Testing of PVODE, a Parallel ODE Solver

Michael R. Wittman

Lawrence Livermore National Laboratory

RECEIVED
DEC 0 9 1996
OSTI

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

UCRL-ID-125562
August 1996

MASTER

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Testing of PVODE, a Parallel ODE Solver*

Michael R. Wittman[†]

August 9, 1996

1 Introduction

The purpose of this paper is to discuss the issues involved with, and the results from, testing of two example programs that use PVODE: `pvkx` and `pvnx`. These two programs are intended to provide a template for users for follow when writing their own code. However, we also used them (primarily `pvkx`) to do performance testing and visualization. This work was done on a Cray T3D, a Sparc 10, and a Sparc 5.

2 The Test Problems

The `pvkx` test problem is one that was used in [1] as an example problem for CVODE [1, 2]. It is a system derived from a one-dimensional diurnal kinetics-diffusion PDE system with two species. The PDEs have the form

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y}\left[K_v(y)\frac{\partial c^i}{\partial y}\right] + R^i(c^1, c^2, t) \quad (i = 1, 2),$$

$$K_h = 4 \times 10^{-6}, \quad K_v(y) = 10^{-8}e^{y/5}, \quad V = 0.001,$$

$$R^1(c^1, c^2, t) = -k_1 c^1 - k_2 c^1 c^2 + k_3(t) \cdot 7.4 \cdot 10^{16} + k_4(t)c^2,$$

$$R^2(c^1, c^2, t) = k_1 c^1 - k_2 c^1 c^2 - k_4(t)c^2,$$

$$k_1 = 6.031, \quad k_2 = 4.66 \cdot 10^{-16},$$

$$k_3(t) = \begin{cases} \exp[-22.62/\sin(\omega t)], & \text{for } \sin(\omega t) > 0, \\ 0, & \text{for } \sin(\omega t) \le 0, \end{cases}$$

$$k_4(t) = \begin{cases} \exp[-7.601/\sin(\omega t)], & \text{for } \sin(\omega t) > 0, \\ 0, & \text{for } \sin(\omega t) \le 0, \end{cases}$$

$$\omega = \pi/43,200,$$

with $0 \leq x \leq 20$ km, $30 \leq y \leq 50$ km, $0 \leq t \leq 86,400$ seconds (one day). The PDEs are discretized by finite differencing on a uniform mesh. Homogeneous Neumann boundary conditions are used. The initial conditions are given by

$$c^i(x,y) = c^i_{scale}(1 - \bar{x}^2 + .5\bar{x}^4)[.75 + .25\tanh(\bar{y}/\delta)] \ ,$$
$$\bar{x} = (x - 10)/10 \ , \quad \bar{y} = (y - 40)/10 \ ,$$
$$\delta = .01 \ , \quad c^1_{scale} = 10^6 \ , \quad c^2_{scale} = 10^{12} \ .$$

For the parallel implementation of the problem, the mesh was split up over a two dimensional array of processors. In fact, the mesh was actually specified by the size of the submesh on each processor, and the size of each dimension of the processor array. The advantage of this approach was the automatic consistency of the size of the data on each processor, which in turn made the code dealing with the data on all processors simpler. The minor disadvantage was the slight inflexibility in the size of mesh that can be used.

Since the problem is stiff, we used the SPGMR module with a block diagonal preconditioner to get our solutions. For the testing, the problem was used as is, with some modifications for writing the mesh to disk and for timing. The specifics of these modifications will be discussed later.

pvnx is a program which models an ODE system based on a non-stiff one dimensional problem similar to pvkx. The testing of this problem was done primarily to verify that the solution was correct and was consistent with the CVODE version of this problem. No visualization was done with this code, but we did need to modify it to do non-blocking communication, which will be discussed in the next section.

3   MPI Communication

One of the problems that became apparent during the testing process was the possibility of deadlocks occurring during MPI communication. Originally in both the pvkx and pvnx problems, all sends and receives were done using MPI_Send() and MPI_Recv(). Because the MPI libraries that we were using (EPCC and MPICH) apparently implemented these calls using buffering, we were not getting deadlocks. However, according to the MPI specification, no buffering is required to be done on these two calls. Thus an MPI_Send() could potentially wait for the corresponding MPI_Recv() before returning, and vice versa. This posed a problem in both pvkx and pvnx, since at certain points in those codes, all processors communicate with their neighbors simultaneously using MPI_Send() and MPI_Recv().

