The Performance of Parallel Iterative Solvers

D. PETCU
Department of Computer Science
Western University of Timișoara
B-dul Vasile Pârvan 4, 1900 Timișoara, Romania

Abstract—A performance model is constructed for parallel iterative numerical methods under the assumption of a message-passing computing system. Arguments are given for the fact that the speedup of parallel iterative methods is mainly influenced by the speedup at one iterative step. Using the theoretical model, it is proved why explicit iterative methods for ordinary differential equations are inefficient in implementation on distributed memory multiprocessor systems. Numerical tests on parallel and distributed computing environments confirm the correctness of the theoretical model at least in the case of iterative methods for ordinary differential equations and time-dependent partial differential equations. © 2005 Elsevier Ltd. All rights reserved.

Keywords—Parallel numerical methods, Differential equations, Code performance.

1. INTRODUCTION

Large computational problems can successfully be treated on modern multiprocessor computers. However, access to a fast high-speed computer is not sufficient. One must also ensure that the great potential power of the computer is correctly exploited. The requirement that the programming work should be properly done is by no means trivial, and, on some of the new computers, the choice of the proper numerical algorithms and their efficient implementation is in fact a very difficult task, which requires knowledge about numerical algorithms and parallel techniques. The message passing technique will be used here. It is a programming paradigm which makes the task to solve a complex problem easier and faster for the user.

Section 2 proposes a performance model for an iterative numerical method. The model is based on the data-flow principles. Section 3 presents some numerical experiments. It is proved the intuitive assumption that the speedup of parallel or distributed implementations of an iterative method can be estimated by the speedup measured in one iterative step. Own parallel codes, recently reported in [1,2] are used on a cluster of PCs. The tests validate the statements regarding the theoretical model.

2. PERFORMANCE MODEL FOR ITERATIVE METHODS

A performance model is proposed for an arbitrary parallel step-by-step numerical method using the programming paradigm of message passing (designed for distributed memory parallel computers, clusters or network of workstations).
2.1. Process-Time Diagrams

A process-time diagram is defined here as a set of threads (or tasks) with a partial order of their execution times. A thread includes only computations and inter-thread communications (Figure 1). The computations are done in a sequential manner.

Let $M$ be the dimension of the problem to be solved by the iterative method, $N$, the number of steps to be performed (can depend on $M$), $n$, the current step, $n \in \{1, \ldots, N\}$.

The partial order (in time) is expressed by thread identifiers: thread $(i, j)$, depending on $M$ and $n$, follows in time after those threads with the first index least than $i$. So, for a given thread $(i, j)$, the execution of all the threads with $k < i$ happens before its execution, the execution of all the threads with $k > i$ happens after its execution, and the execution of the threads $(i, k)$ with $k \neq j$ can happen simultaneously with its execution.

Let $m_{ij}^{(n)}$ be the problem dimension at the thread $(i, j)$, $t_{ij}^{(n)}$, the sequential time for the inherent computations of thread $(i, j)$ (dependent on $m_{ij}^{(n)}$), $s_{ij}^{(n)}$, the number of messages to be sent after the computation is finished, $S_{ij}^{(k, n)}$, the time to send the $k$th message ($k \in \{1, \ldots, s_{ij}^{(n)}\}$; dependent on $m_{ij}^{(n)}$), $r_{ij}^{(n)}$, the number of messages to be received before the computation is started, $R_{ij}^{(k, n)}$, the time to receive the $k$th message ($k \in \{1, \ldots, r_{ij}^{(n)}\}$; dependent on $m_{ij}^{(n)}$), and $A_{ij}^{(n)}$, the time to start the thread $(i, j)$. Let $q(n)$ be the minimum number of thread levels (rows) to accomplish the $n$th step (the height of the process-time diagram; diagram's time component). So, $q(n) \geq 1$ and $i \in \{0, \ldots, q(n)\}$. By definition $t_{0j} = 0$, $r_{0j} = 0$ (the first level indicates the step's initial data) and $s_{q(n)j} = 0$ (the last level contains the step's final data) for all $j$ possible values. Let $p_j^{(n)}$ be the number of threads which can be performed simultaneous at the step $n$ and at the $i$th level. So, $j \in \{1, \ldots, p_j^{(n)}\}$. Each row is composed by at most of $p_j^{(n)} = \max_{i=1,\ldots,q^{(n)}} p_i^{(n)}$ concurrent threads (the width of the process-time diagram; diagram’s process component).

