Exploiting the Multilevel Parallelism and the Problem Structure in the Numerical Solution of Stiff ODEs

J.M. Mantas Ruiz
Software Engineering Dept.
Univ. Granada
Avda. Andalucía 38,
18071 Granada, Spain
jmmantas@ugr.es

J. Ortega Lopera
Dpto. Arquitectura y Tecnología
de Computadores. Univ. Granada
Avda. Andalucía 38,
18071 Granada, Spain
jortega@atc.ugr.es

J. A. Carrillo
Dpto. Matemática Aplicada
Univ. Granada
Avda. Severo Ochoa s/n,
18071 Granada, Spain
carrillo@ugr.es

Abstract

A component-based methodology to derive parallel stiff Ordinary Differential Equation (ODE) solvers for multicomputers is presented. The methodology allows the exploitation of the multilevel parallelism of this kind of numerical algorithms and the particular structure of ODE systems by using parallel linear algebra modules. The approach furthers the reusability of the design specifications and a clear structuring of the derivation process. Two types of components are defined to enable the separate treatment of different aspects during the derivation of a parallel stiff ODE solver. The approach has been applied to the implementation of an advanced numerical stiff ODE solver on a PC cluster. Following the approach, the parallel numerical scheme has been optimized and adapted to the solution of two modelling problems which involve stiff ODE systems with dense and narrow banded structures respectively. Numerical experiments have been performed to compare the solver with the state-of-the-art sequential stiff ODE solver. The results show that the parallel solver performs specially well with dense ODE systems and reasonably well with narrow banded systems.

1. Introduction

One important formulation for the differential equations arising from the modelling process is that of the *Initial Value Problem* (IVP) for ODE [2]. The goal in the IVP is to find a function $y : \mathbb{R} \rightarrow \mathbb{R}^d$ given its value at an initial time t_0 and a recipe $f : \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}^d$ for its slope:

$$y^{'}(t) = f(t, y), \quad y(t_0) = y_0 \in \mathbb{R}^d, \quad t \in [t_0, t_f] \quad (1)$$

Numerical methods for integrating IVPs generally work in a *step-by-step* manner: the interval $[t_0, t_f]$ is divided into subintervals $[t_0, t_1], [t_1, t_2], ..., [t_{N-1}, t_N]$ ($t_N = t_f$), and approximations $y_1, y_2, \ldots, y_N$ for the solution at the end of each interval are computed in a so-called *integration step*.

Stiff IVPs [2, 8] are an important class of IVPs whose solution is indispensable for modelling a wide variety of time-dependent processes in science and engineering [8]. Due to linear stability reasons [8], the efficient numerical solution of stiff IVPs requires the use of *implicit methods* [8] of ODE solution. Such methods demand a great deal of computing power because they require the solution of a non-linear system of equations at every integration step [2].

The enormous computing power required to solve stiff IVPs can be achieved by using efficient parallel algorithms on Distributed-Memory Parallel Machines (DMPMs). The parallel design of a numerical algorithm to solve stiff IVPs must be structured differently to exploit three factors:

- The task and data parallelism exhibited by the numerical method. The numerical methods used to solve stiff ODEs exhibit two different levels of potential parallelism [11]: 1) *task parallelism*, owing to the fact that these methods can be decomposed into several coarse grain calculations which can be executed independently, and 2) *data parallelism*, because the most basic calculations of the decomposition are linear algebra computations which are susceptible to parallelization following a SPMD style.
- The characteristics of a particular parallel architecture.
- The structure of the problem itself. The Jacobian matrix of the function f which defines the ODE system can exhibit a particular structure (dense, banded, sparse, etc.) and the exploitation of this structure is fundamental to achieve an acceptable performance in a stiff ODE solver.

The *Group Single Program Multiple Data* (GSPMD) [11] programming model is specially suitable to exploit the potential parallelism of these applications on DMPMs. In a GSPMD computation, several independent subprograms are executed by independent groups of processors in parallel. A relevant methodological approach for the derivation

of GSPMD programs is presented in [11]. This approach, termed *TwoL*, allows the exploitation of the two levels of parallelism of multiple numerical schemes but may benefit from explicit constructs to:

- Maintain and select among multiple implementations of an operation in an elegant way. This characteristic, the so-called *performance polymorphism* [9, 10], is very desirable to improve flexibility in performance tuning during the design of a GSPMD program.
- Deal with different IVP structures in order to exploit the particular structure of the matrices and obtain substantial performance improvements.

We propose an extension of the TwoL approach which has explicit constructs for performance polymorphism and takes into account the exploitation of the problem structure. The proposal enables the structuring of the derivation process by including three clearly defined methodological steps in which optimizations of different types can be carried out.

On the basis of our methodological proposal, we perform optimizations on an advanced numerical scheme to solve stiff ODEs and derive efficient distributed implementations of this scheme to deal with dense and banded stiff ODE systems. The numerical scheme implemented is a parallelization of a Radau IIA Implicit Runge Kutta method (Radau IIA IRK method) [8] with a lot of multilevel parallelism.

A brief introduction to the methodological proposal is presented in section 2. The numerical scheme implemented is described in section 3. The first step of this methodology is applied in section 4 to derive a generic specification of the functionality and task parallelism of the method. The second step is applied to adapt this specification to the structure of two IVPs in section 5. The parallel design decisions to derive efficient implementations of the method for a PC cluster are described in section 6. Section 7 presents the experimental results and section 8 gives conclusions.

