Orbitmpi Documentation

Lisa L. Lowe
Princeton Plasma Physics Laboratory
(September 22, 2000)

I. INTRODUCTION

Orbitmpi is a parallelized version of Roscoe White's Orbit code [1,2] . The code has been parallelized using MPI, which makes it portable to many types of machines. The guidelines used for the parallelization were to increase code performance with minimal changes to the code's original structure. This document gives a general description of how the parallel sections of the code run. It discusses the changes made to the original code and comments on the general procedure for future additions to Orbitmpi, as well as describing the effects of a parallelized random number generator on the code's output. Finally, the scaling results from Hecate and from Puffin are presented. Hecate is a 64-processor Origin 2000 machine, with MIPS R12000 processors and 16GB of memory, and Puffin is a PC cluster with 9 dual-processor 450 MHz Pentium III (18 processors max.), with 100Mbits ethernet communication.

II. CODE DESCRIPTION

A. Orbitmpi

The general idea of Orbitmpi is that a number of particles are distributed to each processor and their trajectories are calculated in the main loop until each particle has either achieved its maximum toroidal transits or until it is absorbed by the wall. During the run, the magnetic field is assumed static and the particles do not interact with each other. This means that no communication is required between processors during the essential calculations, making Orbit an ideal candidate for parallelization. The main communications consist of a broadcast of information to each processor at the beginning of the code, and a gather routine after the main loop.

The code begins with a call to 'initialize_mpi' which initializes MPI and opens a file for each process to use for write statements to stdout. The master process then opens files and initializes NCAR graphics. The code calls 'parallel_switch' to determine whether or not the code will be run in parallel. The conditions for parallelization for Orbitmpi are that nplot is not 1 or 6 (1 has one particle and 6 skips the main loop) and that the number of particles is greater than the number of processors. The first condition cannot be changed, since there are no gather routines written for those choices of nplot (also there is no reason they need to be run in parallel). The second condition can be changed as desired, as the parallelization is not useful for small numbers of particles.

The master process reads in the numerical equilibrium and broadcasts the values to the other processes. It then does the equilibrium plots. All processes call the different subroutines to make sure all values are initialized until right before the particle deposition, then 'find_nprt' is called. The subroutine 'find_mynprt' assigns an equal number of particles to each process and ensures the remainder is evenly distributed. The subroutine also defines 'firstprt' and 'lastprt', which are the first and last values of the original array index that that processor holds. For example, if there are 10 particles and 3 processors, process 0 (the master process) will have mynprt=3,firstprt=1, and lastprt=3. Process 1 will have mynprt=3,firstprt=4,lastprt=6 and process 2 will have mynprt=4,firstprt=7,lastprt=10.

The next subroutine called sets the initial configuration of the particles. Each processor only initializes its own set of particles according to the choice of distribution (ndist). Most of the initialization routines use a random number generator. The consequences of using a random number generator in a parallel code are discussed in section IV. Three of the distributions in Orbitmpi (ndist = 11,13,14) read in values from a data file. Since only one processor can read in this file, the master process only does the initialization of particles and then it 'scatters' the particles to the appropriate processor (in subroutine 'scatterdist').

After this initial distribution, the master process makes a plot of the initial distribution and must gather the appropriate values from the other processors (unless ndist is 11,13,or 14). The subroutine 'gatherpdist' does this gathering.

The main loop follows. These main subroutines consist of do loops over the particles, from 1 to nprt, so after nprt is set for each processor in the previous routines there were very few changes necessary to parallelize the main loop. After the loop is finished, the variables are gathered on to the main processor in the subroutine 'gather'. The variable 'nprt' is gathered from the processors in case of ejected particles, nsteps is set to the highest value of nsteps, and if eject is used then time(1) is set to the time it takes for the particle that took the highest number of nsteps to finish. After this gather, the final graphs are plotted.

B. Orbitmpi3d

The Orbitmpi3d code is structurally almost identical to the Orbitmpi code with a few exceptions. In the condition for parallelization, the minimum number of particles to run can be changed, but there must be at least one particle per processor. This is because pspline routines complain if there are no particles. Other differences are inherent from the original differences in the code: different variables are needed to broadcast and gather, nplot=6 is parallelized because it does go through the loop, etc. There are no large differences between the codes, but the scaling of Orbitmpi3d is not as good as that of Orbitmpi because of the need to broadcast the large 3D spline data to all the processors (VI).

