Journal of Pure and Applied Mathematics: Advances and Applications
Volume 1, Number 2, 2009, Pages 129-141

PARALLEL ALGORITHMS FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS

CRESCENZIO GALLO

Dipartimento Di Scienze Economiche
Matematiche E Statistiche Centro Di Ricerca
Interdipartimentale Bioagromed
Università di Foggia-Largo Papa
Giovanni Paolo II n.1, 71100 Foggia
Italy
e-mail: c.gallo@unifg.it

Abstract

The availability of high-performance computing tools gives the opportunity of solving mathematical representations of complex systems made through Ordinary Differential Equations (ODEs). This paper illustrates some parallel algorithms suitable for the solution of ODEs, inside an abstract simulator architecture aimed to the modeling of dynamical systems.

1. Introduction

1.1. Topics

The explosion of computing technology today allows to face with the increasing complexity of the problems to be solved, typically through modeling, for its ability to simplify problems by eliminating unnecessary details. Computers can now deal with otherwise unsolvable problems, thanks to modeling software that allows the discover of meanings hidden

2000 Mathematics Subject Classification: 93A30, 37-02, 34A34.

Keywords and phrases: modeling, dynamical systems, ordinary differential equation, algorithm.

Received January 4, 2009

© 2009 Scientific Advances Publishers

by the vast amount of data to be collected during the observation of a system.

Systems are frequently modeled by ordinary differential equations (ODEs) [7], which have been used in different ways to describe a wide range of real world systems. This work is focused on the solution of such models, i.e., on the computational solution of ODEs, in particular through parallel algorithms. The computing tool may so be useful in two ways: firstly, it can provide the means for resolving analytically intractable equations; secondly, it can provide a sufficient computing power to use otherwise unworkable techniques.

1.2. Distributed systems

Probably the most important ongoing development in computer architecture is the distributed computing. Distributed computing systems are the only architecture that has the potential to meet simultaneously all major desirable needs for a computer system, i.e.,

- high performance through parallel processing (using low cost LSI components);
- modular expandability of the system;
- system fault tolerance;
- increased economy by software simplification.

A *distributed system* is a computing system made of several processing nodes which satisfies the following conditions:

- the general system functions are provided through the cooperation of various processes;
- there are no two processes having the same vision of the global state of the system;
- there is no single system process that can ensure that the other processes have a consistent and identical “understanding” of the overall state of the system.

From this definition, it is clear that it is above all the control of the system to be distributed throughout the system itself. In addition, system

processing nodes can be distributed spatially or geographically in a building, in a complex of buildings, in a region, nation or even in different continents. Such distribution implies that such "loosely" coupled nodes can only communicate through message exchange. It must be said that there is also a great interest in the multicore and multiprocessor systems, in which processors (and cores) are not dispersed but are concentrated locally, where control is nevertheless distributed (decentralized).

Many of the principles and techniques developed for the organization of distributed systems can be used profitably for the organization of localized systems; distributed systems, moreover, once reached a certain maturity, may become the preferred architectural solution for a wide range of computing applications.

2. Hierarchies of Systems and Models

A system is a set of subsystems interacting with each other: this definition is universally accepted, and stresses the possibility of usefully breaking a system into its components.

These subsystems typically behave in a parallel manner. All system variables are a function of time, and are constantly exchanged between the subsystems in parallel; it is therefore necessary for a data processing system (to be a "true" simulator) to approximate these important features typical of the nature of parallelism. We shall now see the "hierarchical" nature of systems and models, both from a space and time point of view.

2.1. Spatial hierarchies

A system belongs to a certain region of space. For the temporal behavior of a system, the spatial dependence is not embedded in usual definitions, but for systems broken into subsystems operating simultaneously or in parallel there is some ordering of these subsystems into space. Parallel systems operate in a spatial structure; the direct or indirect interaction between subsystems is expressed by the interconnection matrix (see Figure 1).

Through subsequent breakdowns a system (named the *top* system) is ordered in a hierarchy of *intermediate* and/or *bottom* systems in space. A

breakdown rule for which a system (and each subsystem) is split into k subsystems gives a *recursive spatial hierarchy of order k*, *complete* if for each level all the subsystems are broken.