2. A methodological Approach to derive parallel stiff ODE Solvers

In order to integrate explicit constructs for performance polymorphism into TwoL, we need to decouple the description of the functional behaviour of a linear algebra operation from the multiple implementations which provide the same functionality. We will encapsulate each entity as separate components: *concepts* and *realizations* [9, 10].

A **concept** formally defines the functional behaviour of a linear algebra operation. A concept that denotes the computation of an approximation to the Jacobian of a function f at a point $(t, y) \in \mathbb{R}^{d+1}$ using forward differences is presented in figure 1. The operation functionality is formally specified by supplying a precondition (`REQUIRES` clause) if necessary and a postcondition (`ENSURES` clause) [12].

Considering the importance of the problem structure in

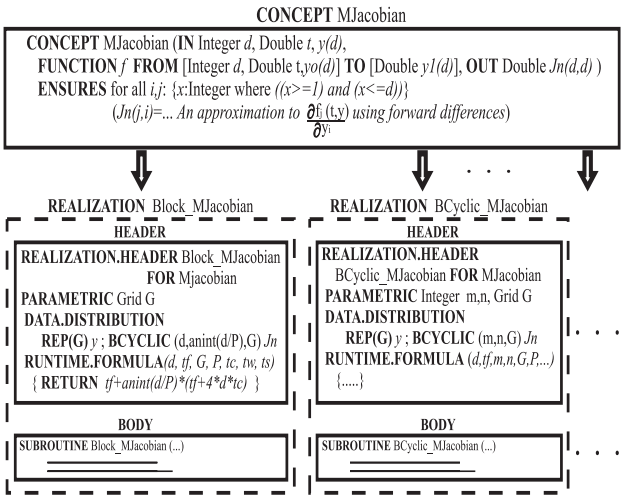


Figure 1. A concept and its realizations

these applications, we define a specialization mechanism to adapt a concept to deal with the particular structure of the input array arguments. When a concept has not been obtained by the specialization of a previously existing one, we say that it is a *general concept*. Here, a general concept is considered as a concept which deals with dense matrices. A general concept can be specialized by restricting the structure of its array arguments. A specialized concept might admit more efficient implementations than a general one because the implementation can take advantage of the special structure of its matrix arguments. For example, we can specialize a concept which denotes the matrix-vector product to deal with banded matrices. An example of hierarchical specializations starting from a general matrix-vector product concept is shown in figure 2. In this figure, the parameters mlA and muA, in the `Banded_MVproduct` concept, represent the number of subdiagonals and superdiagonals respectively of a banded matrix A and the `REQUIRES` clause is used to impose restrictions on the arguments of the new concept obtained.

A **realization** encapsulates the information related to a particular implementation of a linear algebra operation and has two parts: the *header* and the *body*. The *body* is the implementation code of the operation which can be obtained from linear algebra libraries. The *header* is a client-visible part which describes all the performance aspects that the customer programmer needs to know in order to use the component. These aspects include:

a) The data distributions for input and output array arguments. Given an array A and a logical processor grid G, to indicate how A is distributed among the processors of G, we write $REP(G)\ A$, if A is replicated among all the processors of G, and $BCYCLIC(m, n, G)\ A$, if A is block-cyclically mapped on G with $m \times n$ blocks [5].

b) A formula which estimates the execution time of the module from several parameters describing the data distri-

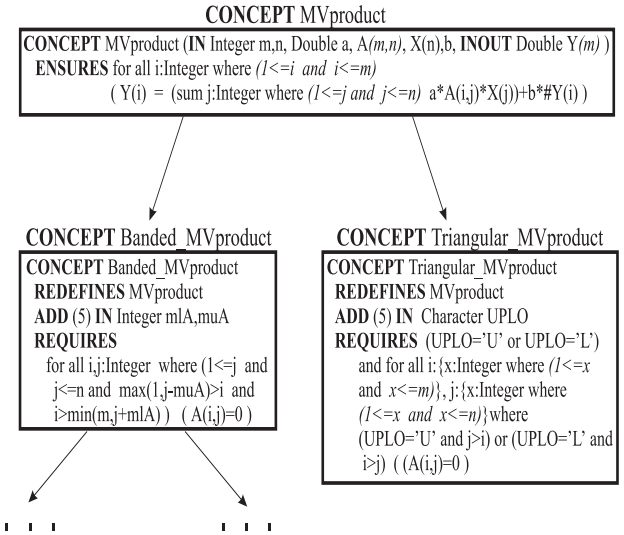


Figure 2. Specializations from a concept

butions (for example, the block size), the target machine (number of processors P, per word communication cost t_w, message startup time t_s, the time taken to execute an arithmetical operation t_c, etc.), and problem size parameters. These formulas are based on a simplified DMPM model and have been used to predict the runtime of GSPMD programs in the TwoL framework [11], achieving good accuracy.

Each concept can have several associated realizations. The explicit distinction between concept and realization as separate components allows us to select the implementation that offers the best performance in a particular context. The choice depends on the target machine characteristics, the problem and the data distribution scheme of the application.

Two different realizations for the $MJacobian$ concept are shown in figure 1. The $Block\_MJacobian$ realization obtains a block column distribution of the Jacobian while the $BCyclic\_MJacobian$ realization works according to a more general block-cyclic distribution of the J_n matrix.

