DOE-ET-53088-265 IFSR#265

PARALLEL PROCESSING PARTICLE TRAJECTORY CODE FOR ANOMALOUS TRANSPORT STUDIES

*J. Biswas,** *L. Leonard* and *W. Horton*
Institute for Fusion Studies
The University of Texas at Austin
Austin, Texas 78712

* Department of Computer Sciences

December 1986

Parallel Processing Particle Trajectory Code for Anomalous Transport Studies

J. BISWAS

Department of Computer Sciences

and

L. LEONARD and W. HORTON

Institute for Fusion Studies

The University of Texas at Austin

Austin, Texas 78712

Abstract

The particle trajectory code that calculates the guiding trajectories of electrons in a tokamak taking into account a spectrum of low frequency electromagnetic fluctuations has been reconfigured to run on eight parallel processors. The new code is expressed in Task Level Data Flow Language (TDFL) developed for parallel processing. A data flow graph of the problem is given to describe the split-up and recombination of the data vectors at the central points in the fortran code. A speed up factor of six over single processor computation time is obtained.

Introduction

The particle orbit code (POC) is a Fortran program that has been run extensively on supercomputers (e.g. Cray, CDC Cyber). This document explains the parallel structuring of the particle orbit code in such a way that parallelism can be exploited at the level of Fortran subroutines. The program has been expressed in TDFL[3,4] (Task Level Data Flow Language), a language in which subroutines are treated as nodes of a data flow graph. Computation is realized by the flow of data values (or tokens) through this graph.

What the program does: The program solves a system of differential equations giving the position of a particle in a tokamak containing fluctuations. At the heart of the program is a differential equation solver that computes the final coordinates of a particle given initial conditions and system parameters. There are several well-known numerical analysis techniques to do this, but all of them involve repeatedly calling a function that calculates an approximate derivative based upon the current particle position and global constants for the run. In some situations (for example, in simulation-type problems), such function evaluations may be replaced by table lookups. However, with the POC this is not possible, and we must undertake these expensive function evaluations. The dynamical equation and their properties are given in the Appendix. The important simplifying feature of this program is that there is no interaction between the particles. This feature, coupled with the expensive function evaluation, made this problem attractive from the standpoint of parallelism.

Where the parallelism lies: In analyzing the POC we observe that the program breaks up naturally into three quite independent modules, a control module that generates the initial and final values for each time step of the integration, a module that performs the integration over a vector of particle coordinates (y values), and a module that computes the final statistics. The corresponding TDFL graph of the program is shown in Fig. 1. The arcs show what values are passed along them.

The existing program: The Cray machine on which the POC had been developed, was the MFE (Magnetic Fusion Energy) Cray I at the Lawrence Livermore National Laboratory. This is a single processor machine, in which parallelism is extracted from the POC by vectorizing the steps within the integrator. The integrator used on the Cray was an IMSL library subroutine called DVERK, which is a 5-6 order adaptive Runga Kutta subroutine.

Our approach: We have used a multiple processor machine, the Sequent Balance 8000, which is a twelve processor, shared memory machine. Each of the processors is a 32 bit microprocessor augmented with memory management support and a floating point processor. Each processor can perform roughly 0.75 MIPs. There is no vector capability on this machine, although the memory access requests are pipelined on the shared bus.

Even though the Sequent does not have any special purpose vector hardware by having independent particles proceed in parallel we have exploited available parallelism to our advantage, getting speedups of up to 6 using 8 processors (as shown in Fig. 2). To do the integration, we replaced the IMSL DVERK subroutine with a simpler, less sophisticated subroutine. Our subroutine is a second order accurate predictor corrector (Adams-Bashforth) method that does a fixed amount of work for each interval, and unlike DVERK does not to adapt itself to a faster rate of convergence.