To resolve these problems, we adopted a strategy of first doing non-blocking receives, then blocking sends, then waiting on the non-blocking receives. This strategy is endorsed by the MPI manual, as it makes it more likely that receive buffers will be available when data gets to each processor, thus limiting the amount of system buffering and copying necessary. The call sequence used was:

MPI_Irecv();
MPI_Send();
MPI_Wait();

Using the non-blocking method of communication involved a small amount of additional complexity over the blocking method. This was due to the propagation of one `MPI_Request` handle from `MPI_Irecv()` to `MPI_Wait()` for each communication, as well as the fact that the data in the user buffer given to `MPI_Irecv()` could not be accessed until after the `MPI_Wait()` call.

4   Tests on Sparc 5's and Sparc 10's

Limited testing was done on Sparc 5 and Sparc 10 workstations. The available Sparc 10 workstations were running SunOS 4.1.3, and the available Sparc 5 machines were running Solaris. On the Sparc 10 machines, an existing MPICH installation (version 1.0.11) was used. On the Sparc 5 machines, version 1.0.12 of MPICH was installed from scratch and used.

MPI on workstations is designed to run parallel programs as separate processes on (possibly different) sequential machines as if they were running as separate PE's on a parallel machine. Thus we were able to test both the **pvkx** and **pvnx** parallel programs on our workstations. On the Sparc 10's, the code for both the test programs executed as would be expected, albeit much slower than on a true parallel machine. The code was able to execute as separate processes on the same computer, and also as separate processes on different computers. On the Sparc 5's, the code executed correctly, but was unable to run using more than one process. This is probably due to the fact that the MPI version for Solaris is still in devopment and has not undergone testing.

5   One-PE Version of MPI

To do simple testing of the non-communication aspects of our parallel programs, the one processor MPI library was developed. The main purpose of this library was to allow programs using MPI to compile and run on sequential machines as if they were running as a single process job on a parallel machine. The implementation of this library involved writing functions that mimicked the MPI calls used in our parallel programs and provided the expected results for a single processor run. In addition, some error checking of proper MPI initialization and communication was included in the library. Using this library we were able to compile and run our codes on individual workstations with very little effort.

6   Visualization of Cray-T3D Results

Visualization of results from the **pvkx** code on the Cray T3D was necessary to ensure that the code was indeed correct, and that the problem that the code was solving was neither too difficult nor too easy for the mesh that was used. The actual visualization procedure for the **pvkx** problem used a three-step approach, due to the fact that the amount of data generated

3

by the program was potentially very large, and the fact that the visualization tools necessary (Matlab) existed on a different machine.

The first step in the visualization procedure was the actual writing to disk of the mesh data after each time interval in the problem. This was done by modifying pvkx to have each processor write its own data to a separate file for each time interval. This approach greatly simplified the writing of data to disk, as it did not require the synchronization of processors necessary to write all data to a single file.

The second step in the visualization was the use of the sequential program collect.c on the T3D which collected the data from a number of individual processor data files and merged them into a single larger data file.

The third step was the use of a Matlab script, running on a Sparc 10, to call the collect program on the T3D and get the necessary data for each time interval for each species in the problem. The Matlab program then did a surface plot of the data for both species, one time interval at a time.

Since we were working with large amounts of data at times, we incorporated some flexibility in dealing with the data into each of the three steps of the visualization. This flexibility entailed allowing any of the three steps to limit the amount of data that was seen by the next step to any rectangular subset of the entire processor mesh. Being able to limit the amount of data was absolutely essential for runs of 64 processors or more. (Visualizing the entire mesh for a 256 processor run would have generated 59 megs of data, far beyond Matlab's capabilities for an interactive session.) See Fig. 1 for an example surface plot.

7   Timing of Cray-T3D Results

In addition to visualization, timing and performance testing of the pvkx code was also done. This testing was done over various processor mesh and submesh sizes for the three message passing libraries that we used: SHMEM, MPICH, and EPCC. The main purpose of the testing was to evaluate whether one of the two MPI libraries (MPICH and EPCC) could compete in terms of speed with the Cray-native SHMEM library. The hope was that at least one of the MPI libraries would be nearly as fast as the SHMEM library, since the MPI code has the benefit of being portable.