Taking into account the two types of components defined, a methodological approach to derive parallel stiff ODE solvers is proposed. In our proposal, four steps are used, as shown in figure 3. The first three steps are the most important and each one is centred on a different type of optimization. The derivation process starts from a mathematical description of the numerical method.

2.1. Concept Composition

During this step, all the operations involved in the numerical method must be identified and analysed in order to reveal all the potential task parallelism of the method. Several general concepts must be selected and combined by using constructors of sequential and concurrent composition to describe the functionality of the method and the maximum degree of task parallelism. Several structured com-

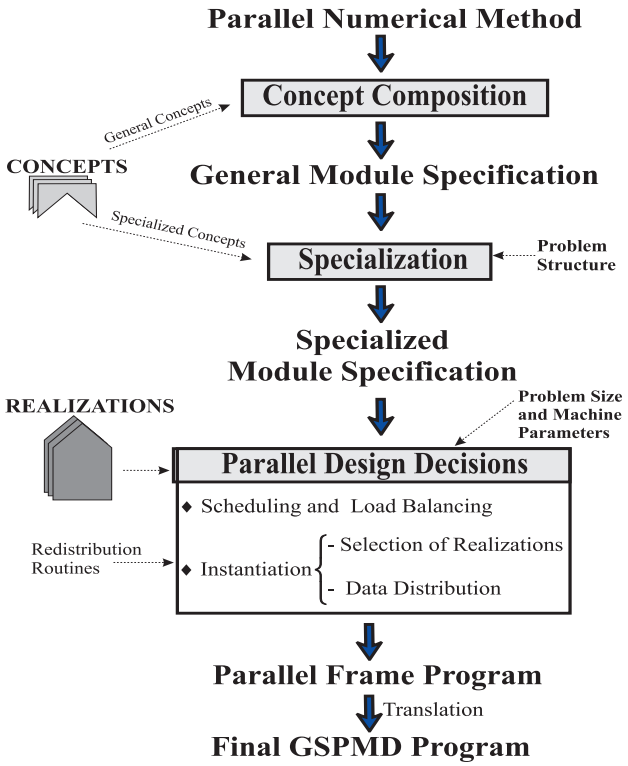


Figure 3. Derivation of Parallel Programs in the proposed approach

binations of general concepts can be encapsulated as more complex components, called composed modules. Finally a general module specification is obtained as a structured combination of general concepts and composed modules.

Therefore, during this step, optimizations that are independent of the architecture and the problem structure can be tackled because only the functionality of the method to be implemented has been taken into account. Moreover, the description obtained is sufficiently generic that it can be used in the solution of problems with different structural characteristics on different architectures.

2.2. Specialization

During this step, the general module specification obtained is adapted to the particular structure of the problem to be solved. Initially, this module specification only contains references to general concepts. According to the structure of the problem to be solved, several concepts are substituted by specializations and if necessary, several important changes must be made to the module specification. As a result, a specialized module specification is obtained.

This adaptation allows the exploitation of the special structure of the problem because realizations adapted to the structure of the matrices can be used in the next methodological step, where all the concepts of the specialized module

specification are substituted by particular realizations.

2.3. Taking Parallel Design Decisions

All the design decisions that affect the performance of the final implementation must be taken at this step. To do this, the following information, besides the previously obtained specialized module specification, has to be available: 1) the existing realizations for each concept of the module specification, and 2) parameters defining the target machine (P, t_w, t_s, t_c, etc.), the problem size (parameters such as the ODE system dimension, the average computational cost of the function f, etc.) and the data distribution types for every array argument of the parallel solver.

The decisions to be taken in this step include an important decision called *instantiation* [9, 10] as well as fixing the execution order of tasks without data dependencies and the size of the processor groups used to execute each task [11]. The instantiation involves the selection of the best realization for each concept of the module specification and the most suitable data distribution parameters for each realization chosen. The insertion of the required data redistribution routines must also be performed in the instantiation. All the decisions must be performed in conjunction to obtain a good global runtime solution.

The problem that arises in this step is NP-Complete. No technique is yet available to obtain suboptimal solutions automatically. We center the instantiation in the more costly operations by using optimal realizations and data distributions for these operations, and we pursue an even distribution of the work among the processors and the minimization of redistribution costs. Following these guidelines, a satisfactory performance solutions have been obtained.

3. The Newton-PILSRK Method

The Radau IIA IRK methods [8] are specially suitable to solve stiff IVPs because they show excellent stability and convergence properties. An advanced numerical method to achieve parallelism across a Radau IIA IRK method is described in [13]. This method, called *Newton-Parallel Iterative Linear System Solver for Runge Kutta methods* (abbreviated as *Newton-PILSRK*), is a mixture of iterative and direct algorithms to solve the nonlinear systems which arise at every integration step, when a Radau IIA method is used.