III. CHANGES AND NEW FILES

The new files added to Orbit are orbitmpi.f, mpicheck.f, and gathersubs.f. Orbitmpi.f contains the initialization and broadcast subroutines and a subroutine "parallel_switch" that determines whether the run will be in parallel or not. It also contains "find_mynprt", a subroutine that finds how many particles should be placed on each processor. Orbitmpi.f contains the scatter and gather routines used to give the master process the correct values for plotting the initial distribution. Mpicheck.f contains some routines that are useful for debugging, but in general unnecessary for running the code. Gathersubs.f contains the subroutines used to gather all of the values needed for the final plots.

The original routines have been altered slightly to accommodate the parallelization. In the main routine (orbit.f or orbit3d.f), there are calls to the above mentioned parallelization routines and added 'if' statements for when only the master processor should perform the subroutines, etc. The Makefile has been altered to compile the new files and to use MPI libraries. New global variables have been added to the common block files. Some of the subroutines have been modified slightly to accommodate differences that arise from either using the array index in an equation or by using 'nprt0' instead of 'nprt'. For two detailed examples of this, see V. The random number generator subroutine ranx in file 'ranff.f' has been modified in order to give a different seed to each processor. One major change that has been applied to each routine is that each processor must have its own output file, and hence, all previous 'write(6,*)' statements have been replaced by 'write(myfile,*)' statements.

There is also a perl script needed to put together the output files from each processor. Most of the main output file, 'orbout', is written out by the master processor in subroutine 'wrt6'. The other processors will write out the information on the lost particles. The perl script takes this lost particle data from each file and inserts it into the proper location in the orbout file and deletes those temporary files.

IV. RANDOM NUMBER GENERATOR

The original Orbit initializes a seed value in the beginning of the code. In order to ensure each processor does not have this same seed, thereby producing so many copies of the same pseudo-random numbers, when ranx is initialized each processor increments the original seed by its process number. This is perfectly acceptable for the physics of the problem, but poses a debugging dilemma- the code will give different answers for different numbers of processors. The code does use a portable random number generating routine so that different machines using the same number of processors and the same seed will get the same answers. In order to debug certain aspects of the code it may be desired to get the same answers for different processors, and in that case an initial distribution that does not use random calls can be used. Alternatively the ranx subroutine can be replaced by the original (with one seed only), and the following loop may be inserted in the beginning of the particle initialization routines:

```
if(myid.gt.0) then
   do i= 1,firstprt-1
      dummy = ranx()
   end do
end if
```

V. FUTURE REVISIONS

If the code is to be modified, there are only a few items that must be kept in mind in order to keep consistent with the parallelization. If more graphs are added or different variables are considered in the final plots, the gather subroutines must be altered. This is easy enough: for instance, if a new plot is added for nplot=2 and it plots rho, the subroutine "gather_2" must be altered by adding a block of code that gathers the variable 'rho'. It should be pointed out that the gather subroutines currently gather only those variables necessary to make the appropriate plots and output files for a given nplot.

Since each particle is independent from the others, in general there is no problem adding more code without worrying about the parallelization. The exceptions are when the index of a loop is used in the equation and when there are sums over the variables from 1 to nprt. For example, in rcrd0, k0 is recorded. This had to be modified from:

```
do k=1,nprt
   k0(k) = k
end do
```

to:

```
do k=1,nprt
   k0(k) = k + firstprt -1
end do
```

Similar modifications can be done if the array index is used in an equation. When there are sums taken on each processor, that value must be gathered at the end of the loop. For example, in rcrd9 there is a sum that is divided by the number of particles:

```
zv(kplt) = dum/nprt
```

This had to be changed to

```
zv(kplt) = dum/nprt0
```

the intention of this was to divide by the total number of particles. Also, in the gather subroutine the values of zv for each processor are added together with 'MPI_REDUCE' in order to get the correct value.