In order to execute a step-by-step method the threads are associated to concurrent processes. One process consists in the execution sequence of at least one thread. The threads with the same column index are associated to a unique process. So, all threads $(i, j)$ are associated to the $j$th process. A processor of a computational environment will run at least one process. Cyclic dependencies are not allowed in a process-time diagram. In order to ensure this requirement, all the threads of a cycle in a preliminary process-time diagram (for example, a data-flow diagram) are coupled together in one bigger thread which will be sequentially executed.

The inter-threads communications are split into two categories: inter-process communications and inner-process communications. Inter-process communication time depends on the computational environment. Inner-process communications are characterized by small values of $R_{ij}$ and $S_{ij}$, smaller than the corresponding values for inter-process communications (in fact, in the case of inner-process communications, these values reflect the access time of the memory shared...
by the communicating threads, meanwhile, in the opposite case, the values are those characteristic
for a message passing interface).

We state the following conjecture. *Any step-by-step numerical method can be represented, at
each iterative step, by a process-time diagram*. Some examples are provided in the next section.

A process-time diagram can express the degree of concurrency in a step-by-step method. This is,
in particular, useful for parallel and distributed codes. A step-by-step method has an inherent
degree of concurrency when it has a process-time diagram with \( p(n) \geq 2 \). Theoretically, the
diagram with the minimum height is the one who can lead to the best time reduction in a parallel
computing environment. In the case of two diagrams with the same height and same width, the
one with the minimum number of inter-process communication is preferred. One method can
have more than one process-time diagram. A parallel method, by definition, has some inherent
threads which can be performed simultaneously, i.e., its process-time diagram has at least two
columns. Same method can be implemented on a sequential computational environment: then,
its process-time diagram has only one column, i.e., one process. So, we have at least two diagrams
for each parallel method.

### 2.2. Examples

We consider three kinds of problems: recursive relations, finite-difference methods for ordinary
differential equations (ODEs), and volume element methods for parabolic partial differential
equations (PDEs). In each example, we use the same formula on the entire iterative process, i.e.,
\( p_i, p, q, s, r, R, S, I, m \) are independent of \( n \).

\[
\begin{align*}
\text{Process Time} & \quad \begin{array}{c}
\text{Process inter-communication} \\
0 & \quad 1 \\
1 & \quad 2 \\
3 & \quad 0
\end{array} \\
\text{Task which computes } I & \quad \text{Task inter-communication}
\end{align*}
\]

(a) (b) (c)

Figure 2. Process-time diagrams for the recurrence relation: (a) sequential implementation with one inter-thread inner-process communication; (b) sequential implementation with multiple inter-thread inner-process communications; (c) concurrent implementation with one inter-process communication.

Corresponding to the following recurrence relation,
\[
a_{n+2} = \begin{cases} 
3a_n & \text{if } n \text{ is even}, \\
\frac{(a_n + a_{n-1})}{2} & \text{if } n \text{ is odd},
\end{cases} 
\]
\[
a_0 = a_1 = 1, \quad 0 \leq n \leq M - 2,
\]
three process-time diagrams have been drawn in Figure 2. Since the recurrence formula is applied
to scalar values, \( m_{ij} \in \{0, 1, 2\} \) for any thread \((i, j)\). Table 1 shows the model’s parameters in
the three depicted cases.

