

ORNL Accelerator Physics Memo

Parallel Computing with ORBIT

8/17/1999

John Galambos

ORNL

Spallation Neutron Source Project  
Oak Ridge National Laboratory  
PO Box 2009 MS 8218  
Oak Ridge TN 37831

## Parallel Computing with ORBIT

This note serves to describe the initial parallel computing implementation of the ORBIT code and show some timing results. The bulk of ORBIT is parallelized, using PVM as the message passing mechanism. Tests have been performed on the SNS Accelerator Physics Wonderland cluster.

### I. General Parallel Structure

It is now possible to operate ORBIT on parallel machines, using a message passing parallel implementation. ORBIT is a particle tracking code for rings, and incorporates a programmable interactive driver shell. Serial runs are set up by writing small scripts (or programs) as the input file. The parallel implementation approach taken here is to start a run just as a serial run, by reading an input file. The initial process started serves as the parent. The parent process spawns multiple additional ORBIT processes on different CPUs, which each parse the same input script file (see Fig. 1), and perform the “same tasks”. In this sense, each process running on a parallel node acts as a serial job.

#### Input Script File for Parallel Runs

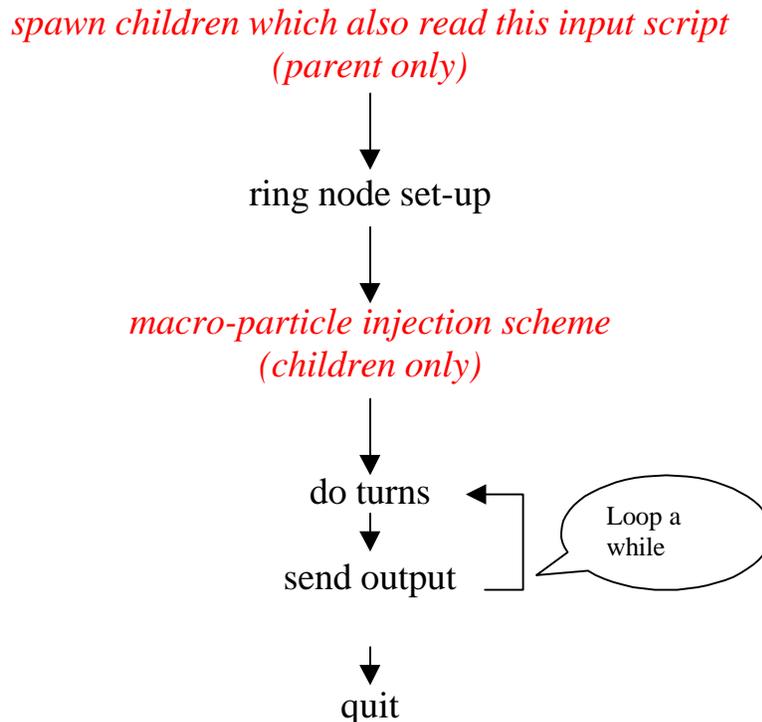


Fig 1 General input file script flow for a typical parallel run. Parts of the input script which would be different from a serial run are highlighted in red italics.

The data decomposition scheme is extremely simple with this implementation, macro-particles are never shifted from one processor to another. Most of the calculation on each parallel node proceeds independently just as if it was a stand-alone serial calculation. There are several exceptions however, which require communication between the parallel processes, namely:

- Initialization of the random number seed used to generate macro-particles from prescribed distributions (so that each processes has a unique macro-population)
- Space charge calculations
- Diagnostic calculations (e.g. overall beam moments)
- Output

Other calculations such as lattice matrix advances, aperture checks, thin lens kicks, etc. have no need for parallel communication and proceed the same as for a serial run. As such, most modules in ORBIT are unchanged in the new parallel implementation. The message passing has been coded only where needed in the above mentioned parts. The most critical message passing occurs in the transverse space charge section. As multiple herds are tracked in parallel around the ring on different processors, at each space charge kick node, the collective force from all the particles must be gathered and communicated between nodes. Typically this happens 100's of times per turn. The strategy taken to parallelize this calculation is shown in Fig. 2. Note that each child calculates its own FFT of both the Greens function and the global charge distribution<sup>1</sup>. This is faster than waiting for one processor to do it, and passing the results<sup>2</sup>. Also note that the Greens function FFT is done in-between the gather and scatter of the global charge distribution, in order to hide as much latency from this scatter/gather as possible. This calculation arrangement was arrived at for the SNS "Wonderland" beowulf cluster using 533MHz alpha chip processors and a 100 Mbs switched private network for communication. Using systems with faster communication/(processor time) ratios may make it worthwhile to calculate the FFT of the charge distribution on the parent in parallel while the children calculate the Greens function FFT, and then pass the calculated charge distribution to the children.

