Solving Initial Value Problems with Parallel Maple Processes

Dana Petcu

Western University of Timișoara, B-dul V.Pârvan 4, 1900 Timișoara, Romania,
petcu@info.uvt.ro,
<http://www.info.uvt.ro/~petcu>

Abstract. The recently developed PVMaPle (Parallel Virtual Maple) provides a message-passing programming environment to solve complex problems, like large systems of ordinary differential equations (ODEs), on network of workstations. The computing facilities proposed by PVMaPle are briefly presented. Numerical tests concerning the approximate solutions of some particular ODEs are reported and discussed.

1 Introduction

Computer algebra systems, like **Maple** or **Mathematica**, can be used with success in prototyping sequential algorithms for symbolic or numeric solution of mathematical problems. Constructing prototypes and libraries of parallel algorithms in such currently used systems is an actual challenging problem.

Several successful attempts have been made to combine **Maple** with parallel or distributed computation features (see for example [15] or [17]). We shortly present in Section 2 a prototype system, **Parallel Virtual Maple**, shortly **PVMaPle**, which interconnects **PVM** (Parallel Virtual Machine) and **Maple**. **PVMaPle** package has been developed in the idea to allow several independent **Maple** kernels on various machines connected by a network to participate in computation, hopefully leading to faster results. More details about can be found in [11].

The aim of this paper is to investigate in what extent the theoretical parallelization of special iterative methods can be achieved in practice using **PVMaPle**. In particular, we solve large initial value problems (IVPs) for ordinary differential equations (ODEs), problems of the form $y'(t) = f(t, y(t))$, $y(t_0) = y_0$, where $t \in [t_0, t_0 + T]$, $y_0 \in R^n$ and $f : [t_0, t_0 + T] \times R^n \rightarrow R^n$, and y is the unknown function. For the simulation of many problems of practical importance, such as large scale integrated circuits (VLSI), the dimension n may be extremely large. Standard techniques have been shown to be adequate for solving moderately sized systems, but they may be inappropriate for systems consisting of many thousands of equations. As a consequence, substantial effort has been focused towards the development of numerical methods which are capable of solving very large systems of ODEs. With the advent of parallel technologies, it is natural to desire to take advantage of the increasing computing power available. Section 3 discuss some implementation issues of parallel ODE solving methods. Previously proposed parallel methods are examined from the perspective of computation graph

distribution and PVMMaple facilities. Section 4 presents and interprets the results of some numerical tests which have been run in a PC network. We show that an ODE solver for large problems can significantly benefit from the distributed computing environment provided by PVMMaple. We consider the resulting Maple code for ODEs as a first step in the direction of designing and testing a more extensive library in Maple for parallel algorithms.

2 PVMMaple

Parallel Virtual Maple is a prototype system allowing to study the issue of interconnecting PVM and Maple. Its design principles are very similar to that of Distributed Maple [15], a portable system for writing parallel programs in Maple and for network environments where Maple and Java are available. Other systems combining Maple with parallel or distributed computation features are ||Maple|| [17] (built as an interface between the parallel declarative programming language Strand and the sequential CAS Maple), Sugarbush [2] (combines the parallelism of C/Linda with the Maple kernel), or FoxBox [3] (a client/server style interface to Maple which provides an MPI-compliant distribution mechanism allowing for parallel execution of FoxBox programs). The PVMMaple functionality is described by Figure 1. Combining Maple with parallel computation factors is done in PVMMaple wrapping Maple by an external system which takes care of the parallel execution of tasks. PVM active daemons (PVMs) ensure the inter-processor communications. In order to start PVMMaple, a workstation user must have access to a functional Maple V version and it must have the possibility to create a parallel virtual machine with PVM. Each node connected to a PVMMaple session comprises three main components: command messenger, Maple interface