Second, we consider three parallel Runge-Kutta methods for the numerical solution of ordinary
differential equations. For an initial value problem for ordinary differential equations, \( y'(t) = f(t, y(t)), \quad t \geq t_0, \quad y(t_0) = y_0 \), where \( y_0 \in R^M, f : R \times R^M \rightarrow R^M \), the general formula of a
\( w \)-stages Runge-Kutta method has the following form,
\[
y_{n+1} = y_n + h \sum_{i=1}^{w} b_i k_i, \quad k_i = f \left( t + c_i h, y_n + \sum_{j=1}^{w} a_{ij} k_j \right), \quad i \in \{1, 2, \ldots, w\}, \quad 0 \leq n \leq N.
\]
The problem dimension is $M$, the number of ordinary differential equations. At each step, $m_{ij}^{(n)} \geq M$. Each particular Runge-Kutta method can be described by Butcher table,

$$ c \mid A \mid b $$

where $c = (c_1, \ldots, c_w)^\top$, $A = (a_{ij})_{1 \leq i, j \leq w}$, $b = (b_1, \ldots, b_w)$.

In a parallel Runge-Kutta method, $A$ is a block-structured matrix. The specific matrices of the three particular methods considered here are presented in Figure 3.

**Figure 3.** Parallel implicit Runge-Kutta methods: (a), an $L$-stable fourth-order diagonally implicit Runge-Kutta (DIRK) method [3]; (b) and (c), the Hammer-Hollinworth $L$-stable fourth-order method [3], a block diagonally implicit method (BDIRK); (d), an $A$-stable fourth-order parallel singly diagonally iterated Runge-Kutta method (PDIRK) method [5].

The process-time diagrams of the three Runge-Kutta methods are presented in Figure 4. Table 2 specifies the main diagram parameters. Note that in the particular case of a parallel Runge-Kutta method the process-time diagram is similar to the processor-level description introduced in [3] (using a direct-graph, shortly digraph method). The difference between the two models can be exemplified using BDIRK method: in the digraph model different threads are constructed to evaluate $k_1$ and $k_2$ with some “transmissions” between them, but the task of solving
the nonlinear system in \(k_1, k_2\) cannot be decomposed into two independent threads. The digraph of the method is driven by the informational flow (or computational front) of an algorithmic step and can have cycles, meanwhile in a process-time diagram cycles are impossible.

The process-time diagram can be constructed also for other finite-difference methods designed for ordinary differential equations [7]. In the case of predictor-corrector methods for example, the closest model to the process-time diagram is that of the computational front broadening [6]. Each horizontal line from Figure 4 represents the front of computation which is defined in [6] as the imaginary straight line that separates the values which are next to be computed from all the previously computed values. The arrows indicate that calculations ahead the front depend on information on back side of the front.

The information flow graph was introduced in [7] to explain the techniques of parallelism across space (system) and across time (steps) in solving ordinary differential equations. In this case, the computational part of a thread is restricted to some vector operations and function evaluations. Cycles in such a graph are possible. Execution of a node consists of computing the output values when all the input values to that node have been computed. Parallel execution of a computation graph consists of assigning each node in the graph to a processor. One objective is to restructure the computation graph to have a large width so that there is a large degree of parallelism. From this point of view, our process-time diagram is the result of the restructuring operation, plus cycle elimination. In order to reduce the communications in our scheme, entire columns of threads are associated with processors.