The actual message passing was done with the PVM package [2] here. The MPI package may be substituted in the future. Details of the use of ORBIT for parallel runs are discussed in the User Manual [3].

---

<sup>1</sup> We repeatedly FFT the Green's function because our grid is not fixed throughout the calculation. It is allowed to grow as the particle emittances grow.

<sup>2</sup> Passing the charge distribution involves only ¼ of the grid points since the grid extends 2x's the particle extent to avoid false aliasing. Also passing the FFT output involves 2x's the number of grid points (real & complex).

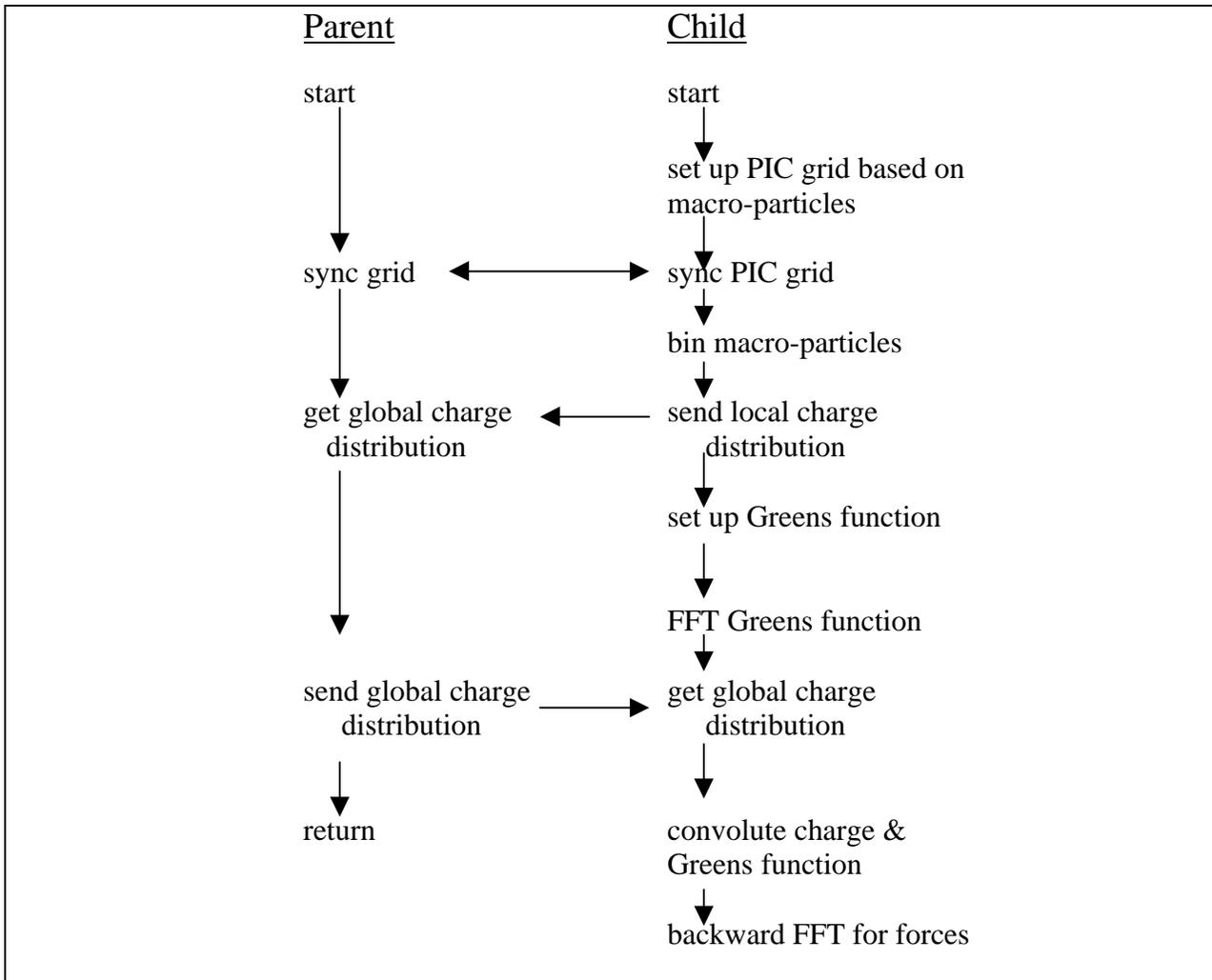


Fig. 3.2 Parallel flow logic for the transverse space charge calculation.