VI. SCALING

The scaling shown in the following graphs show the speed up for increasing number of processors, plotted with the "ideal" speedup- twice as fast for twice as many processors. The actual numbers are a comparison of the parallelized code to itself, not between the original code and the parallelized version, although there is no significant difference between the run time of the original code with the new code using one processor. The deviation from ideal scaling is due to two major things- the communication time and the IO time. The master processor must processes all of the plots. This plotting will take a certain time regardless of the loop speedup and will increase with an increase in particles. The second issue is communication. The master process must gather and broadcast values to each process, so this time increases with the number of processors and with the number of particles. The effect of this communication is more dramatic in Orbitmpi3d, because the 3D spline must be broadcast to each processor and this is a very large amount of data. The following scaling runs are from nplot=5 and ndist=1.

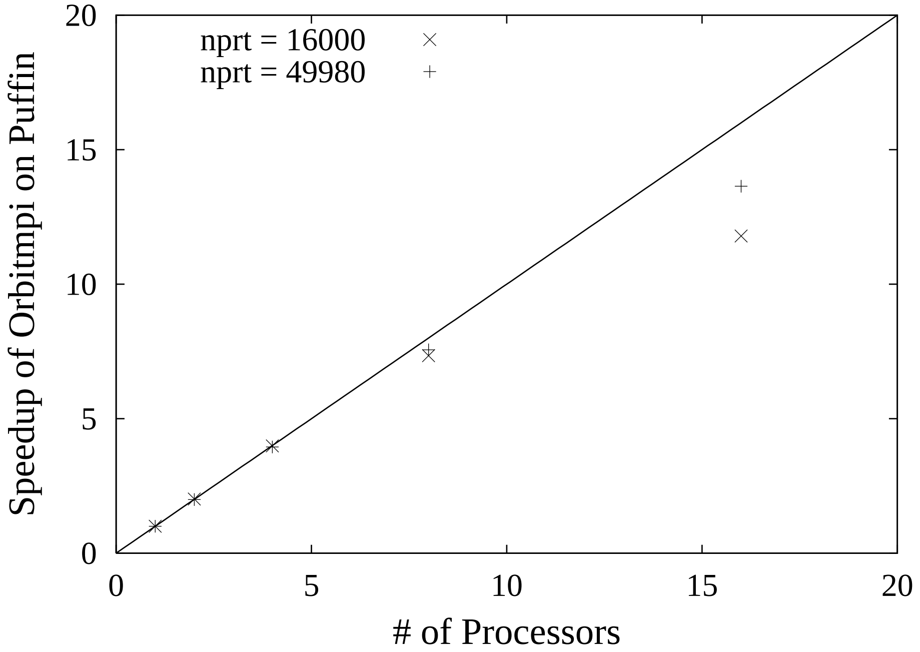

FIG. 1. Orbitmpi: Speedup on Puffin- nplot=5 and ndist=1 for 16000 and 49980 particles

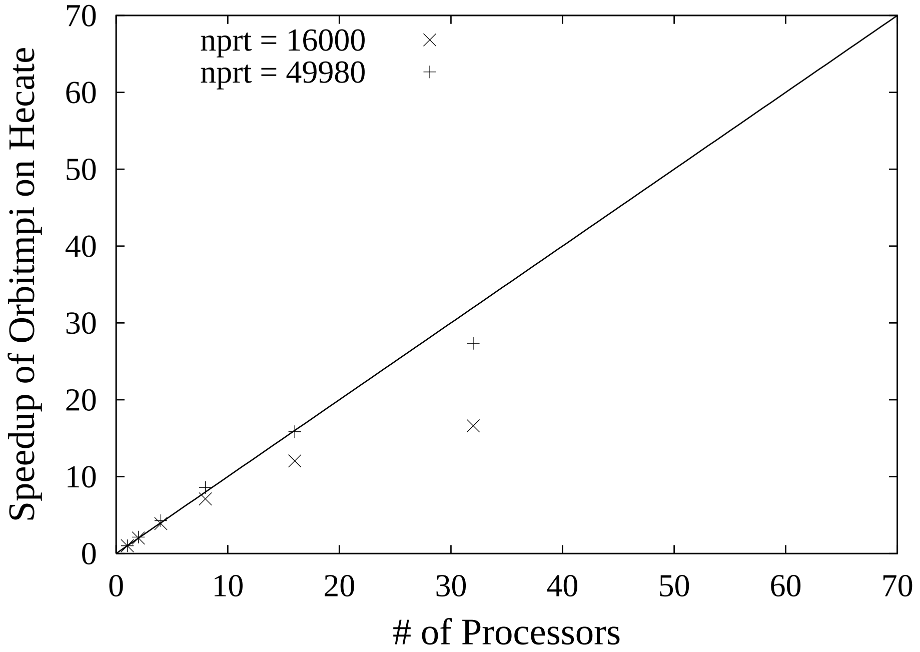

FIG. 2. Orbitmpi: Speedup on Hecate- nplot=5 and ndist=1 for 16000 and 49980 particles