We have applied the Newton-PILSRK scheme to the Radau IIA IRK method with 4 stages [8] because it is possible to obtain an acceptable degree of task parallelism, and this method exhibits a good convergence order [8]. When the method is used to solve a d-dimensional IVP-ODE given by (1), one has the numerical scheme whose pseudocode description is presented below [13, 10]:

Algorithm

$Y_0 = (y_0, y_0, y_0, y_0)^T \in \mathbb{R}^{4d}$
for $n = 1, \ldots, N$ { /N=number of integration steps/
  $X_n^{(0)} = (S^{-1}P \otimes I_d)Y_{n-1}$; $W = (S^{-1}E \otimes I_d)Y_{n-1}$
  for $j = 1, \ldots, m$ { /m is determined dynamically/
    $R^{(1)} = h_n(S^{-1}A \otimes I_d)F(Y_n{}^{(j-1)}) + W - X_n{}^{(j-1)}$
    (v1) parfor $i = 1, 4$ $\{D_i^{(j,1)} = (I_d - d_i h_n J_n)^{-1} R_i^{(1)}\}$
    $R^{(2)} = R^{(1)} + (I_{4d} - L \otimes h_n J_n) D^{(j,1)}$
    (v2) parfor $i = 1, 4$ $\{D_i^{(j,2)} = (I_d - d_i h_n J_n)^{-1} R_i^{(2)}\}$
    $X_n^{(j)} = X_n^{(j-1)} + D^{(j,1)} + D^{(j,2)}$ }
  $Y_n = Y_n^{(m)}$; $y_n = Y_{n,4}$}

Notation and Definitions

- $Y_n^{(j)} = (S \otimes I_d)X_n^{(j)}, \forall n = 1, \ldots, N, \forall j = 1, \ldots, m$.
- $Y_{n,i} = i$-th d-block of $Y_n = (Y_{n,1}, \ldots, Y_{n,4}) \in \mathbb{R}^{4d}$.
- $F(Y_n) = (f_1, f_2, f_3, f_4)^T$, $f_i = f(t_{n-1} + c_i h_n, Y_{n,i})^T$.
- $E = (e1, e1, e1, e1)^T \in \mathbb{R}^{4\times 4}$, $e1 = (0, 0, 0, 1)$.
- $\forall R \in \mathbb{R}^{4d}$, $R_i = i$th d-block of $R = (R_1^T, ..., R_4^T)^T$.
- I_d denotes the identity matrix of dimension $d \times d$.
- $h_n = t_n - t_{n-1}$ is the *stepsize*.
- J_n is an approximation to the Jacobian of f in (t_{n-1}, y_{n-1}).
- $\otimes$: matrix direct product or Kronecker product [4].

The matrix $A \in \mathbb{R}^{4\times 4}$ and $c = (c_i)_{i=1,\ldots,4} \in \mathbb{R}^4$ are the main parameters of the Radau IIA IRK method with 4 stages [8]. The matrices $L, P \in \mathbb{R}^{4\times 4}$ and the vector $d = (d_i)_{i=1,\ldots,4} \in \mathbb{R}^4$ are constructed in [13] such that: $d_i > 0$, S is a regular matrix, L is a block-diagonal matrix with two blocks and P is a matrix defined in [4] to obtain a good initial approximation to $X^{(0)}$.

We can identify three main sources of task parallelism in the method: **1)** the solution of 4 independent systems of dimension d in the steps (v1) and (v2), **2)** the evaluation of $F(Y^{(j-1)})$ is equivalent to 4 independent evaluations of f and **3)** the computation of $R^{(2)}$, for each value of j, can be decoupled into two independent computations on different blocks of L:

$$\textbf{parfor } i = 1, 2 \ \{R_{i,2d}^{(2)} = R_{i,2d}^{(1)} + (I_{2d} - L^i \otimes h_n J_n) D_{i,2d}^{(j,1)}\} \quad (2)$$

where L^i= i-th 2×2-block of L and $R_{i,2d}$= i-th $2d$-block of $R = (R_{1,2d}^T, R_{2,2d}^T)^T \in \mathbb{R}^4$.

The method also exhibits a lot of data parallelism, due to linear algebra operations which can be implemented following a data-parallel style.

4. Concept Composition for the method

In this step, the Newton-PILSRK method has been optimized. We focus on the computation of $R^{(2)}$. If we expand (2), we obtain: $R_{i,2d}^{(2)} = R_{i,2d}^{(1)} - D_{i,2d}^{(j,1)} + T_{i,2d}$, $i = 1, 2$, where $T_{i,2d} = (L^i \otimes h_n J_n) D_{i,2d}^{(j,1)}$. Since $D_{i,2d}^{(j)} = \left[D_{1+2(i-1)}^{(j)}, D_{2i}^{(j)}\right]^T$, then we have, for $i = 1, 2$:

$$T_{i,2d} = \begin{bmatrix} L_{1,1}^i h_n J_n D_{1+2(i-1)}^{(j)} & L_{1,2}^i h_n J_n D_{2i}^{(j)} \\ L_{2,1}^i h_n J_n D_{1+2(i-1)}^{(j)} & L_{2,2}^i h_n J_n D_{2i}^{(j)} \end{bmatrix}$$

Table 1. Some general concepts used

Concept	Functionality description
MVproduct(m, n, a, A, X, b, Y)	$Y \leftarrow aAX + bY$, $X, Y \in \mathbb{R}^n$,$a, b \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$
LUdecomp($.., A, Ip$)	LU Factorization of A
SolveSystem(n, A, Ip, X)	$X \longleftarrow A^{-1}X$, assumes LUdecomp($.., A, Ip$)
MJacobian(n, t, y, f, A)	$A \approx \frac{\partial f}{\partial y}(y,t)$ $y \in \mathbb{R}^n, t \in \mathbb{R}, A \in \mathbb{R}^{n \times n}$
MsumId(n, a, A, B)	$B \longleftarrow I_n + aA$ $A, B \in \mathbb{R}^{n \times n}$, $a \in \mathbb{R}$

Clearly, in the computation of $T_{i,2d}$, the terms $JD_i = h_n J_n D_i^{(j)}, \quad i = 1, ..., 4$, can be computed only once to be reused in the rest of the computation. As a result, to compute $R^{(2)}$, we can use the scheme presented below with a lower computational cost and more task parallelism.

$$\textbf{parfor } i = 1, \ldots, 4 \{ \quad JD_i = h_n J_n D_i^{(j)}; \quad \sum_{l=1}^{2} \left(L_{\frac{i+1}{2}\ (i-1) \bmod 2+1, l} \cdot JD_{l+2(\frac{i-1}{2})} \right) \}$$

In order to describe the functionality of the new version of the method, we have selected several general concepts. The functionality of some of these concepts is briefly described in Table 1. To combine these concepts, we can use a graphical notation which expresses the concurrent and sequential composition as a directed graph (see figure 4).