Finally, we consider the particular case of a computational fluid dynamics problem. The governing equations which describe fluid flow are Navier-Stokes equations, continuity and additional conservation equations. The discretization procedure of the finite volume method [8] is well known from its application to fluid flows. Solution techniques that employ finite volume method require the generation of the solution grid that conforms to the geometry of the flow region (a grid of small volume elements for which the average values of flow quantities are stored). Therefore,
the choice of the grid is an important issue for the quality of the numerical simulations. In this connection, both the numerical resolution and the internal structure of the grid are very important. Computational method is an important issue which includes two parts. One is the solution procedure, often referred as outer iteration, and the other is the solver for the linear equations, referred as inner iterations [9]. The solution of large systems of linear equations uses most of computing time in PDE codes. Due to the nonlinearity and strong coupling of PDEs, outer iterations are used to update the coefficients and source terms in the linear equation systems. The number of required iterations increases with grid refinement. So, the efficiency of a PDE code strongly depends on the efficiency of the solver for the linear equation systems. A multiblock structured grid system [10] uses advanced matrix solvers for the inner iteration, and multigrid technique for the outer iterations. Stone's strongly implicit procedure (SIP, [11]), designed for banded matrices, is used here. The primary concern for any multiblock solution technique is transferring information through the block boundaries. A matched multiblock method is used in our simulation: the grid lines match each other at the block conjunction. The tests presented here are concerning the simulation of a silicon melt flow into a rotating crucible. The computational domain is subdivided into \( p \) blocks consisting of control volumes. The number of control volumes is depicted by \( M \). A process-time diagram can be construct with \( q = 1 \): a thread \((i,j)\), \( j \in \{1, \ldots , p\} \), is responsible for around \( m_{ij} \approx M/p \) control volumes, and for the SIP solution computation of the block's linear system, which is performed in \( t_{ij} = O(m_{ij}^{3/2}) \). The information exchange at the boundaries is performed in \( \sum_{k} R_{k}^{(k)} + \sum_{k} S_{k}^{(k)} = O(m_{ij}^{2/3}) \). Several iterative steps must include the overhead due to the outer iterations: at the thread level, this means \( A_{ij} \neq 0 \) for some \((i,j)\).

2.3. Speedup Measurement

In order to simplify the notations in the following definitions, we introduce some empty threads in the process-time diagram, so that at each level, we have exactly \( p^{(n)} \) threads. By definition, the empty thread \((i,j)\) has \( t_{ij} = r_{ij} = s_{ij} = S_{ij} = A_{ij} = 0 \), i.e., it does nothing.

Note that in the case of using some kind of parallel machines we are forced to introduce in the process-time diagram send-receive threads with \( t_{ij}^{(n)} = 0 \) (for example, when \( p^{(n)} = 3 \), one-to-all communication are requested, and we have a ring-type inter-connection between the \( p > p^{(n)} \) available physical processors).

The total time to perform the thread \((i,j)\) is the sum of the computation time, the overhead introduced by parallelization, and the communication time,

\[
T_{ij} = t_{ij} + A_{ij} + \sum_{k=1}^{r_{ij}} R_{ij}^{(k)} + \sum_{k=1}^{s_{ij}} S_{ij}^{(k)}.
\]

In the worst case (when the communications cannot be overlapped with computations) at the \( i \)th level of each step, the estimated level-time will be

\[
T_{i} = \max_{1 \leq j \leq p} T_{i,j},
\]

and the total estimated time spent at one iterative step will be

\[
T_{p} = \sum_{i=1}^{q} T_{i}.
\]

In the best case (no idle times at each processor, namely, an asynchronous model), the total estimated time spent at one step will be

\[
L_{p} = \max_{1 \leq j \leq p} \sum_{i=1}^{q} T_{i,j}.
\]
The total sequential time of one step of the same algorithm implemented in a sequential mode is

$$T_s = \sum_{i=1}^{q} \sum_{j=1}^{p} t_{ij}.$$  

For simplicity, we neglect the step index $n$; all the above time-values are dependent on $n$ and $M$.

We define the step-speedup as the following ratio depending on the step index, $n$,

$$S_p^{(n)} = \frac{T_s^{(n)}}{T_p^{(n)}}.$$  

The above ratio is upper-bounded by the optimistic value of the step-speedup,

$$S_o^{(n)} = \frac{T_s^{(n)}}{L_p^{(n)}}.$$  

We define the algorithmic-speedup as the following ratio,

$$S_p = \frac{\sum_{n=0}^{N} T_s^{(n)}}{\sum_{n=0}^{N} T_p^{(n)}},$$  

where $T_s^{(0)}$ and $T_p^{(0)}$ are the execution times necessary to prepare the iterative process ($T_s^{(0)} \leq T_p^{(0)}$). The speedup values are dependent on $M$. The algorithmic-speedup is upper-bounded by the optimistic algorithmic-speedup,