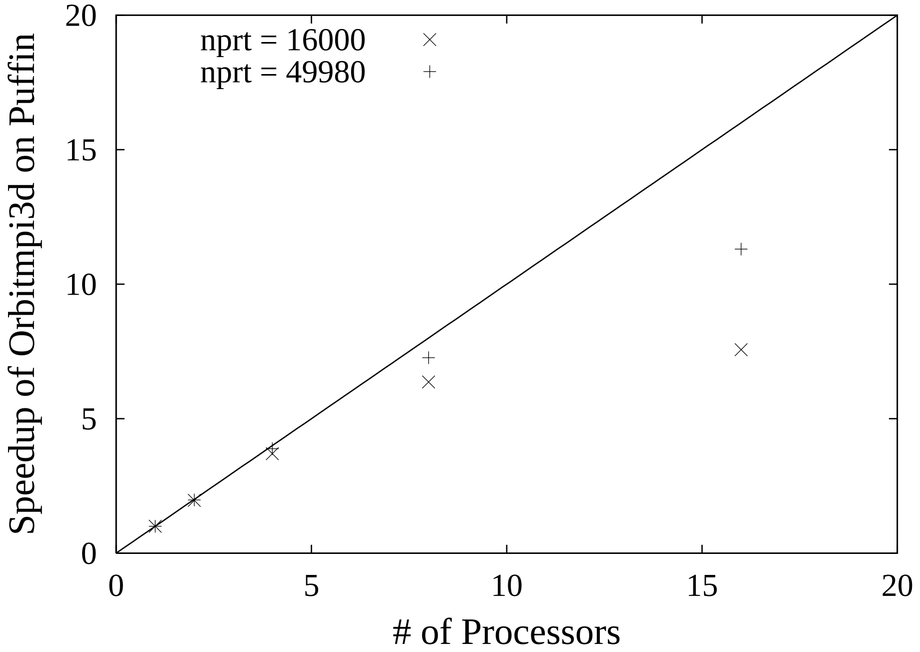

FIG. 3. Orbitmpi3d: Speedup on Puffin- nplot=5 and ndist=1 for 16000 and 49980 particles

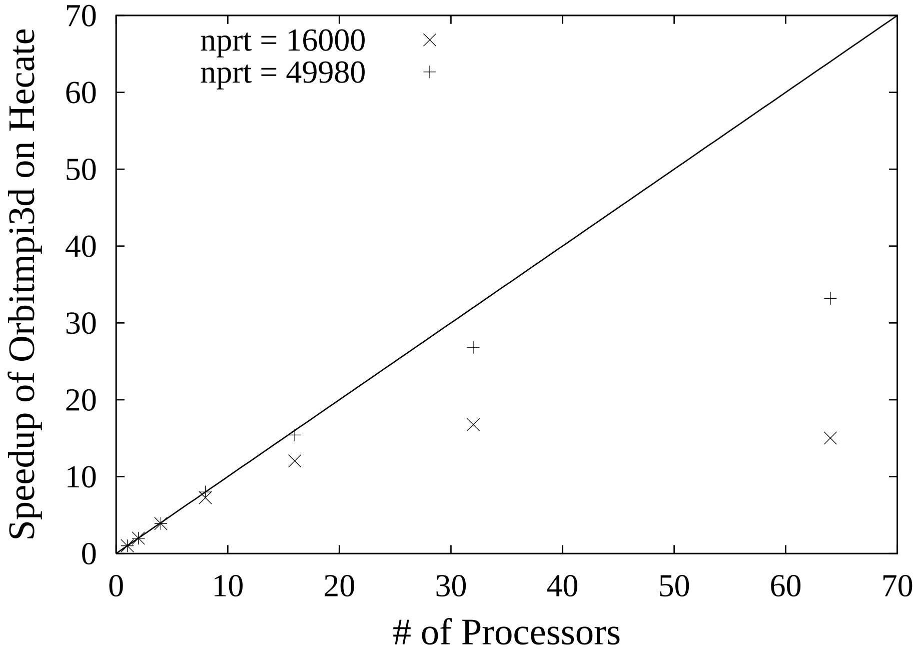

FIG. 4. Orbitmpi3d: Speedup on Hecate- nplot=5 and ndist=1 for 16000 and 49980 particles