In figure 4, the arrows denote data dependencies between operations. A node of the graph can represent a reference to a single operation or a concurrent loop (`PAR i=1,N`) applied on a sequence of operations. Some nodes of the graph include descriptions of steps in the algorithm.

5. Specializations for the method

5.1. Description of the IVPs to be solved

We have applied the method to two stiff IVPs:

a) An IVP, noted as *Vortex*, which models the evolution of a singular vortex patch governed by a two-dimensional incompressible fluid flow [3]. An integro-differential equation of the form $\frac{\partial r(t,\xi)}{\partial t} = \ldots$ is known, where $r(t, \xi) \in \mathbb{R}^2$ is a parameterization of the boundary of the patch contour for $t \geq 0$ and $\xi \in [0, 1]$.

Supposing the vortex patch boundary has been parameterized by using N points (N must be even) equally distributed $r_i = (r_{i1}, r_{i2}), \quad i = 1, \ldots, N+1$ ($r_1 = r_{N+1}$ and $r_2 = r_{N+2}$), the integro-differential equation can be numerically evaluated [3] to obtain the following ODE system:

$$\frac{\partial r_i}{\partial t} = \sum_{j=1, r_i \neq r_j}^{N+1} \frac{\alpha_j (r_i - r_j)}{6\pi |r_i - r_j|^2} ((r_i - r_j) \cdot (r_{j+1} - r_j)) + \frac{\alpha_i (r_{i+1} - r_i)}{6\pi}, \qquad i = 1, \ldots, N$$

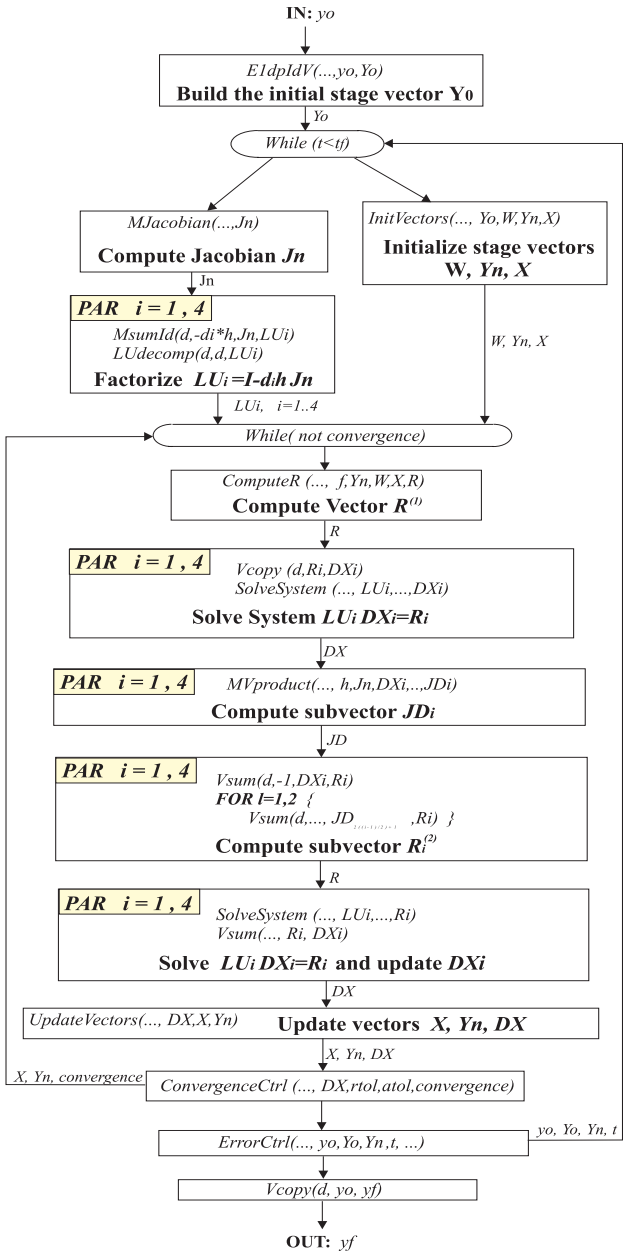


Figure 4. Graphical description of the general module specification for the method

where $\alpha_1 = \alpha_{N+1} = 1$ and, for all $j = 2, \ldots, N$, if j is even then $\alpha_j = 4$ else $\alpha_j = 2$. The initial value is given for the parameterization of the boundary of a circle with radius 1 centred at the origin.