$$S_o = \frac{\sum_{n=0}^{N} T_s^{(n)}}{\sum_{n=0}^{N} L_p^{(n)}},$$  

Generally, the theoretical speedup is defined relative to the execution time $T_0$ of the "best" sequential algorithm which solves the problem in a sequential manner ("best" in the case of an ODE and PDE integration is ambiguous). The limits corresponding to the worst and best cases are

$$S_p = T_0 \left/ \sum_{n=0}^{N} T_p^{(n)} \right., \quad S_o = T_0 \left/ \sum_{n=0}^{N} L_p^{(n)} \right..$$  

For a specific implementation of the iterative method (sequential, concurrent, parallel, or distributed), we can measure the time spent to solve a problem and the time spent in one iterative step. The measured code speedup of a multiprocessor version is denoted by $S_p$, while the measured step-speedup is denoted by $S_p^{(n)}$.

### 2.4. Speedup Limits

We expect that

$$S_p^{(n)} \leq S_p^{(n)} \leq S_o^{(n)}, \quad S_p \leq S_p \leq S_o.$$  

Since $L_p^{(n)} \leq T_p^{(n)}$, $S_p \leq S_o$. Moreover, $S_p \leq S_p$, i.e., the worst value of the speedup is under the value of the algorithmic-speedup. If the last equality holds, the parallel algorithm is said to be cost-optimal (the "best" sequential method has been detected in this case).

Between the algorithmic-speedup and the step-speedup, we can establish the following inequalities which hold for all $M$-values,

$$\min_{0 \leq n \leq N} S_p^{(n)} \leq S_p \leq \max_{0 \leq n \leq N} S_p^{(n)}, \quad \min_{0 \leq n \leq N} S_o^{(n)} \leq S_o \leq \max_{0 \leq n \leq N} S_o^{(n)},$$  

where $S_p$ and $S_o$ are the parallel and algorithmic speedups, respectively.
i.e., the boundaries of the step-speedup give an estimation of the algorithmic-speedup. Equalities are possible when the step-speedup is constant (do not depends on the step number). Moreover, if $T_s^{(n)} = p^{(n)} T_p^{(0)}$, then

$$S_p \geq \frac{\sum_{n=1}^{N} T_s^{(n)}}{\sum_{n=1}^{N} T_p^{(n)}} \geq \min_{1 \leq n \leq N} S_p^{(n)}.$$

As upper bounds for the step-speedup, we have

$$S_p^{(n)} \leq \frac{p^{(n)}}{\sum_{i=1}^{q^{(n)}} \max_{1 \leq j \leq p^{(n)}} t_{ij}^{(n)} + \sum_{k=1}^{r_{ij}^{(n)}} R_{ij}^{(k,n)} + \sum_{k=1}^{s_{ij}^{(n)}} S_{ij}^{(k,n)} + A_{ij}^{(n)}} \leq p^{(n)}.$$

Hence, the step-speedup at the $n$th step attains its ideal value only when the following three conditions are fulfilled.

- Overhead: $A^{(n)}_{ij} = 0$, i.e., the thread starting and stopping time is null.
- Communications: $S_{ij}^{(n)} = 0$, i.e., no message will be send, meaning that all information necessary for computation are local, or $S_{ij}^{(k,n)} = 0$, for all $k$, which is possible only on an ideal parallel computer. Similarly, $r_{ij}^{(n)} = 0$ or $R_{ij}^{(k,n)} = 0$, for all $k$.
- Load balance: $t_{ij}^{(n)} = \max_{1 \leq j \leq p^{(n)}} t_{ij}^{(n)}$, for all $i$ and $j$, i.e., the work is load balanced. All the concurrent processes of a level must have the same sequential complexity, null-process with $t_{ij}^{(n)} = 0$ are not admitted, and a constant number of processes will be used at all stages of the $n$th step.

So, the ideal value of the step-speedup is achieved when a number $p^{(n)}$ of processes of equal sequential complexity use only local data for computation, do not cooperate with each other (independent processes, i.e., embarrassing parallelism), and, after $q^{(n)}$ stages, produce $p^{(n)}$ solution values which will be used at the next integration step.