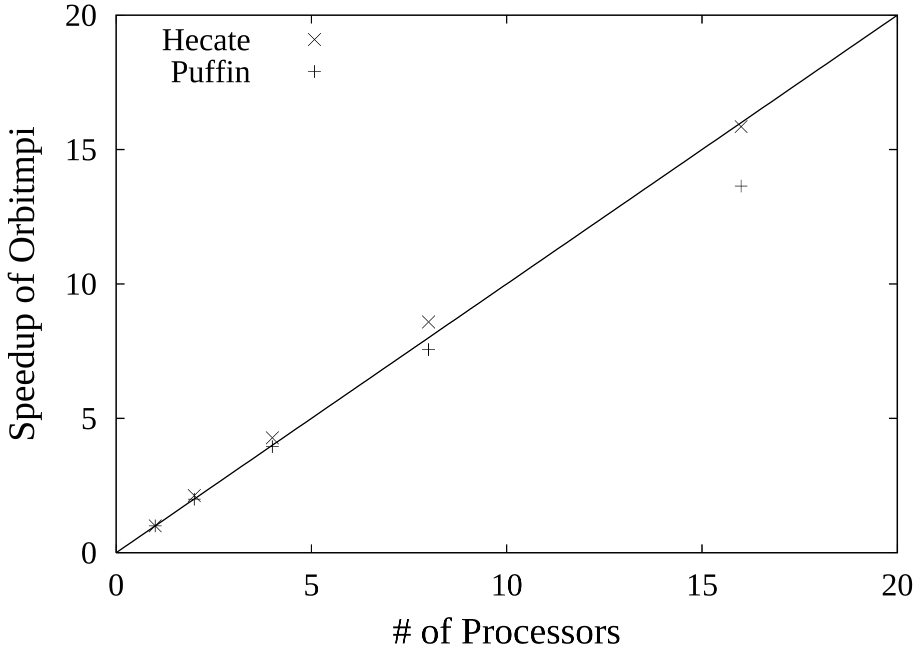

FIG. 5. Orbitmpi: Speedup on Hecate and Puffin for 49980 particles

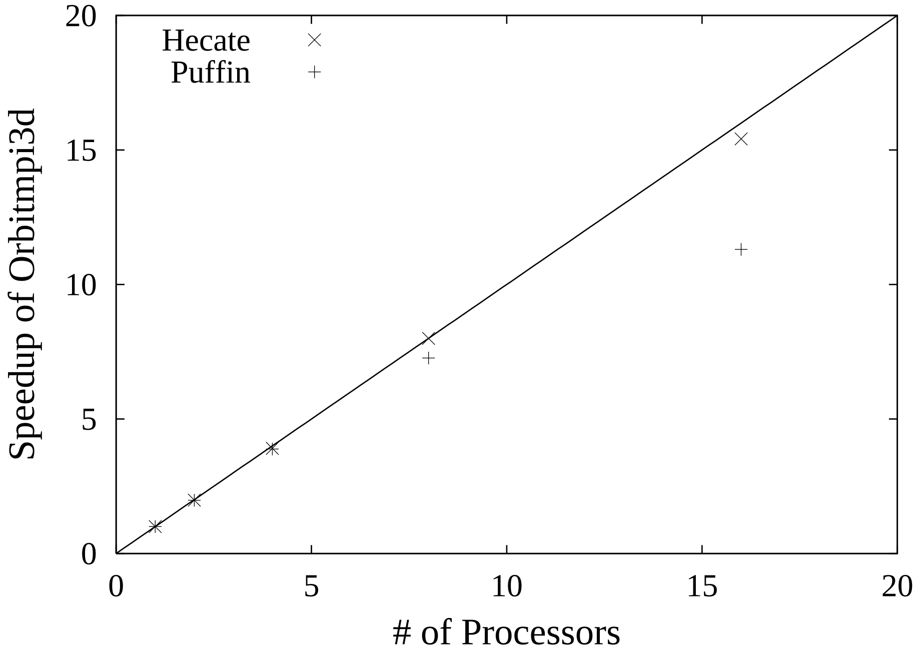

FIG. 6. Orbitmpi3d: Speedup on Hecate and Puffin for 49980 particles

Below are the tables for the actual run time in seconds for Orbitmpi and Orbitmpi3d on Puffin and Hecate. For Orbitmpi with 49980 particles, the time for a run on Puffin decreases from 17.4 minutes on one processor to 1.3 minutes on 16 processors. On Hecate this time decreases from 5.5 minutes to 12.1 seconds.

Puffin 49980 particles		Puffin 16000 particles	
Processors	Time (seconds)	Processors	Time (seconds)
1	1041.129	1	329.265
2	521.570	2	163.337
4	263.705	4	82.563
8	137.715	8	44.839
16	76.320	16	27.927
Hecate 49980 particles		Hecate 16000 particles	
Processors	Time (seconds)	Processors	Time (seconds)
1	329.8179	1	95.492
2	154.976	2	47.434
4	77.1536	4	24.566
8	38.412	8	13.445
16	20.8025	16	7.930
32	12.0586	32	5.393

TABLE I. Run Time in Seconds for Orbitmpi

Puffin 49980 particles		Puffin 16000 particles	
Processors	Time (seconds)	Processors	Time (seconds)
1	1610.140	1	514.871
2	811.819	2	261.968
4	414.307	4	139.113
8	221.569	8	80.876
16	142.453	16	68.075
Hecate 49980 particles		Hecate 16000 particles	
Processors	Time (seconds)	Processors	Time (seconds)
1	795.152	1	238.695
2	400.866	2	119.302
4	203.348	4	61.266
8	99.450	8	32.872
16	51.597	16	19.819
32	29.648	32	14.239
64	23.959	64	15.886

TABLE II. Run Time in Seconds for Orbitmpi3d

The scaling improves with the number of particles, and the parallelized versions of Orbit are not very useful for a small number of particles particularly on machines with less than superb communications. For Orbitmpi on Puffin, the scaling starts to approach expected values for between 1000 and 5000 particles. For Orbitmpi3d the scaling begins to improve between 5000 and 10000 particles.

VII. LOAD BALANCING