Since each point $r_i, \quad i = 1, \ldots, N$ has two real coordinates (r_{i1}, r_{i2}), the resultant system has $2N$ ODEs and involves a fully dense Jacobian.

b) A stiff IVP, noted as *ConvDiff2D*, which is based on a partial differential equation describing the linear decay of a two dimensional model convection-diffusion equation

bounded by two parallel walls [14] where $0 < x < 200\pi$, $0 < y < 1$ and $t \geq 0$. A suitable finite-difference discretization of the spatial derivatives by using an uniform $N_x \times N_y$ grid leads to a system of $N_x N_y$ ODEs:

$$\frac{\partial u_{i,j}}{\partial t} = -\frac{11u_{i,j} - 18u_{i-1,j} + 9u_{i-2,j} - 2u_{i-3,j}}{6\Delta x} + \frac{1}{10}\frac{-u_{i,j+2} + 16u_{i,j+1} - 30u_{i,j} + 16u_{i,j-1} - u_{i,j-2}}{12\Delta y^2} - \frac{-u_{i,j+2} + 8u_{i,j+1} - 8u_{i,j-1} + u_{i,j-2}}{12\Delta y}, \quad i=1,\ldots,N_x,\ j=1,\ldots,N_y$$

Here, $\Delta x = \frac{200\pi}{N_x - 1}$, $\Delta y = \frac{1}{N_y + 1}$. The translation of the boundary conditions is suitably defined in [14]. The initial conditions are $u(x, y, 0) = e^{Ry/2} sin(3\pi y) cos(0.01x)$. The components of the system are ordered according to: $u_{11}, \ldots, u_{1N_y}, \ldots u_{21}, \ldots, u_{2N_y}, \ldots, u_{N_x N_y}$. With this ordering, the Jacobian of the problem has a banded structure with 2 subdiagonals and $3N_y$ superdiagonals.

5.2. The specialized module specification

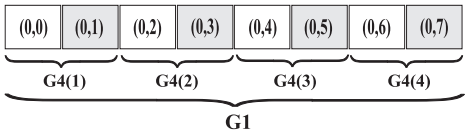
In the case of the dense IVP, it is not necessary to perform changes on the general module specification because the matrices involved in the operations are dense. In the case of the IVP with banded Jacobian (*ConvDiff2D*), several concepts must be replaced by their specializations for banded matrices. The new specialized concepts include new parameters (mlA, muA) which represent the number of subdiagonals and superdiagonals respectively of the matrix involved (A). Although, in this case, the specialization consists of replacing the concepts affected by specializations, in numerous circumstances it may be necessary to carry out more important changes in the general module specification.

6. Parallel Design Decisions to derive implementations of the method

Starting from the specialized module specifications (for dense and banded systems) obtained for the method, we derive efficient implementations of this parallel numerical scheme for a cluster of 8 PCs based on Pentium II (333 MHz). The interconnection network is a switched fast Ethernet where we have experimentally obtained the values $t_s \approx 286\mu s$ and $t_w \approx 1.34\mu s$ for MPI Send and Receive primitives [7]. We assume that all the vector arguments of the solver (the initial vector y_0 and the solution vector y_f) are replicated among the processors. For the *Vortex* IVP, we consider a dimension d such that $d \leq 1400$. For the *ConvDiff2D* IVP, we suppose that N_y varies between 24 and 40.

6.1. Scheduling and Load Balancing

A graphical description of the processor groups involved in the GSPMD computation is shown in figure 5. A logical 1×8 processor grid (the global group $G1$) is partitioned into four groups, each with 2 processors ($G4(i)_{i=1,\ldots,4}$).



Dense case

```
DATA.DISTRIBUTION
 CONST (MB=D/2)
 BCYCLIC(1,MB,G1) VDG1
 BCYCLIC(D,D/8,G1) MDG1
 REP i=1,4
  BCYCLIC(64 , 1 ,G4(i)) VDG4(i)
  BCYCLIC(64,64,G4(i)) MDG4(i)
```

Banded case

```
DATA.DISTRIBUTION
 CONST (NR=MlJn+MuJn+1,
        MB=D/2)
 BCYCLIC(1,MB,G1) VDG1
 BCYCLIC(NR,D/8,G1) MDG1
 REP i=1,4
  BCYCLIC(1,MB ,G4(i)) VDG4(i)
  BCYCLIC(NR,MB,G4(i)) MDG4(i)
```

Figure 5. Description of the processor groups and the data distribution types

A graphical description of the parallel frame program for dense systems is shown in figure 6. We do not describe the parallel frame program for banded systems because we simply explain the differences regarding the dense case.

In figure 6, the nodes represent a realization call or parallel loops applied on a sequence of realization calls. Every realization call is assigned to a processor group by using the keyword `ON`. The arrows denote the real execution order among realizations. When data redistribution between connected nodes is necessary, these arrows are labelled with redistributions which are indicated as transitions among arrays distributed according to different distribution types:

Distrib.Type1 Array1 $\longrightarrow$ Distrib.Type2 Array2

As can be seen in figure 6, the operations executed in parallel on the disjoint groups $G4(i)_{i=1,\ldots,4}$, correspond to concurrent loops, which denote most of the task parallelism of the method. The remaining operations are executed consecutively on the group $G1$. With these decisions, an even distribution of the computational load is achieved.

6.2. Instantiation

The particular data distribution types used in the instantiation are described in the `DATA.DISTRIBUTION` section of the parallel frame program (see figure 5), where the block-cyclic distribution template is instantiated by specifying a block size (m rows and n columns) and a previously defined group G: $BCYCLIC(m, n, G)$ Distrib.Type.