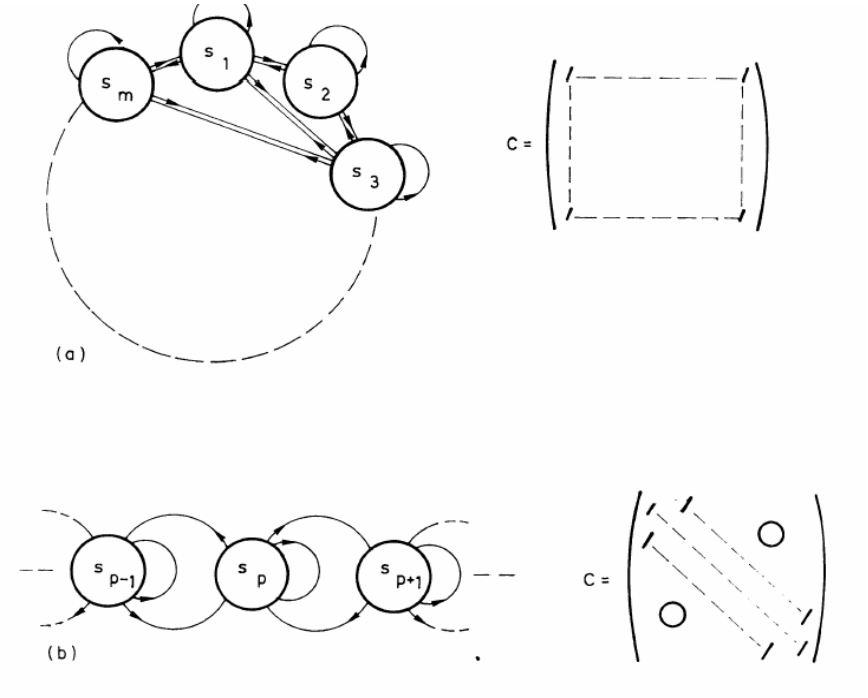

Figure 1. Interconnection matrix C for a system with m subsystems, where (a) all subsystems are mutually interconnected, and (b) only neighbor subsystems are interconnected.

2.2. Time hierarchies

The input-output relationship of a system can be constant or variable over time, and this dependency can occur in several ways: or the structure of an interconnected system remains constant but the intensity of existing interactions changes over time, or the interconnection structure varies over time. In the latter case, at a certain time only one of the potential subsystems may be an effective subsystem, but over time many of the potential subsystems can become effective. Of course, there are causal "decision" rules to make the transition from the actual subsystem to his successor at the right time.

3. Simulators and Related Principles

A necessary condition for the simulation of a system is the *one-to-one correspondence in space and time* between the system and its simulation (model). This determines the architecture of a parallel processor, in order to implement as much as possible adherently the spatial and temporal hierarchies present in the model of the system, but very often it is impossible to implement a model on a system developed in such a way as to preserve the structure of the model in space and time.

A conventional digital computer can handle a spatial hierarchy transforming it into a time-ordered hierarchy, having reshaped the spatial hierarchy in a hierarchy with dimension of Level 1. This leads to the fact that even small changes in the description of the system may require a significant computational modeling overhead.

In the next section are discussed some important aspects of a parallel processor useful to solve differential equations, leaving out the details of hardware/software (for a more complete overview see [2, 3, 5, 8]).

4. Architecture of the Proposed Parallel Processor

Because of the need for a modular architecture with great regularity for the hardware and software, only recursive spatial hierarchies of data processing modules can be considered relevant for a parallel data processor.

When the subsequent breakdown turns into models of lower systems under several levels of decomposition, transformation into a fully recursive spatial hierarchy will result in a high percentage of models of unused subsystems, as well as an increase in the percentage of dummy elements, both introduced to meet regularity in breaking rules.

Similarly, because of demand for modular architecture, the time-ordered hierarchy to be incorporated into a parallel data processor must be recursive. The transformation in a recursive time-ordered hierarchy is only possible if a time-ordered hierarchy of a system model has a maximum number of possible successors less than or equal to the number of possible transitions in the recursive time-ordered hierarchy.

In this respect, it seems acceptable to restrict the class of time-ordered hierarchies implemented in a transparent manner by limiting the number of different definitions of system model as well as the number of possible successors for each transition level.

As for the implementability of spatial hierarchies, it is necessary to analyze the time-ordered hierarchies of models of practical systems in order to design the implementability in greater detail.

4.1. Interconnections

As already mentioned, for large systems and models the interconnection matrix is sparse, so the reduction of interconnectability is permissible in the spatial hierarchy of the data processor. Interconnectability reduction is considered in Figure 2 in the case of a parallel data processor with a spatial ternary truncated hierarchy.