The issue of load balancing is important to optimize the performance of a parallel code. Most of the initializations for orbit are random distributions, so the current system of simply making sure that each processor has an equal number of particles gives decent load balancing. Some distributions, however, are not random but based on angles increasing with a loop index, such as ndist=9 and 12 from Orbitmpi3D. Such distributions give poor load balancing when distributed in contiguous blocks to each processor. For example, all of the particles on processor 8 may be at an angle facing the wall and therefore exit the calculation almost immediately, while the particles on processor 1 may have a steady orbit in the middle of the machine and continue for some time. Such distributions are better handled by the master process doing the initialization and then distributing the particles in a cyclic fashion rather than in blocks.

Other distributions that involve reading in data from a file may also be biased in certain ways, and it is possible that even in random distributions there may be more particles getting lost or hitting the wall early. In further development of the code, subroutines that categorize the phase space of each particle and distribute them among the processors accordingly could be written for each initial distribution. This version of the Orbit code has been deliberately constructed to deviate as minimally as possible from the original code, and also to ensure that the results were exactly the same as the original code during the testing process. Because of this, the previous suggestions for load balancing optimizations are discussed but not implemented. If load balancing is indeed a concern for future work, it must be handled on a case by case basis. The following analysis has found that while some load balancing techniques improve the performance of the code significantly, in other cases the load balance insures only that each processor goes about as slow as the slowest processor. The time it may take for the code to do load balancing analysis at the beginning of the initialization will also be of importance.