For the actual timing itself, we used both the native Cray cpused() timing function and the MPI_Wtime() call within the pvkx code itself. The results we got indicated that the values returned by the MPI_Wtime() function were consistent with the cpused() function, and simply different by a matter of a scaling factor. So for our results and analysis, we simply used MPI_Wtime().

The actual timing done was of almost the entire main function of the pvkx code; the only code not executed during the timing was the finalization code which could not be executed before the last MPI call. During the time testing, the pvkx-specific visualization code was "turned off" using a preprocessor switch. This was done in an attempt to avoid effects of disk I/O which might skew the time data.
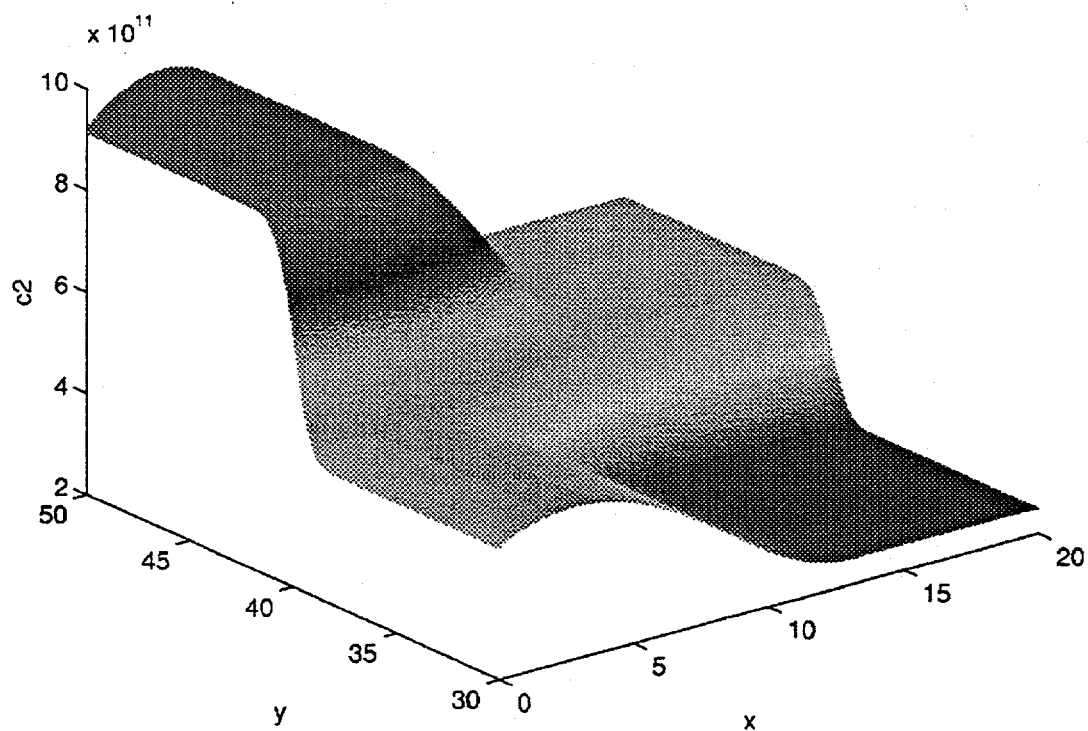
4

c2 – 400x400

Figure 1: Visualization output for a 64 PE run of **pvkx** on a 400×400 mesh.

8 Testing Procedure

The testing procedure for the pvkx code was essentially an edit-compile-run cycle for each test done. The processor submesh sizes as well as the size of the processor mesh and the toggle for mesh printing were all hardwired using #define. In order to do a new test at a different sized mesh, these settings in the pvkx file were changed. The compile was very straight-forward, with the exception that changing libraries (e.g., MPICH to EPCC) involved changing variables in the Makefile. The run was also straight-forward for runs of 32 processors or less, which could be submitted interactively. For 64 to 256 processor runs, we used a C-shell script which simplified the batch queue submission process by creating a batch script on-the-fly and submitting it to the NQS to run later.

For those runs that were for timing purposes, the ouput containing the data for the run was filed based on the run specifications. For runs that were for visualization purposes, the processor data files were kept in a subdirectory, and the Matlab script on the Sparc 10 was run to do the visualization.