There are systems that show a structure of interaction that does not allow the reduction of interconnectability in its spatial hierarchy, if it is necessary to preserve the one-to-one analogy in space. The simulation of these systems by the only spatial extent will not be possible. Simulator's advanced sequential processing power must be used by allowing the efficient extension in time and applying parallel overlays, for example, in iterative blocks.

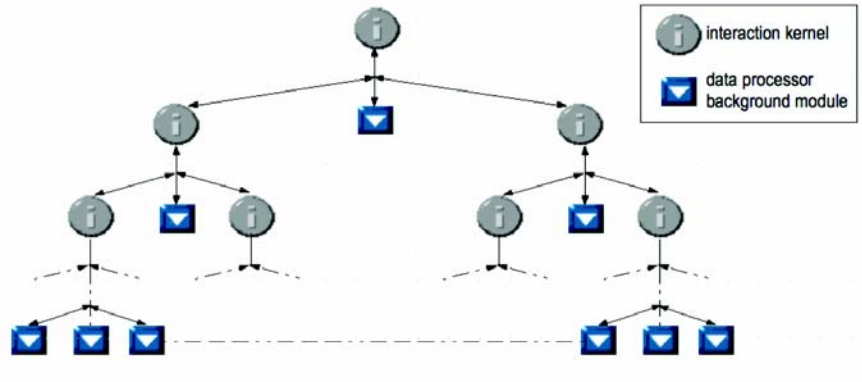

Figure 2. Spatial interconnection structure of a parallel data processor (where interaction kernel is an algorithm for $\underline{z}_k = \underline{g}(\underline{v}_k)$ and data processor b. m. is an algorithm for $\underline{z}_{k+1} = \underline{z}_k + \underline{\alpha g}\ (\underline{v}_k, u_k, \underline{z}_k, \ldots, \underline{z}_{k-r})$).

4.2. Parallel algorithms

The design of a data parallel processor may be based primarily on the methodology in terms of algorithms and methods of data processing, where parallel data processing is relative to the parallel management of continuously conflicting tasks.

In each form of data processing at all levels in Figure 2 a certain application has to be implemented using compound algorithms designed through the principle of extension in space and time and taken from the basic algorithms operating in discrete-time set S_p (with processing cycle time θ_p). A basic algorithm is pre-programmed in a spatial configuration of arithmetic components operating in parallel in time across the time-discrete set S_p (with basic cycle time $\theta_b \ll \theta_p$) and is then named *parallel algorithm*.

Consider an example of the use of a parallel algorithm in sequential advanced data processing: a multi-point compound algorithm for a set of n homogeneous differential equations of the first order

$$\frac{d(\underline{y}(u))}{du} = \underline{g}(\underline{y}(u)), \quad u_0 < u < u_1; \ \underline{y}(u_0) = \underline{y}_0, \tag{1}$$

where multi-point means that a point algorithm (application in a point of the space of arguments) must be run for a sequence of values of the input variable. The applications of parallel intervals continuously interacting (such as those representing the n time-continuous models of first order in (1)) and these algorithms are said to be *applications of true parallel intervals* and *true parallel algorithms*.

The recursive interval application (1) is achieved by a multipoint algorithm $y_k = G(y_{k-1})$, $k = 1, 2, \ldots$ with $u_k - u_{k-1} = \theta$. This "c" algorithm is of compound type, and is made of n algorithms c_j, $j = 1, \ldots, n$. The c_j parallel algorithms are made of arithmetic components of type "delay", with delay θ_b, implying that the parallel algorithm c_j is a delay algorithm with delay θ_j, $\theta_j \gg \theta_b$. Delay algorithms are algorithms that evaluate a certain function with a delay

between the input and output that depends on the complexity of the function; therefore the value of θ_j depends on the function g_j.

Even algorithm "*c*" becomes a delay algorithm, with delay θ, $\theta \gg \theta_b$; so, it is not a true parallel algorithm in S_b, but a true parallel algorithm in S_a, $S_a = \{u_k\}$. The values of θ and $\theta_j (j = 1, \ldots, n)$ satisfy the inequality:

$$u \geq \max_j \theta_j . \tag{2}$$

The implementation of the interval application (1) in this way, as a parallel delay algorithm is very fruitful only in the case of a balanced complexity: in the best case, the spatial extension of the delay algorithm "*a*" in n parallel delay algorithms c_j gives an increase in processing speed by a factor n. In the worst case, the improvement is almost imperceptible: then, the implementation of (1) as a sequential delay algorithm is a good alternative.

A better accuracy is achieved when, instead of parallel delay algorithms are adopted *true parallel algorithms* in discrete time, allowing the exchange of data "useful" for accuracy in all discrete moments of basic time set S_b.