Consequently,

$$S_p \leq \max_{0 \leq n \leq N} p^{(n)}$$

and the above remarks are valid also for algorithmic-speedup.

Amdahal's law [12] about the limitation of the speedup says that the speedup of a parallel algorithm is always limited by the serial fraction of the parallel algorithm no matter how many processors are used. We apply Amdahal's law in the case of the step-speedup. If $\alpha^{(n)}$ is the serial fraction of the $n$th step of an algorithm, then $S^{(n)} \leq 1/\alpha^{(n)}$. If $\alpha$ is independent on $n$ and $M$, the serial fraction will limit also the algorithmic-speedup.

2.5. Algorithm Implementation Issues

In the case when the thread $(i,j)$ will be implemented as a machine process, then $A_{ij}^{(n)} > 0$. $A_{ij}^{(n)} = 0$ if all threads $(i,j)$, $1 \leq i \leq q^{(n)}$ (from a diagram column $j$) will be implemented as parts of a unique process $j$ which will be started at step $0$ (starting time will be included in $T_p^{(0)}$).

We suppose in what follows that this is the case.

The value $L_p^{(n)}$ is the lowest value that we can get as step's time. Unfortunately, it is very improbable to attain $L_p^{(n)}$ since it excludes the waiting times. Moreover, due to the computational system delays, we can expect that the theoretical highest value $T_s^{(n)}$ will be sometimes over-passed.

The communication time is the main impediment in attaining the ideal speedup. Note that $r_{ij}^{(n)}$ can be reduced only by changing the method or by changing the task identifiers to attain the minimum inter-process communications.
$S_{ij}^{(k,n)}$ can be reduced by overlapping the send operations with the same message (broadcasting technique replacing a send-receive with a send-to-all operation).

Overlapping communications with receive operations can improve also the step-speedup. This is possible if there is at least one tuple $(i, j, k)$, such that $T_{i,j}^{(c,j_2)} < \max_{1 \leq j \leq n} T_{i,j}^{(k,n)}$ and the $k^{th}$ message is waiting to be received by thread $(i, j)$. This is more probable to happen if the messages are not received in a established order and/or the $k^{th}$ message has come from a thread of the $(i-d)^{th}$ level where $d > 1$, or from some previous steps. The value $L_p^{(n)}$ is the lowest value that we can get as parallel step time by overlapping communications and computations.

Suppose that the $k_1^{th}$ message from the thread $(i_1, j_1)$ is received at the step $n_1$ is received as the $k_2^{th}$ message by the thread $(i_2, j_2)$ at the step $n_2$. According the linear model of inter-processor communication \cite{12}, $S_{ij}^{(k_1,n_1)} + R_{ij}^{(k_2,n_2)} = l/w + o = O(l)$ where $l$ is the length in computer-words of the message, $w$ is the channel bandwidth in words per second, and $o$ is the time required to handle a message, including the time to prepare the message, the time to execute the routing algorithm, the time to establish an interface between the local processor and router, and the time to interpret the message at the receiving processor. Theoretically, for large $l$, $o$ is not significant, and for small $l$, $o$ can dominate the communication time.

The execution times $t_{ij}^{(n)}$ can be different not only due to sequential complexity of the threads but also from implementation motives. In a heterogeneous computing environment, if we know that a particular column $j$ of the diagram will be the most charged one, we can ask to run the corresponding threads on a faster processor than the ones allocated to other threads. User-independent motives must be also taken into account: if the computation is done of workstations some threads on a processor can be delayed due to their concurrence with other threads running on the same processor (in this case also some delays in the communication times can be possible).

### 2.6. Case Studies

Theoretically, an implementation of parallel recurrence formula on a parallel or distributed environment must lead to a execution time reduction (concurrent version versus sequential one). For our example, unfortunately, in practice, on a message passing computational environment, $\sum_k S_{ij}^{(k)} + \sum_k R_{ij}^{(k)} > t_{ij}$ for any particular thread $(i, j)$; so, any concurrent code implementing the recurrence formula and using message passing interfaces is inefficient.