Fig. 1. Active processes and intercommunication routes

Table 1. Package pvm.m

Function	Meaning	Equivalence with Distributed Maple
spawn	create local or remote Maple processes	initialize
send	send commands to Maple processes	start
receive	receive results from Maple processes	wait
settime	start a chronometer	trace (post-processing)
exit	stop PVM Maple	terminate
time	post-processing time diagram in Maple	visualise (on line)
ProcID	process identifier on one machine	-
MachID, TaskID	station/process identifier into PVM Maple	-

and PVMD daemon. The special binary named `command-messenger` is responsible for the message exchanges between the Maple processes (into the PVM machine), coordinates the interaction between Maple kernels via PVM daemons, and schedules tasks among nodes. The Maple file `pvm.m` implements the interface between the Maple kernel and `command-messenger`. Each Maple process has its own `command-messenger` (his pair) which receives Maple commands from other Maple processes (local or remote), sends its results in Maple format to other processes, activates or destroy other Maple processes. At least two Maple processes must be activated to apply PVM Maple functions. The `command-messenger` must be placed on the workstation which will start the virtual machine. This binary reads a special file with the paths of the Maple and PVM binaries. The remote Maple processes and their pairs, `command-messengers`, are activated using the PVM facilities of dynamically hosts and processes adding. Table 1 shows the current available functions of this package and the equivalence with Distributed Maple functions. Table 2 presents some suggestive examples of function synopsis.

`Command-messenger` and `pvm.m` (binaries for Win9x and Unix, respectively for Maple Release 4, 5 and 6) are available for free download from the web address <http://www.info.uvt.ro/~petcu/pvmaple>. The user interacts with the system via the text oriented Maple front-end: initialization of a PVM Maple session, activation of local or remote Maple processes and message-passing facilities are provided by the new Maple package, `pvm.m`. `Pvmaple` does not allow shared objects like Distributed Maple and on-line visualization of active processes, but it allows more than one Maple command to be send once to remote processes. Tests performed

Table 2. Command examples (PC1 and X are workstation names)

Maple command	Meaning
<code>pvm[spawn]([PC1,3],[X,2]);</code>	start 3/2 Maple processes on PC1/X machine
<code>pvm[send]([X,2], 'x:=pvm[ProcId]');</code>	send to X's 2th process an assigning comm.
<code>a:=pvm[send]('all', 'int(y^ x,x)');</code>	send to all Maple processes an int comm.
<code>pvm[send]([X,2], 'pvm[send]([X,1], "...</code>	send a sending command
<code>pvm[receive]('all', 'all');</code>	receive all incoming messages
<code>pvm[receive](a, 'all');</code>	receive the results of a specific command
<code>pvm[receive](a, [X,1]);</code>	receive all the messages from a workstation

on Unix networks where both applications are available have not reveal significant differences in application execution times [14].

A facility for post-execution analysis of the dynamic behaviour of a session is provided by the `time` function. In [11] is presented the graphical result of a `pvm[time]` call in the case of a distributed square matrix-vector multiplication (of hundred order of magnitude). Another example is presented in Section 4.

3 Implementation Issues of Parallel Methods for ODEs

Finding methods and strategies for solving an IVP on parallel computers has been the matter of several researches in the last thirty years (see [5] for a more complete overview). Such methods have been classified in different classes, according to the way they explore the potentialities of a parallel machine. A general way of devising parallel ODE solvers is that of considering methods whose work per time-step can be split over certain number of processors. The so-called solvers with parallelism across the method are then obtained. Other options are parallelism across system and parallelism across steps. Parallelism across system can be applied if the current IVP can be split into independent subsystems. Parallelism across steps is encouraged when a linear IVP is solved.

Concerning the Runge-Kutta methods a special scheme was proposed in [7] to distinguish their degree of parallelism: the digraph method. We have extended this method to the case of an arbitrary difference method for IVP [9] and we have implemented it in an ODE solving environment [10]. So, we have propose in [9] the following form for a difference method for ODEs:

$$Y^{(n+1)} = AY^{(n)} + h_n \Phi(t_n, Y^{(n)}, h_n), \quad Y^{(0)} = \Psi(h), \quad e^{(n)} = Y^{(n)} - Z(t_n, h_n)$$

where Ψ is a starting procedure, and A is a problem independent matrix. The first equation is an advance formula, $e^{(n)}$ is a global error estimation for the approximate solution, and $Z(t_n, Y)$ is a procedure for approximate solution validation. Ψ is a problem-dependent function, $h_n = t_{n+1} - t_n$ is the step-size used to the current iteration step, and Y_n is a vector representing numerical values of the dependent variable y at different values of the independent variable t or numerical values of f at different values of y and t .

Consider the set S of all values of the dependent variable y which appear in the above iterative formulae of a method. The computation necessary to advance the solution from one step to another can be expressed by some relationships between the values from S . These relationships can be expressed by a graph. Execution of a graph means computing the output values of a computation node when all of the input values to that node have been computed. When there are cycles in the graph, computation is not possible until the graph has been modified to break the cycles (by predictor-corrector schemes). Examples of such graphs are presented in [9].