9 Test Results

For the pvkx code, the test results were very encouraging. Our trials were done over 64, 128 and 256 processors, in which we either kept the total mesh size constant, or scaled it linearly with the number of processors. In all of these trials, the EPCC implementation of MPI was consistently competitive with the native SHMEM library. The MPICH implementation, however, was considerably slower than either of the other two libraries, and its performance actually appeared to degrade with increasing numbers of processors.

See Figs. 2 and 3 for plots of the performance of the pvkx problem over a variable sized grid. Figs. 4 and 5 show the performance of the problem on a constant size mesh, with a varying number of processors.

References

[1] S. D. Cohen and A. C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, September 1994.

[2] Scott D. Cohen and Alan C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics, vol. 10, no. 2 (March/April 1996), pp. 138-143.
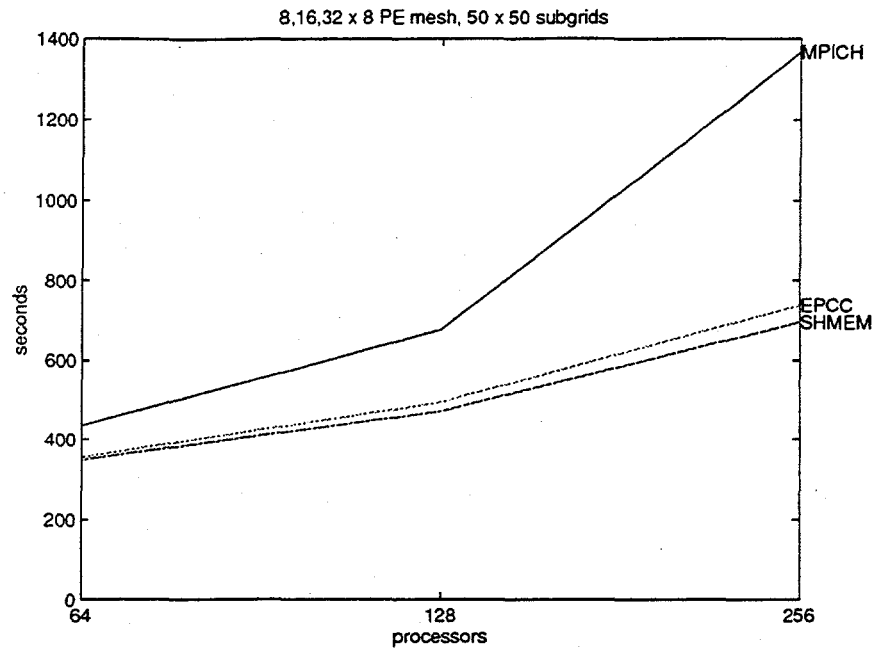
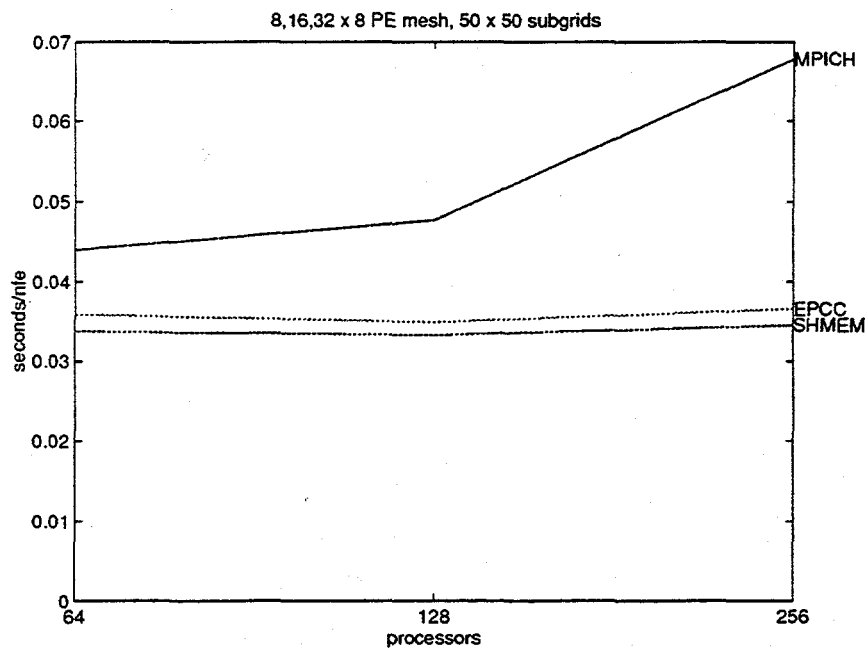Figure 2: Execution time for 50×50 submeshes.

