-differences are not too large. High-order parametric multilevel schemes are impractical mainly due to the overhead of the search for needed interface points. Finally, the hybrid schemes (HYB) (in which high-order finite-differences schemes are used within each subdomain and subdomains' boundaries values are determined by Green's solution) provide high-order approximation and high efficiencies. Furthermore, current investigation shows that the hybrid algorithm can be modified and implemented for non linear parabolic PDEs. Synchronization overhead could be, under certain circumstances, tens of percents of the corresponding synchronous processing time.

References

[1] D. Amitai, A. Averbuch, S. Itzikowitz and E. Turkel, *Asynchronous and Correcred Asyhcnornous Numerical Solution of parabolic PDEs on MIMD Multiprocessors*, Numerical Algorithms, Vol. 6, No. III-IV, pp. 275-296, 1994; Also in Proc. of the 4^{th} SIAM Conference on Parallel Processing for Scientific Computing, 131-136, Dec. 11-13, Chicago, USA (1989); also ICASE Report No. 91-59, July 1991.

[2] D. Amitai, A. Averbuch, M. Israeli, S. Itzikowitz, *Parallel Adaptive and Time-Stabilizing Schemes for Constant-Coefficients Parabolic PDEs*, Computers & Mathematics with applications,Vol. 24, No. 10, pp. 33-53, 1992; also Fifth SIAM Conference on Parallel Processing for Scientific Computing, March 25-27, 1991, Houston, USA;

[3] D. Amitai, A. Averbuch, M. Israeli, S. Itzikowitz, *On Parallel Asynchronous High-Order Scheme for PDEs in a Bounded Domain*, IMACS PDE-7, June 22-24, 1992, Rutgers, USA; (submitted for publication).

[4] G. M. Baudet, *Asynchronous Iterative Methods for Multiprocessors*, J. of the ACM, **25**, 226-244 (1978).

[5] R. Barlow, D. Evans, *Synchronous and Asynchronous Iterative Parallel Algorithms for Linear Systems*, Comput. J., **25**(1982), pp. 56-60.

[6] D. Chazan and W. Miranker, *Chaotic Relaxations*, J. of Linear Algebra Applications, **2**, 199-222 (1969).

[7] N.M. El-Tarazi, N.M. Anwar, *Asynchronous algorithms for heat conduction in composite media*, J. of University of Kuwait, **16**(1989), pp. 193-206.

[8] D.J. Evans, *New parallel algorithms for partial differential equations*, Parallel Computing 83 (Eds J. Joubert M. feilmeier and U. Schendel), 3-56. Elsevier Science Publishers B.V., North Holland (1984).

[9] A. Lin, *Aspects of computation on asynchronous parallel processors*, Proceeding of the OFIP WG 2.5 Working Conference (1989) 79-85.

[10] D. Mitra, *Asynchronous Relaxations for the Numerical Solution of Differential Equations by Parallel Processors*, SIAM J. Sci. Stat. Comput., **8**, 43-53 (1987).

[11] R. Morandi, F. Sgallari, *Parallel semi-asynchronous algorithms for the iterative solution of linear systems*, Supercomputer **6**(1989), 17-25.

[12] J. M. Ortega and R. G. Voigt, *Solution of Parallel Differential Equations on Vector and Parallel Computers*, SIAM Review, **27**(2), 149-239 (1985).

[13] D. Reed, M. Patrick, *A Model of Asynchronous Iterative Algorithms for Solving Large Sparse Linear Systems*, Proceedings 1984 International Conference on Parallel Computing, 1984, pp. 402-409.