Distributed execution of a computation graph consists in assigning each node in the graph to a process. All processes can operate simultaneously only if every process has nodes which are ready to be evaluated at all times. A method is

a said to be a s -value q -level p -process if q is the smallest integer for which the s new dependent variable values can be evaluated in q time-steps and p is the smallest number of processes for which this value of q can be attained. For an explicit IVP method each time-step is equal to the time required by some function evaluations plus the time for some vector operations, while for an implicit IVP method, each time-step is equal to the time required to solve some implicit equations plus the time for some vector operations. For load balancing, formulas with $s \approx pq$ are preferred.

4 Numerical Experiments

We present the results obtained by using PVM_{Maple} prototype and implementing the procedure from Table 3 (sketches the main parts of a PVM_{Maple} procedure for a parallel method using p processors). We have selected some large IVPs and known parallel methods. Numerical tests have been performed on a network composed by four identical PCs Pentium at 450 MHz with 64 MB of RAM running Windows'98 and connected by Ethernet links.

Three test problems are referred here. The first one, a linear problem, is denoted by $T(n)$ and it is was described in [16] (a partial differential equation solved with the method of lines):

$$y'_i = \pi^{-2}(n+1)^2(y_{i+1} - 2y_i + y_{i-1}), \quad y_{-1} = y_{n+1} = 0, \quad i = 1, \dots, n$$

on the interval $[0, 2.5]$, with the initial values $y_i(0) = \sin(\pi i(n+1)^{-1})$, $i = 1, \dots, n$. The second one (denoted by $V(n)$) is a nonlinear problem [1]:

$$y'_i = (y_{i-1}y_{i+2} - y_{i+1}y_i)^2 - 0.1iy_i, \quad i = 2, \dots, n-2, \quad y'_i = -0.1iy_i, \quad i = 1, n-1, n$$

on the interval $[0, 1]$, with the initial values $y_i(t) = 1$, $i = 1, \dots, n$ and the exact solution $y_i(t) = e^{-0.1it}$. The third one is a mathematical model of the movement of a rectangular plate under the load of a car passing across it (plate problem denoted by $P(n)$, from [8]):

$$u_{tt} + \omega u_t + \sigma \Delta \Delta u = f(x, y, t), \quad u|_{\partial\Omega} = 0, \quad \Delta u|_{\partial\Omega} = 0, \quad u_t(x, y, 0) = 0$$

on the interval $[0, 7]$, with $\Omega = [0, 1]^2$, $\omega = 1000$, $\sigma = 100$ and f being the sum of two Gaussian curves with four wheels which move in the x -direction. Applying

Table 3. Implementation of a s -value q -level p -process iterative scheme (the procedures `sys_solve`, `update`, `P`, and the iterative equation `equ`, specific for each particular method, must be defined outside the discussed procedure)

```

one_step_parallel_scheme:=proc(Y, h)
  for j from 1 to q do comm:=‘eqs:={} }; ukn:={} };
    for i in P(pvm[TaskId],j) do eqs:=eqs union equ[i]; ukn:=ukn union {uk[i]}; od;
  sys_solve(eqs, ukn); S:=pvm[send](‘all’, cat(‘Y:=’, Y, ‘; h:=’, h, ‘;’, comm));
  K1:=execute(comm); K2:=pvm[receive](S, ‘all’); od;
update(y, K1, K2); end;

```

the method of lines, the partial derivatives in x and y directions are replaced with approximation evaluated at different spatial grid points. The procedure lead to an ODE system with independent variable t . The number of spatial grid points depends on the accuracy required in the PDE solution. As the accuracy requirement increases, the spatial grid needs to be refined and this leads to a larger ODE system. Obtaining the solutions of such problems like the above ones can be a time and space expensive process. For example, using a simple method like Euler's implicit rule for the discretized plate problem with 180 ODEs, the time necessary to complete the integration with EpODE [10], with the highest step allowed by the problem stiffness is around 45 hours. One solution to this inconvenient situation is to apply a parallel method which can solve faster the above problems.