The following load balancing data is from nplot=5 with various distributions. The statistical variables are defined as follows: The run time is the time in seconds that each process takes to complete the main loop. The mean value μ is defined as the sum of the run times (T) for each processor divided by the total number of processors N:

$$\mu = \frac{\Sigma T}{N}.$$

The variance V is

$$\mathrm{V} = \frac{\Sigma(T-\mu)^2}{N}$$

and the standard deviation is

$$\sigma = \sqrt{V}.$$

The run time from the processor that took the longest is given in the table as L. The deviation of L from the mean value and that deviation relative to the mean value are also given in the tables.

ndist	μ	σ	L	L $-\mu$	$\frac{\mathrm{L}-\mu}{\mu}$
1	15.586	0.295	16.226	0.640	0.041
2	9.512	0.402	10.188	0.676	0.071
4	22.409	0.238	22.876	0.467	0.021
5	19.967	0.563	20.827	0.086	0.043
7	48.287	0.776	49.447	1.160	0.024
10	2.946	0.052	3.037	0.091	0.031
14	9.775	0.210	10.091	0.316	0.032

TABLE III. Load Balancing for Orbitmpi, 16000 particles

ndist	μ	σ	L	L $-\mu$	$\frac{L-\mu}{\mu}$
1	54.904	0.745	56.399	1.495	0.027
2	32.138	0.831	33.763	1.625	0.051
4	80.240	1.047	81.331	1.091	0.014
5	69.873	1.399	70.958	1.085	0.016
7	169.601	2.467	173.514	3.913	0.023
10	10.477	0.212	10.739	0.262	0.025
14	34.454	0.686	35.743	1.289	0.037

TABLE IV. Load Balancing for Orbitmpi, 49980 particles

The next two tables show the results from Orbitmpi3d. The particle numbers are different because in ndist=9, the number of particles must be divisible by 60 (ntheta) in order to ensure all particles are initialized properly. The distributions 9 and 12 have distributions based on increasing theta(9) or pitch angle(12), and therefore have poor load balancing with the original distribution routine that distributes the particles in contiguous blocks. This data from this routine is shown in the tables as 9,1 and 12,1. A test subroutine was created to test the load balancing if the particles were distributed so that each processor had a similar set of particles. This was done by the the master process doing the initialization and then distributing the particles in a cyclic fashion to each processor. The data from the test routine is given by 9,2 and 12,2. For both cases, this test subroutine improved the load balancing significantly, however, for ndist=9, there was no significant increase in performance. As illustrated by comparing L in the tables, the load balancing for ndist=9 only makes each processor go equally slow. For ndist=12, the average time to complete the loop increased from 12,1 to 12,2, but the longest run time L significantly decreases, giving better performance with better load balancing.

ndist	μ	σ	L	L $-\mu$	$\frac{L-\mu}{\mu}$
1	30.701	0.519	31.435	0.734	0.024
3	16.008	0.290	16.285	0.277	0.017
4	27.410	0.751	28.910	1.500	0.055
5	29.088	0.779	30.198	1.110	0.038
6	20.711	0.396	21.360	0.649	0.031
10	15.561	0.102	15.767	0.206	0.013
11	29.518	0.475	30.209	0.691	0.023
9,1	13.027	2.268	15.172	2.145	0.165
9,2	14.510	0.262	15.101	0.591	0.041
12,1	9.090	3.133	11.686	2.596	0.286
12,2	9.915	0.137	10.043	0.128	0.013

TABLE V. Load Balancing for Orbitmpi, 16320 particles

ndist	μ	σ	L	L $-\mu$	$\frac{L-\mu}{\mu}$
1	101.400	2.259	103.398	1.998	0.020
3	53.098	1.252	54.250	1.152	0.022
4	92.009	2.179	94.042	2.033	0.022
5	96.834	2.201	100.198	3.364	0.034
6	69.125	1.464	70.416	1.291	0.019
10	53.360	0.548	54.027	0.667	0.013
11	98.949	2.019	100.743	1.794	0.018
9,1	42.823	7.476	49.855	7.032	0.164
9,2	47.823	0.931	49.073	1.25	0.026
12,1	29.683	10.190	37.855	8.172	0.275
12,2	32.197	0.658	32.674	0.477	0.015

TABLE VI. Load Balancing for Orbitmpi, 49920 particles

ACKNOWLEDGMENTS

I would like to thank Stephane Ethier for his assistance and advice on parallelizing the code and for running the code on Hecate to provide the scaling and load balancing data.

[1] R. B. White and M. S. Chance, Phys Fluids **27, 2455** (1984).
[2] R. B. White, Phys Fluids B 2, **845** (1990).