Experiences gained:

Processes and processors: Our measurements were made with the machine in a time-sharing mode, with one DYNIX process being forked for every processor desired. Thus the values of execution time shown are not accurate. We shall be taking accurate measurements in future, with the machine in single user mode. With this understanding, we shall use the terms processors and processes synonomously.

Common blocks and parameters: We broke up the Cray code into a preprocessing phase, during which common blocks and parameters are set up for the entire run. This step is serial, and is done only on one processor, thereby postponing the allocation of the multiple processors until they are really needed.

In the Fortran version of POC there are a number of common blocks that are common to the main program and the function that calculates the derivatives. Since this function is called thousands times per particle, it is better to have such (read only) constant values permanently bound to the function bodies, instead of being passed in and out as parameters with each invocation of the function body. This is an example of a circumstance in which we find it necessary to go outside the purely *value based* paradigm of data flow from efficiency considerations. *Annotated sibling* arcs[3] are another circumstance in which such deviations are useful.

In the TDFL version of POC we have used common blocks to store read only data of a nature that is needed frequently by the function that gets called repeatedly. The tradeoff between using common blocks and passing parameters is a moot one. In our example, it was clearly better to have the common blocks for the 24 or so "read only" parameters to the innermost function; because of the numerous times this function was invoked. However, in a general case it is not obvious that the tradeoffs will be so apparent.

Self-loops and their repercussions:

Side effects: Self-loops provide for state retention between consecutive firings of a node. However, self-loops have the unpleasant property of allowing side-effects. For instance, in the POC code, the lower limit of the integrator gets changed upon return from the subroutine that does the integration. Parallelizing this subroutine with a DOALL node, we had to take care to see that this unwanted side-effect did not interfere with the execution of processes running parallel.

Random number seeds: Using a self-loop arc to pass the seed for a random number generator, causes each parallel integrator to get the same seed. This is undesirable. For this reason, we have to allow for some additional state that is local to each parallel iterate. This is easily implemented by means of *static* declaration in *C*, but entails an alteration in the basic definition of the TDFL model. We have not yet decided whether to make this modification.

Measurements: Timings have currently been taken with a 10 millisecond granularity clock. This makes it difficult to accurately measure overhead. In future we shall have access to a 100 microsecond granularity clock, which will enable the measurement of overhead in our scheduling.

Errors: The major source of error in this problem is truncation error. The error seems to depend more on the number of integration intervals within each time step, than on the precision of the arithmetic. With 32 bits of precision (i.e. *C* language *floats*), and 100 divisions per time step, we got results that compared very well against the *Cray*, which uses 64 bit precision. However, with 20 divisions per time step, the difference was considerable.

Graphics: The Cray code plots resultant outputs on a high speed plotter. On our implementation we have removed this section of the program, dumping the output on to disk files. In future, software may be written that puts numbers in a useful plotted format, for displaying on a Sun workstation window and for printing on a laser printer.

Conclusions: We have demonstrated the feasibility of porting programs from Cray to a multiprocessor machine, using the language TDFL. We have achieved speedups very close to linear. The entire porting took more than three weeks, however, most of this time was spent in learning the application program and debugging errors in the TDFL runtime system. It is expected that a programmer more familiar with his/her code would be able to do the porting faster. The portion of the code that was doing the control had to be modified considerably in order to fit it into the data flow framework. However, the smaller subroutines that did the actual integration were left virtually untouched.

References

1. W. Horton, D-I. Choi, P.N. Yushmanov, and V.V. Parail, "Electron Diffusion in Tokamaks due to Electromagnetic Fluctuations", IFSR#236 (1986) (to be published in Plasma Physics and Controlled Fusion).

2. W. Horton, Plasma Phys. **27**, 937 (1985).

3. J. Biswas, "Parallel Resource Management in Task Level Data Flow", Dissertation Proposal, Department of Computer Science, The University of Texas at Austin, March 1986.

4. P. Suhler and J. Biswas, "Task-level Flow Language, Users' reference manual, Version - 2", Parallel Programming Group, Department of Computer Science, The University of Texas at Austin, (under preparation).