In figure 6, the arguments of a realization call which are distributed (not replicated) on a group with more than one processor are written in boldface and preceded by the particular data distribution type.

In the following, we will briefly analyse the decisions taken in the instantiation for dense and banded systems.

Dense system version

The operations which can be computed independently for each stage (see figure 4) are performed in parallel on dis-

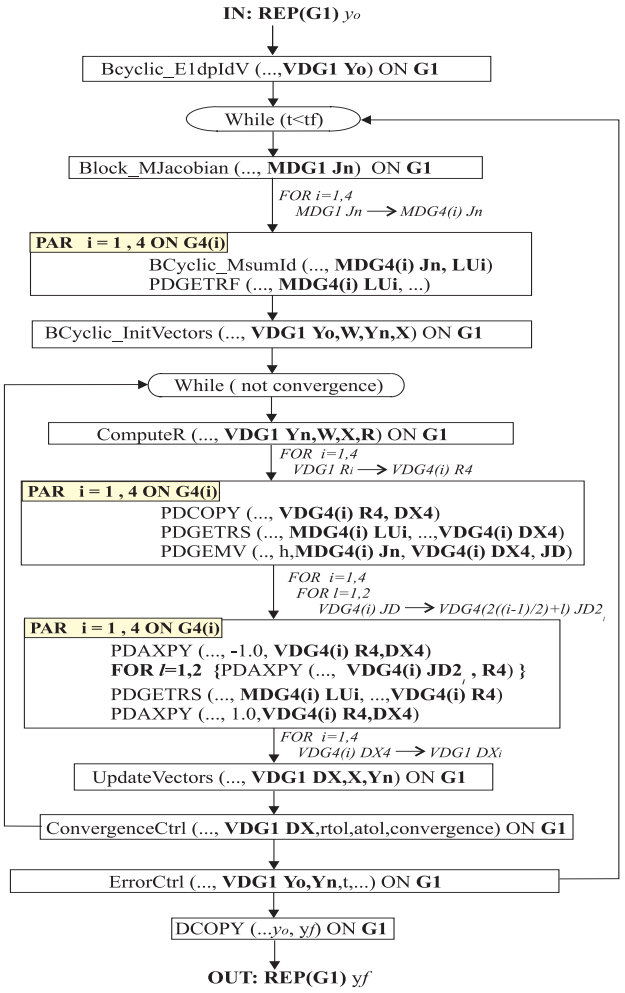


Figure 6. Graphical description of the parallel frame program for dense Jacobian

joint groups of two processors ($G4(i)_{i=1,\ldots,4}$). To perform the calculations associated to each stage of the method, we selected realizations from the parallel libraries PBLAS and ScaLAPACK [1].

In order to perform the LU decompositions and system solutions for each stage, we used the ScaLAPACK realizations `PDGETRF` (LU Factorization) and `PDGETRS` (system solution) following a 64×64 block-cyclic distribution ($MDG4(i)_{i=1,\ldots,4}$) for their matrix arguments and a 64×1 block-cyclic distribution ($VDG4(i)_{i=1,\ldots,4}$) for the vector. These choices provide good performance.

To compute an approximation to the Jacobian (concept `MJacobian`), we used the `Block_MJacobian` realization which computes the Jacobian according to a block column distribution and gives the best runtime results.

The layout of the vectors R, DX ($D^{(j,v)}$) and JD does not match perfectly in different computation phases. However the redistributions of these vectors do not require a great deal of interprocessor communication and, despite the redistribution of the Jacobian matrix, redistribution expenses are not high compared with computation costs.

Banded system version

The specialized concepts used to deal with the banded structure of their array arguments have been instantiated with realizations which assume a compact storage scheme for banded matrices [1] in which an $n \times n$ banded matrix A with bandwith $mlA + muA + 1$ is stored in an $(mlA + muA + 1) \times n$ array. Moreover, these realizations assume a block column distribution scheme for the matrix arguments. The ScaLAPACK realizations `PDGBTRF` and `PDGBTRS` [1] have been used to solve the banded linear systems. These realizations assume a block column distribution scheme for the compact representation of the banded matrix, and a block row distribution for the vector.

To compute the vectors $JD_i = h_n J_n D_i^{(j)}$, we used a parallel version of the banded matrix-vector product, which achieves good results and assumes the same data distribution types as the `PDGBTRS` realization. This choice minimized the redistribution expenses.

Now, the redistribution costs are lower than in the dense case because the distributions of the vectors is hardly ever modified during computation. However, the locality is reduced, because the computation costs are much lower.

7. Experimental Results

The parallel frame programs obtained were translated into a message passing program which is expressed in Fortran augmented with calls to routines of a version of the BLACS library [6] implemented onto MPI. We compared the runtime performance of these programs with two sequential solvers: one of the most efficient stiff ODE solvers, the experimental code *RADAU5* [8], and a sequential version of the optimized Newton-PILSRK method that we called SNPILSRK. Both solvers can take advantage of the banded structure of the Jacobian.

Speedup values have been obtained by comparing the parallel execution times with the execution time of the sequential programs over one integration step. The resultant speedup values obtained for the dense IVP with different dimensions reveal (see figure 7) that with more than 300 equations, a speedup of 5 to 7.55 can be achieved on 8 nodes with regard to RADAU5. The results obtained with regard to SNPILSRK are better, achieving a superlinear speedup with more than 700 equations. When the structure of the Jacobian is banded (see figure 8), we considered the most significant parameter of the problem size to be the number of nonzero entries in the Jacobian. As can be seen, performance is degraded because the locality is reduced. However, a speedup of 3.5 to 4.35 can be achieved with respect to RADAU5 and a speedup of 3.8 to 5.7 can be achieved with respect to SNPILSRK.