Four parallel methods are discussed here. The first one, denoted by PC is the second-order explicit block predictor-corrector scheme proposed in [6] (in our definition, a 4-value 1-level 4-process method):

$$\begin{aligned} y_{2n+2}^P &= y_{2n-2}^C + 4hf_{2n}^P, & y_{2n+1}^P &= y_{2n-2}^C + 3h(f_{2n}^P + f_{2n-1}^P)/2 \\ y_{2n}^C &= y_{2n-3}^C - h(3f_{2n}^P - 9f_{2n-1}^P)/2, & y_{2n-1}^C &= y_{2n-3}^C + 2hf_{2n-2}^C. \end{aligned}$$

The second one, DIRK method, is the fourth-order L-stable block diagonally implicit Runge-Kutta method from [7] (a 5-value 2-level 2-process method):

$$\begin{aligned} y_{n+1} &= y_n + h[a(k_1 + k_2) + (d - a)(k_3 + k_4)], \\ k_1 &= f(v_n, y_n + h(ck_1 + (f - c)k_2)), \quad k_2 = f(u_n, y_n + h((e - c)k_1 + ck_2)), \\ k_3 &= f(v_n, y_n + h(dk_3 - bk_4)), \quad k_4 = f(u_n, y_n + h(bk_3 + dk_4)), \end{aligned}$$

$a = \frac{3}{2}, b = \frac{\sqrt{3}}{6}, c = \frac{5}{12}, d = \frac{1}{2}, e = d + b, f = d - b, u_n = t_n + eh, v_n = t_n + fh$. The third one is a PDIRK method, a third-order L-stable parallel diagonally implicitly iterated Runge-Kutta method [5] (a 7-value 4-level 2-process method):

$$\begin{aligned} y_{n+1} &= y_n + hb^T f(t_n e + hc, Y^{(m)}), \quad Y^{(0)} = y_n e, \quad e = (1, \dots, 1)^T, \\ Y^{(j)} - hDf(t_n + hc, Y^{(j)}) &= y_n e + h(A - D)f(t_n e + hc, Y^{(j-1)}), \\ \text{with } m = 3, \quad D &= 0.4358665 I, \quad A = \frac{1}{12} \begin{pmatrix} 5 & -1 \\ 9 & 3 \end{pmatrix}, \quad b = \frac{1}{4} \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \quad c = \frac{1}{3} \begin{pmatrix} 1 \\ 3 \end{pmatrix} \end{aligned}$$

The last one is a 9-stages, 3-processor, 4th-order, A-stable parallel singly diagonally iterated Runge-Kutta method (PSDIRK) presented in [4] and [12] (a 10-value 4-level 3-process method, with a too big iterative representation to be reproduced here). Between the first method and the other three is a big difference in time requirements to fulfill an integration step. The first method is explicit and do not need supplementary procedures in order to solve some implicit equations at each integration step. Unfortunately, large IVPs are almost all stiff, i.e. cannot be integrated by explicit methods since then time step-sizes are restricted to very small values due to method stability conditions. Moreover, if we think to implement a parallel method we must take into account that an exchange information between two processors is a time-expensive operation. For

Fig. 2. Time diagram of 4 steps of DIRK applied to $T(30)$. The active periods of Maple processes and command messengers are indicated by horizontal lines. The total active time (between a `settime` and an `exit`) of a Maple process is expressed in %, so that the user can easily estimates the load balance of parallel algorithm and the delays sources

example, the time necessary to send a n -dimensional vector between two PCs via PVM environment is $10^4/2$ times greater than the time of a function evaluation and 10 times greater than the time of a Newton iteration in the case of solving with DIRK method the discretized plate problem with 80 ODEs.

We solve the advance equations corresponding to each level of the above mentioned methods on different processors (like in Table 3). If we use the Maple implicit equation solver, `fsolve`, as `sys_solve` procedure, most of the time is spent in solving stage equations. The reported execution times of problem solving sessions, provided by `pvm[time]` procedure, demonstrate this assumption. A concrete example is presented in Figure 2.