APPENDIX: Guiding Center Equations of Motion

The particle orbit code calculates the motion $x(t)$, $y(t)$ of the electron guiding center across the confining magnetic field $B\hat{z}$ from the Hamiltonian equations

$$\dot{x} = -\partial H(x,y,t)/\partial y \qquad \dot{y} = \partial H(x,y,t)/\partial x. \tag{A1}$$

In the code configuration used in the present tests the motion of electrons trapped in the toroidal magnetic well in the presence of a low order (5×5) $\vec{k}$-spectrum of electromagnetic fluctuations is calculated from Eqs. (A1). The Hamiltonian is derived from the electrostatic fluctuation spectrum

$$\phi(x,y) = \sum_{\mathbf{k}} \frac{A}{k_\perp^{2p+1}} \sin(k_x x + \alpha_{\mathbf{k}}) \cos(k_y y + \beta_{\mathbf{k}}) \tag{A2}$$

where p is the spectral index, $\alpha_{\mathbf{k}}$, $\beta_{\mathbf{k}}$ are random phases and $k_\perp = \left(k_x^2 + k_y^2\right)^{1/2}$.

The plasma physics of deriving the Hamiltonian form (A2) is given in Ref. 1: in the reference case we have

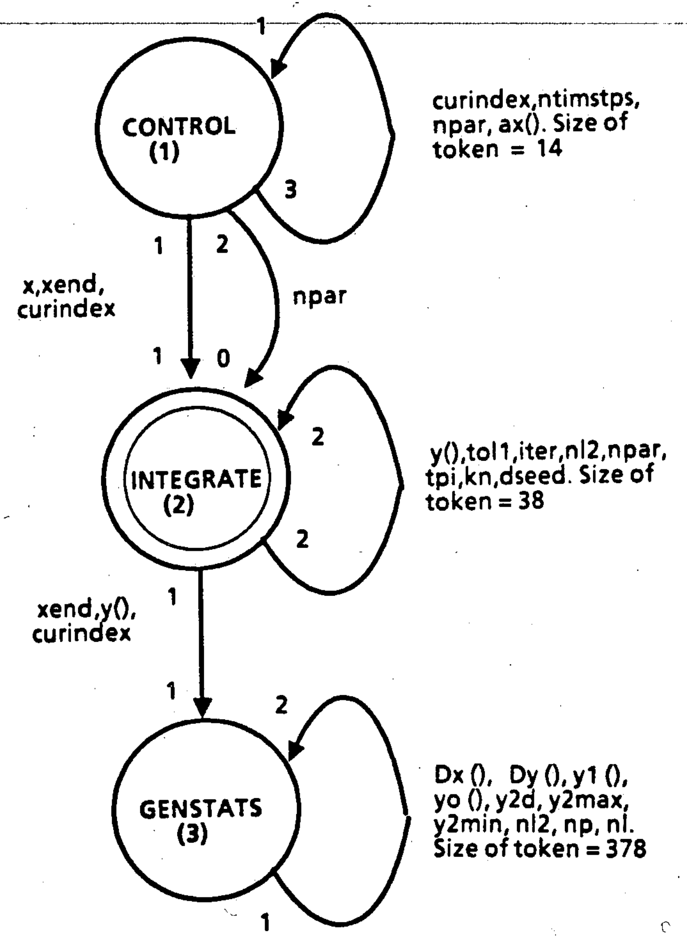
$$H(x,y,t) = \sum_{\mathbf{k}} \left(1 + \frac{\gamma_0 + \gamma_1 \sin t}{1 + k_\perp^2}\right) \phi_{\mathbf{k}}(x,y). \tag{A3}$$

The electron motion changes from integrable for $\gamma_1 = 0$ to stochastic for $\gamma_1 > \gamma_c(A, p, \gamma_0)$ due to the overlapping nonlinear resonances.