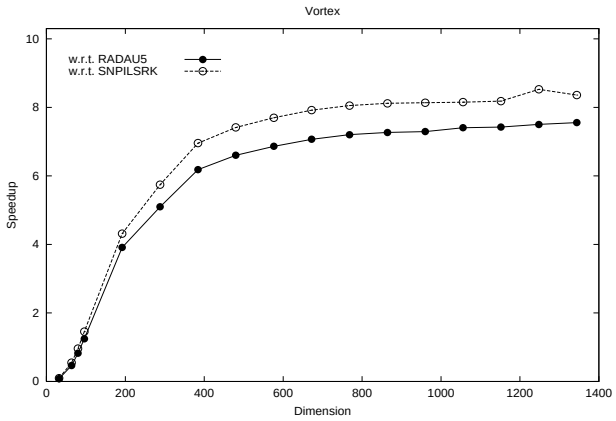


Figure 7. Speedup with the *Vortex* IVP

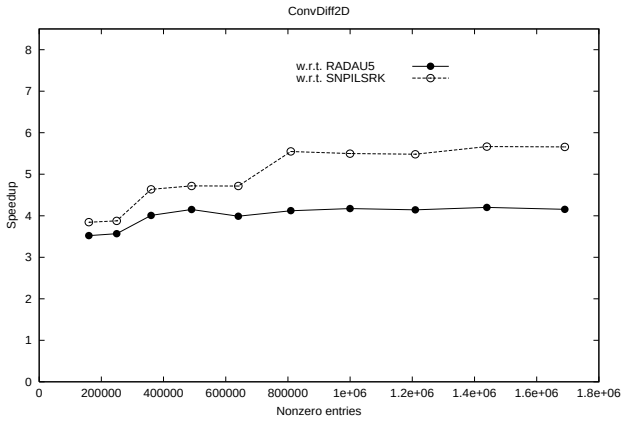


Figure 8. Speedup with the *ConvDiff2D* IVP

8. Conclusions

A methodological approach to enable the separate treatment of different aspects during the derivation of a parallel stiff ODE solver is proposed. These aspects are:

a) *Functional aspects*: A reusable description of the functionality and the task parallelism of the method is obtained in the *concept composition* step regardless of assumptions about the particular problems to be solved and the parallel machine. This permits us to postpone as late as possible dealing with architectural details and to focus on optimizations independent of the machine and the problem.

b) *Aspects dependent on the problem structure*: The adaptation of the results of the above step to the problem structure is carried out in a machine-independent way during the *specialization* step. This enables the exploitation of this structure in the next step.

c) *Performance aspects dependent on both the architecture and the problem*: All the parallel design decisions which affect the performance of the final program have to be taken in the last step, taking into account both the parameters of the problem and the machine.

The methodological proposal has been employed to derive an efficient stiff ODE integrator for a PC cluster with 8 nodes. The integrator is based on an advanced numerical method which has been optimized and adapted to the solution of two modelling problems with different structural characteristics. Speedup comparisons with regard to one of the most advanced sequential stiff ODE solvers have been made. The implementation for dense systems achieves a speedup of 5 to 7.55. The implementation for banded systems achieves a speedup of 3.5 to 4.35.

Acknowledgements

This work was supported by the project TIC2000-1348 of the Ministerio de Ciencia y Tecnología.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[2] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford Science Publications, 1995.

[3] J. A. Carrillo and J. Soler. On the evolution of an angle in a contour dynamic. *The Journal of Nonlinear Science*, 1997.

[4] J. J. B. de Swart. *Parallel Software for Implicit Differential Equations*. PhD thesis, Amsterdam University, 1997.

[5] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 1995.

[6] J. J. Dongarra and R. C. Whaley. A user's guide to the blacs v1.0. Technical Report CS-95-292, Computer Science Dept. University of Tennesee, 1995.

[7] M. P. I. Forum. *MPI: A Message Passing Interface Standard*. Univ. of Tennessee, Knoxville, Tennessee, 1995.

[8] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential Algebraic Problems*. Springer-Verlag, 1996.

[9] J. M. Mantas and J. O. Lopera. A component-based stiff ode solver on a cluster of computers. *4th International Meeting on Vector and Parallel Processing, VECPAR'2000*, June 2000.

[10] J. M. Mantas, J. O. Lopera, and J. A. Carrillo. Component-based derivation of a stiff ode solver implemented on a pc cluster. *To appear in the International Journal of Parallel Programming*.

[11] T. Rauber and G. Rünger. Deriving structured parallel implementations for numerical methods. *The Euromicro Journal*, 41:589–608, 1996.

[12] M. Sitaraman and B. Weide. Special feature: Component-based software using resolve. *ACM SIGSOFT, Software Engineering Notes*, 19, 1994.

[13] P. J. Van der Houwen and J. J. B. de Swart. Parallel linear system solvers for runge-kutta methods. *Advances in Computational Mathematics*, 7:157–181, March 1997.

[14] X. Zhong. Additive semi-implicit runge-kutta methods for computing high-speed nonequilibrium reactive flows. *Journal on Computational Physics*, 128:19–31, 1996.