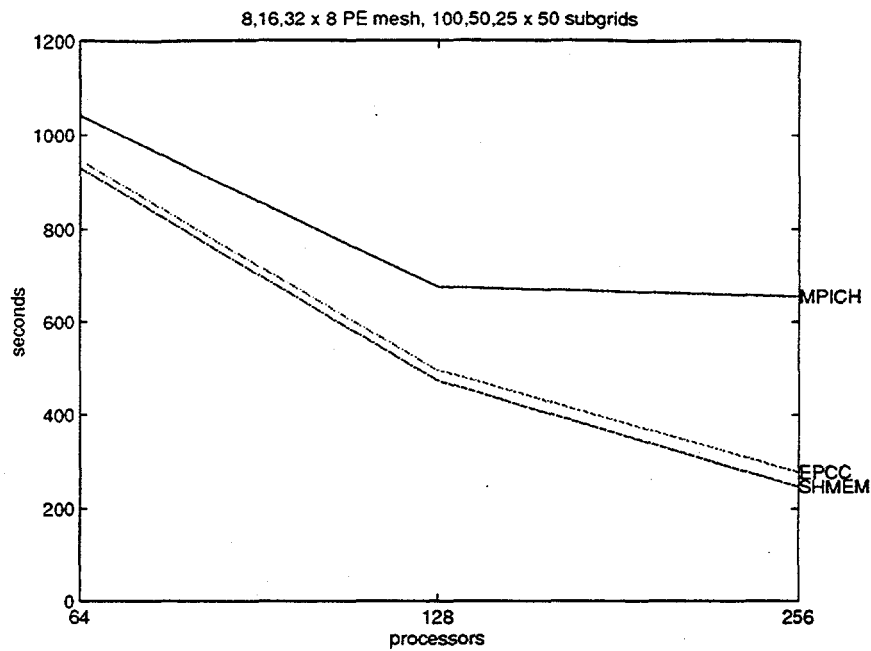Figure 3: Execution time weighted by work done (number of f() calls).

7

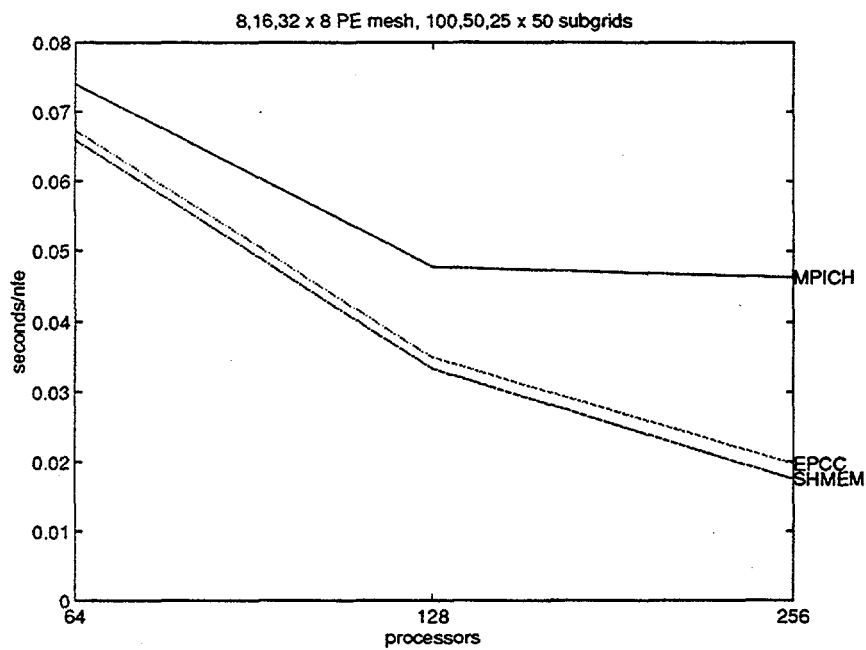Figure 4: Execution time for a fixed 800×400 mesh.

Figure 5: Execution time weighted by work done (number of f() calls).

8