The principal statistical quantity required is the radial diffusion coefficients D_x defined as the long-time limit, $D_x = \lim_{t\to\infty} D_x(t)$ of

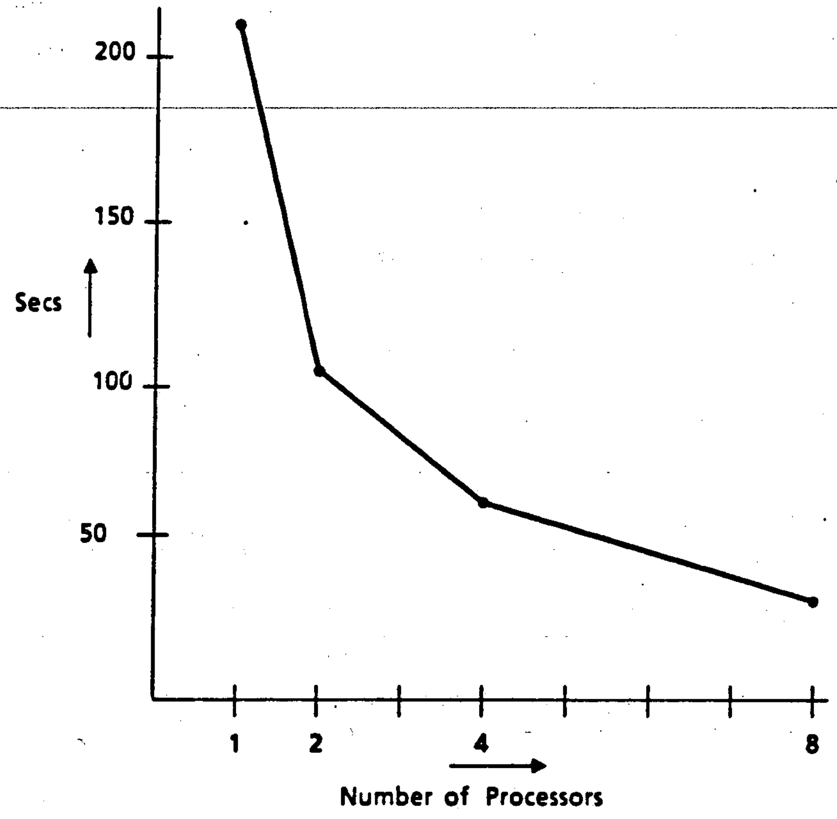
$$D_x(t) = \frac{1}{2tN} \sum_{i=1}^{N} \left(x_i(t) - x_i(0)\right)^2. \tag{A4}$$

Accurate determination of the diffusion coefficient $D_x(A, p, \gamma_0, \gamma_1)$ requires samples with the number of particles $N \sim 10^2 - 10^3$ and the number of periods of integration $N_p = t/2\pi \sim 10^2$ to 10^3 for typical parameters. Other two-time statistical quantities such as $\langle x(t)(t+\tau)\rangle$ and the variance of $D_x(t)$ from $\bar{D}_x$ may be computed by storing a moving time window $\tau_{\max}$ of data for all particles $\{x_i\}_1^N$.



1
CONTROL
(1)
3
curindex,ntimstps, npar, ax(). Size of token = 14
1
2
x,xend, curindex
npar
1
0
2
INTEGRATE
(2)
2
y(),tol1,iter,nl2,npar, tpi,kn,dseed. Size of token = 38
1
xend,y(), curindex
1
2
GENSTATS
(3)
Dx (), Dy (), y1 (), yo (), y2d, y2max, y2min, nl2, np, nl. Size of token = 378
1

Fig 1. TDFL graph for the Particle orbit code



P	Time	Speedup
1	207.20	1.000
2	106.44	1.947
4	56.53	3.665
8	33.71	6.147

Fig 2. Execution Time (secs) and speedups on 8 processors (Note: These timings were taken with the machine in time-sharing mode. Actual times are expected to be smaller.)