Table 4. Efficiency results $E_p = T_s/(pT_p)$. T_s is the sequential time reported by Maple as problem solving time using one of the above described methods; T_p is registered by PVM Maple in solving same problem with same method but using p kernels on p machines

Pro- blem	No. ODEs eqs.	Methods			
		PC	DIRK	PDIRK	PSDIRK
		$p = s = 4, q = 1$	$p = q = 2, s = 5$	$p = 2, q = 4, s = 7$	$p = 3, q = 4$
$T(n)$	10	1%	35%	14%	10%
	20	3%	63%	29%	23%
	60	5%	88%	77%	68%
$V(n)$	5	1%	63%	28%	24%
	10	4%	92%	51%	52%
	20	8%	99%	97%	99%
$P(n)$	32	8%	96%	94%	93%

The efficiency measurements of the distributed implementation of the selected methods are presented in Table 4 and, shortly, in [13]. We conclude that small linear IVPs cannot be integrated in a distributed computational environment faster than using a sequential computer, since the distributed computation tasks are small (in time sense) relative to the communication tasks. In the case of nonlinear problems of similar dimensions, almost all computation time is spent by a Runge-Kutta type method on computing stage solutions. The time overhead introduced by distributing the computational effort to different Maple kernels is not significant when we deal with large IVPs. We cannot expect to obtain similar efficiency results when we use explicit methods, since one integration step involves only a small number of function evaluations and vector operations (see first column of Table 4). The efficiency results are similar to those reported by D-NODE [12] which is based also on Distributed Maple and PVM on a cluster of SGI Octanes (T_p 's reported on such Unix stations using PVM Maple and Distributed Maple are approximately identical). Comparative to the results obtained using EpODE [9], a specialized numerical software for ODEs, in our case an acceptable efficiency level can be obtained for smaller problems due to the fact that T_s and T_p are greater in the case of using a CAS. The limit on problem dimension which can be solved is unfortunately much lower. On other side, solving exactly the stage equations of a Runge-Kutta type method can lead to more accurate solution and an easier error control mechanism.

References

1. Bruder, J., Numerical results for a parallel linearly-implicit Runge-Kutta method, *Computing* **59**, 1997, 139-151. 930
2. Char, B. W., Progress report on a system for general-purpose parallel symbolic algebraic computation, *ISSAC '90*, ACM Press, New York, 1990, 96-103. 927
3. Diaz, A., Kartofen, E., FoxBox: a system for manipulating symbolic objects in black box representation, *ISSAC '98*, ACM Press, New York (1998), 30-37. 927
4. Franco, J. M., Gomez, I., Two three-parallel and three-processor SDIRK methods for stiff initial-value problems, *J. Comput. Appl. Math.* **87** (1997), 119-134. 931
5. Houwen, P. J., Parallel step-by-step methods, *Appl. Num. Math.* **11** (1993), 68-81. 929, 931
6. Hutchinson, D., Khalaf, B. M. S., Parallel algorithms for solving initial value problems: front broadening and embedded parallelism, *Paral.Comp.* **17** (1991), 957-968. 931
7. Iserles A., Nørsett, S. P., On the theory of parallel Runge-Kutta methods, *IMA J. Num. Anal.* **10** (1990), 463-488. 929, 931
8. Lioen, W. M., de Swart, J. J. B., van der Veen, W. A., Test set for IVP solvers, Report NM-R9615, CWI, 1996, available at <http://www.cwi.nl/cwi/projects/IVPtest>. 930
9. Petcu, D., Solving initial value problems with a multiprocessor code, *LNCS* **1662**, Springer (1999), 452-466. 929, 933
10. Petcu, D., Drăgan, M., Designing an ODE solving environment, *LNCSE* **10**, Springer (2000), 319-338. 929, 931
11. Petcu, D., PVM Maple: a distributed approach to cooperative work of Maple processes, *LNCS* **1908**, Springer (2000), 216-224. 926, 929

12. Petcu, D., Numerical solution of ODEs with Distributed Maple, *LNCS* **1988**, Springer (2001), 666-674. [931](#), [933](#)
13. Petcu, D., Experiments with PVM Maple and Parallel Methods for ODEs, *Cluster 2000*, IEEE Computer Society (2000), 393-394. [933](#)
14. Petcu D., Working with multiple Maple kernels connected by Distributed Maple or PVM Maple, Technical Report RISC Series No. 01-18, March 2001. [929](#)
15. Schreiner, W., Developing a distributed system for algebraic geometry, *EuroCM-Par'99*, Civil-Comp. Press, Edinburgh (1999), 137-146. [926](#), [927](#)
16. Shampine L. F., Reichelt, M. W., The Matlab ODE suite, *SIAM J. Sci. Comput.* **18**, No. 1 (1997), 1-22. [930](#)
17. Siegl, K., Parallelizing algorithms for symbolic computation using ||Maple||, *4th ACM SIGPLAN*, ACM Press, San Diego (1993), 179-186. [926](#), [927](#)