In the particular case of an $s$-stage parallel Runge-Kutta method, the serial fraction of the $n^{th}$ step refers especially to the last stage when the solution approximation is evaluated (no concurrent threads in the last level of any diagram from Figure 4). Normally, $\alpha^{(n)} \rightarrow 0$ when $M \rightarrow \infty$, but there are some special cases when $\alpha^{(n)}$ does not decrease. For example, when we use explicit Runge-Kutta methods and we integrate sparse linear IVPs, $\alpha^{(n)}$ will be approximately constant and the speedup is limited.

Parallelism across method, which is a problem independent technique for ordinary differential equations, is characterized by a limited number of processors used, and therefore, a limited speedup. If we use a fixed method, $p^{(n)}$ is constant and $T^{(n)}_{(i,j)} \approx T^{(q)}_{(i,j)}$, for all $q > n$ (small difference are due to the solving procedure of nonlinear equations, the changes in the sparsity structure of the involved matrices, or the delays in communication networks), i.e., the algorithmic-speedup can be approximate by the constant step-speedup. We will use this remark in our numerical tests. If we use a variable-order integration scheme, it is possible to increase the algorithmic speedup over the bound given by the step-speedup corresponding to the first steps (this can be the case when a stiff IVP integration is performed). If we solve a linear IVP, with diagonal matrix, banded matrix, or block matrix (band and block size do not increase with $M$), at least for explicit methods, $t_{ij}^{(n)} = O(M)$. Since communication time is also $O(M)$, the step-speedup $S^{(n)}$ will be limited under the ideal value. In other cases, $t_{ij}^{(n)} = O(M^d)$ much greater than the communication time when $d \geq 2$. Therefore, when we use an implicit method to solve
a nonlinear IVP, requiring the solution of some nonlinear equations at each integration step, it is probable to get a speedup value close to the ideal one.

A parallel algorithm is scalable if the efficiency \( (S_p / \max_{1 \leq n \leq N} p^{(N)}) \) can be held constant linearly as the number of processors increases by increasing the problem size. This goal cannot be attained when we use an iterative scheme based on a fixed formulae (in the recurrence or ordinary differential equation cases). In the PDE case, it can be found a formula for \( M \) on \( p \), so that the efficiency remains constant when \( p \) increases.

3. NUMERICAL TESTS

Our objective is to find to what extend the performance model is validated by practice. Here, we will focus on the three parallel Runge-Kutta method and the finite-volume method.

The computational environment consists of a cluster of PCs with Pentium III at 650 MHz, 1 Gb RAM per node, connected by 3 x Fast Ethernet (100 M bit/s switched), and running Linux.

We have used our own software applications, EPODE (Expert system for ODEs [1]) with support for parallel and distributed computation based on PVM, and the parallel version of STHAMAS 3D for flow simulation [2], with support for parallel and distributed computation based on MPI.

The test initial value problem is obtained by applying the method of line to the two-dimensional Brusselator's problem, a model of a chemical reaction-diffusion process,

\[
\begin{align*}
\frac{d}{dt} u_1(t,x) &= - u_1^2(t,x) u_2(t,x) + a(t) - (b(t) + 1) u_1(t,x) + \alpha_1 \Delta u_1(x,t), \\
\frac{d}{dt} u_2(t,x) &= b(t) u_1(t,x) - u_1^2(t,x) u_2(t,x) + \alpha_2 \Delta u_2(t,x),
\end{align*}
\]

with \( x \in D = [0,1]^2 \), \( t \in E = [0,1] \), the boundary conditions \( u_1(t,x) = a(t), u_2(t,x) = b(t)/a(t) \), for \( x \in \partial D \), \( t \in E \) and the initial conditions \( u_1(0,x) = c(x), u_2(0,x) = b(0)/a(0) \), for \( x \in D \).