Consider the design in greater detail of true parallel algorithms. An application $z = g(v)$ that cannot be computed at once, can be approximated by $z_{k+1} = g(v_k)$. The value of the function at time u_k can be approximated by an application g^*, a function of a finite number of previous argument values:

$$z_k \approx g^*(v_{k-1}, v_{k-2}, \ldots, v_{k-L-1}),$$

where:

- the previous argument values have a decreasing influence on the value of function when time increases;
- for a constant argument, g^* will be equal to g.

Suppose now that the application g^* can be split into simple applications g_i (generable within a time θ_b) that can be hierarchically nested:

$$z_k = g_0(v_{k-1}, g_1(v_{k-2}, g_2(v_{k-3}, \ldots, g_L(v_{k-L-1})\ldots), u_k \in S_b, k \geq L+1, (3)$$

where $g_j, j = 0, \ldots, L$ represents the application of the algorithm as implemented in block j, and where u_0 is the initial time. If time indexes in equation (3) are all equal to $k - L - 1$, we have the classic "sequential *pipeline*". If indexes are different, we get a "hybrid[1] *pipeline*". The output of a hybrid pipeline has a delay only equal to θ_b. The initial value of a hybrid pipeline algorithm is a matrix $n \times L$ for $v \in \mathcal{R}^n$.

The generation of trigonometric functions, for example, can be performed using

- a sequential pipeline algorithm:

$(\sin v)_{k+5} =$

$$\left(v_k \cdot \left(\frac{1}{1!} + v_k^2\left(-\frac{1}{3!} + v_k^2\left(\frac{1}{5!} + v_k^2\left(-\frac{1}{7!} + v_k^2 \cdot \frac{1}{9!}\right)\right)\right)\right)\right);$$

- a hybrid pipeline algorithm:

$(\sin v)_{k+1} =$

$$\left(v_k \cdot \left(\frac{1}{1!} + v_{k-2}^2\left(-\frac{1}{3!} + v_{k-3}^2\left(\frac{1}{5!} + v_{k-4}^2\left(-\frac{1}{7!} + v_{k-5}^2 \cdot \frac{1}{9!}\right)\right)\right)\right)\right).$$

In this case, all pipeline segments, except the last, must perform similar arithmetic operations. In [1], several examples are discussed of hybrid pipeline algorithms. Note that in some cases, the pipeline may become too long: in such cases, we must take a combination of methods of table consulting and hybrid pipeline arithmetic [2].

---

[1] The term *hybrid* is due to the fact that blocks do not only operate sequentially in space but in parallel too, with respect to the input variable.

The design of a compound algorithm often includes one or more closed *loops*. Some stability studies [4, 6] show that in a parallel data processor compound algorithms should be implemented in different time sets S_{pj}, $j = 1, 2, \ldots$ when true parallel algorithms are used. So some input and output buffers are needed in arithmetic components in order to transform these different time sets into the data transfer time set S_d $(\text{with } \theta_d \leq \min \theta_{pj})$.

True parallel algorithms' stability conditions do not only depend on the mathematical properties of the topology of the spatial structure of their arithmetic components operating in parallel, but also on the stability intervals of such components. In a such way, the stability interval of the linear open-loop hybrid pipeline can be evaluated:

$$z_k = \alpha_1 v_{k-1} + \alpha_2 v_{k-2} + \cdots + \alpha_m v_{k-m} \qquad \text{with} \sum_{j=1}^{m} \alpha_j = 1,$$

making use of the transform analysis, and the results for the particular case $\alpha_j = \dfrac{1}{m}$ for $j = 1, \ldots, m$ in $-m <$ feedback gain < 1.

If you use the internal feedback from the output of a hybrid pipeline to the inputs of all blocks, you obtain a closed-loop hybrid pipeline algorithm, described by:

$$z_k = g_0(v_{k-1}, z_{k-1}, g_1(v_{k-2}, z_{k-2}, g_2(v_{k-3}, z_{k-3}, \ldots, g_L(v_{k-L-1}, z_{k-L-1}) \ldots). \tag{4}$$

A particular case of the linearized version $z_k = \sum_{j=1}^{m} (\beta_j z_{k-j} + \alpha_j v_{k-j})$, the true parallel algorithm (named single *step* algorithm) $z_k = z_{k-1} + \dfrac{\theta_b}{\theta}(v_{k-1} - z_{k-1})$ shows a stability interval $-\dfrac{2\theta}{\theta_b} + 1 <$ feedback gain < 1. Thus a large stability interval requires $\theta \gg \theta_b$.

Note that the single step algorithm is a discrete version of the unconditionally stable time-continuous algorithm $\theta \frac{dz(u)}{du} + z(u) = v(u)$. For $v(u)$ constant both algorithms have the same equilibrium state.

A final consideration concerns the integration of ODEs. There are several ways to perform the numerical integration of:

$$\frac{d\underline{y}(u)}{du} = \underline{g}(\underline{y}(u), \underline{v}(u)),\ u \in [u_0, u_e);\ \underline{y}(u_0) = \underline{y}_0,$$

but almost all are unsuitable for parallel data processing. The application of methods with local truncation errors of order higher than $O(\vdots \Delta \vdots u_k)^2$ is unnecessary [4]. When the integration is achieved through discrete-time integration algorithms continuous in S_b, the Euler method (first order) is attractive. An important objection to this method is that the magnitude of the step must be pretty low, which increases the accumulation of rounding errors.

From Euler's numerical integration method $\underline{y}_k = \underline{y}_{k-1} + \Delta u_k \underline{g}(\underline{y}_{k-1}, \underline{v}_{k-1})$ an attractive parallel algorithm can be derived. With the smallest Δu_k equal to θ_b :

$$\underline{y}_k = \underline{y}_{k-1} + \theta_b \underline{w}_{k-1},$$

$$\underline{w}_k = \underline{g}(\underline{y}_{k-1}, \underline{v}_{k-1}). \tag{5}$$

As arithmetic components of the parallel algorithm, we need a “linear combination” delay algorithm of size two and a “function generating” algorithm. The function is typically too complex to be implemented in θ_b; quasi-time-continuous table consulting algorithms need to be applied [2]. The data transfer between arithmetic components requires that all data are transferred in the same word length; then, it is necessary to round off each output variable before the transfer. On the other hand, in true parallel data processing, due to the reduced time step θ_b, we must take

account of the term $\theta_b \underline{w}_{k-1}$ in such a way that the cumulative result is in agreement with the required accuracy; so in the next processing cycle, a *rounding correction* has to be introduced. With

$$\underline{y}_k = \underline{y}_{l,k} + \underline{y}_{r,k}, \tag{6}$$

where $\underline{y}_{l,k}$ is the rounding value of $\underline{y}_k$ and $\underline{y}_{r,k}$ is the local rounding error, the rounding correction applied to (5) results in:

$$\underline{y}_k = \underline{y}_{k-1} + \underline{y}_{r,k-1} + \theta_b \underline{w}_{k-1}, \tag{7}$$

$$\underline{w}_k = \underline{g}(\underline{y}_{l,k-1}, \underline{v}_{k-1}).$$

In a parallel data processor, all arithmetic components must be equipped internally with such rounding procedure.

5. Conclusions

In this work, we analyze a possible parallel computing architecture for the solution of ordinary differential equations, both in localized and distributed environments, aiming to develop concepts for the construction of a top layer of software for modeling and simulation (in its restricted meaning of model execution).

We also present some experimental results with the aim of providing the modeling process with better computing approaches. A series of research hints are given, to further guide future research.

References

[1] J. H. M. Andriessen, Discrete-time parallel algorithms with continuous-time response, Proc. Simulation 80, Interlaken, Acta Press (1980).

[2] L. Dekker, E. J. H. Kerckhoffs, G. C. Vansteenkiste and J. C. Zuidervaart, Outline of a future parallel simulator, Proc. of the IMACS Congress on Simulation of Systems, Sorrento, North Holland Publishing Co. (1979), 837-864.

[3] P. M. Dickens, A workstation-based parallel direct-execution simulator, PADS'97 Proceedings of the eleventh workshop on Parallel and distributed simulation (1997).

[4] European Simulation Meeting Algorithms in parallel data processing and simulation, Delft, Holland (1979).

[5] S. L. Ferenci, K. S. Perumalla and R. M. Fujimoto, An approach for federating parallel simulators. PADS'00, Proceedings of the fourteenth workshop on Parallel and distributed simulation, IEEE Computer Society (2000).

[6] K. A. Frenkel, Special issue on parallelism, Communications of the ACM 29(12) (1986).

[7] L. Ince, Ordinary Differential Equations. Dover, New York (1956).

[8] D. Nicol and P. Heidelberger, Parallel execution for serial simulators, Transactions on Modeling and Computer Simulation (TOMACS) 6(3) (1996).

■