The spatial derivatives are discretized on a grid in \( D \). The number \( M \) of ordinary differential equations increases with the number of grid points. For small \( M \) values, the communication time is significant. Increasing \( M \) leads to an improvement of the speedup.

Table 3 presents the values of the step-speedup and algorithmic-speedup measured at code execution. Note that the code speedup is almost the same as the step-speedup.

The test time-dependent partial differential equation is concerning the problem of a silicon melt flow into a rotating small crucible. The melt flow is governed by the three-dimensional equations describing mass, momentum, and heat transport. In a three-dimensional Cartesian coordinate system \( (i = 1, 2, 3) \), we have the following set of partial differential equations [2],

\[
\begin{align*}
(pu_i)_i &= 0, \\
(\rho T)_t &= - (pu_jT - T \nabla T)_j, \\
(pu_i)_t &= - (pu_j u_j - \mu ((u_i)_j + (u_j)_i))_j - \rho u_i - \rho(T_{ref}) g_i \beta (T - T_{ref}).
\end{align*}
\]

Table 3. Speedup measured in the case of the discretized two-dimensional Brusselator problem.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
Method & \( M \) & Ideal & \multicolumn{4}{|c|}{\( S^{(20)} \)} & \multicolumn{4}{|c|}{\( S \)} \\
\hline
& & Speedup & 20 & 40 & 80 & 160 & 320 & 20 & 40 & 80 & 160 & 320 \\
\hline
DIRK & 2 & 1.58 & 1.80 & 1.87 & 1.90 & 1.94 & 1.69 & 1.80 & 1.89 & 1.92 & 1.94 \\
\hline
BDIRK & 2 & 1.76 & 1.86 & 1.91 & 1.94 & 1.97 & 1.75 & 1.87 & 1.93 & 1.94 & 1.96 \\
\hline
PSBDIRK & 3 & 1.89 & 2.02 & 2.34 & 2.56 & 2.87 & 1.92 & 2.12 & 2.30 & 2.56 & 2.85 \\
\hline
\end{tabular}
\end{table}

Table 4. Speedup measured in the case of the melt flow simulation.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\( p \) & \( M/10^3 \) & \multicolumn{4}{|c|}{\( S^{(20)} \)} & \multicolumn{4}{|c|}{\( S \)} \\
\hline
& & 25 & 75 & 150 & 450 & 1100 & 25 & 75 & 150 & 450 & 1100 \\
\hline
2 & 1.45 & 1.70 & 1.79 & 1.90 & 1.91 & 1.36 & 1.58 & 1.64 & 1.70 & 1.70 \\
\hline
3 & 1.57 & 1.95 & 2.13 & 2.31 & 2.37 & 1.47 & 1.81 & 2.00 & 2.10 & 2.15 \\
\hline
6 & 1.92 & 2.90 & 3.39 & 4.09 & 4.30 & 1.80 & 2.75 & 3.20 & 3.87 & 4.02 \\
\hline
\end{tabular}
\end{table}
The computational domain (the crucible) is subdivided into 6 blocks consisting of control volumes. An important issue for the quality of the numerical simulations is the choice of the grid. Several grid levels were used in simulations, from a coarse grid with around $M = 20,000$ control volumes to a fine grid with around $M = 1,000,000$ control volumes.

Table 4 presents the values of the step-speedup and algorithmic-speedup, measured at code execution. The algorithmic-speedup is lower than the step-speedup due to the outer iterations and the serial fraction: in order to start the iterations, the geometry of the control volumes must be computed and this sequential process needs around 10% of the code (serial fraction). Remeasuring the algorithmic speedup without this 10%, we get speedup values close to the step-speedup ones.

4. CONCLUSIONS

In a message-passing system, the code speedup of a concurrent solver of a numerical problem via an iterative method is mainly influenced by the problem dimension and the management of the communications. The code speedup can be estimated knowing the speedup of the code in some iterative steps. This statement was proved by numerical experiments performed with a solver for ordinary differential equations and one for partial differential equations, both allowing parallel and distributed computations.

REFERENCES