### Example Runs

#### Constant macro-particle number, varying CPU number

As a first example we compare the results of the same calculation performed with varying number of processors. This calculation consists of injecting 100,000 particles and tracking them 10 turns. The injected particles are sampled from a gaussian distribution with  $\epsilon_{RMS} = 30 \pi$ -mm-mrad, and truncated at  $240 \pi$ -mm-mrad (both horizontal and vertical). Fig. 3 shows the resultant distributions after 10 turns for the cases of (1) a serial calculation, (2) a parallel calculation using 2 processors, and (3) a parallel calculation using 4 processors. The distributions are very similar, and differences at the level  $< 0.01\%$  are due to the small number of particles (10) in this region and the use of different random numbers in the particle initialization sampling for these cases. This demonstrates that the space charge parallel algorithm calculates the same distribution as the serial method. Also shown is the injected distribution tracked without space charge. The change in the distribution due to space charge is noticeable.

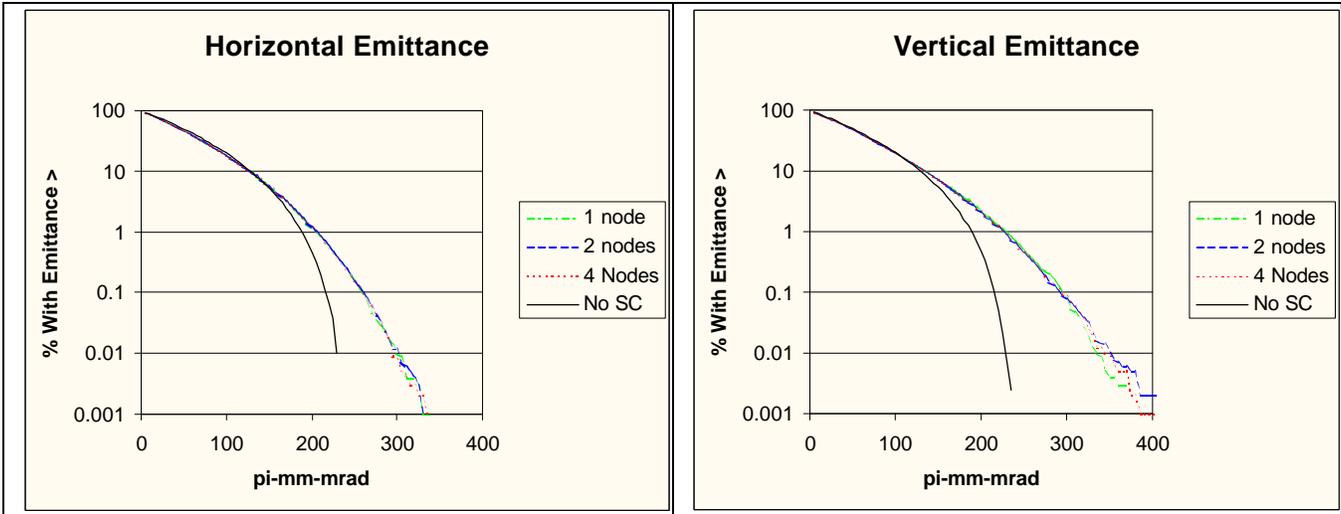


Figure 3. Comparison of emittance distributions calculated with serial and parallel algorithms.

Constant CPU number, varying macro-particle number

As another example, we show the impact of using more macro-particles, on the final painted emittance. This case is for a full injection scenario into the SNS ring for a case in which the painting scheme is optimized to minimize growth beyond the collimator acceptance ( $180 \pi$ -mm-mrad), and to also produce fairly flat profiles for the target. The calculation involves tracking for 1158 turns, used 480 azimuthal steps/turn, and used a  $64 \times 64$  grid for the transverse space charge. The closed orbit bump scheme here is programmed to paint particles below  $115 \pi$ -mm-mrad. Figure 4 shows the resultant horizontal and vertical emittance distributions. The legend numbers refer to the final number of macro-particles accumulated at the end of the run. The 105k macro-particle case is a serial run (used  $\sim 19$  hrs CPU time). The 210 k, 300k and 1000k macro-particle cases are parallel runs using 4 processors and took 11.7, 15.5 and 52 hrs respectively to run. Running 1,000,000 macro-particles with a serial calculation is impractical, from a run turn-around time perspective. It is evident that the space charge has caused halo growth well beyond the injection limits. For the 1,000,000 macro-particle case the horizontal distribution begins to show resolution at the  $10^{-4}$ - $10^{-5}$  level, where beam spreading has occurred.

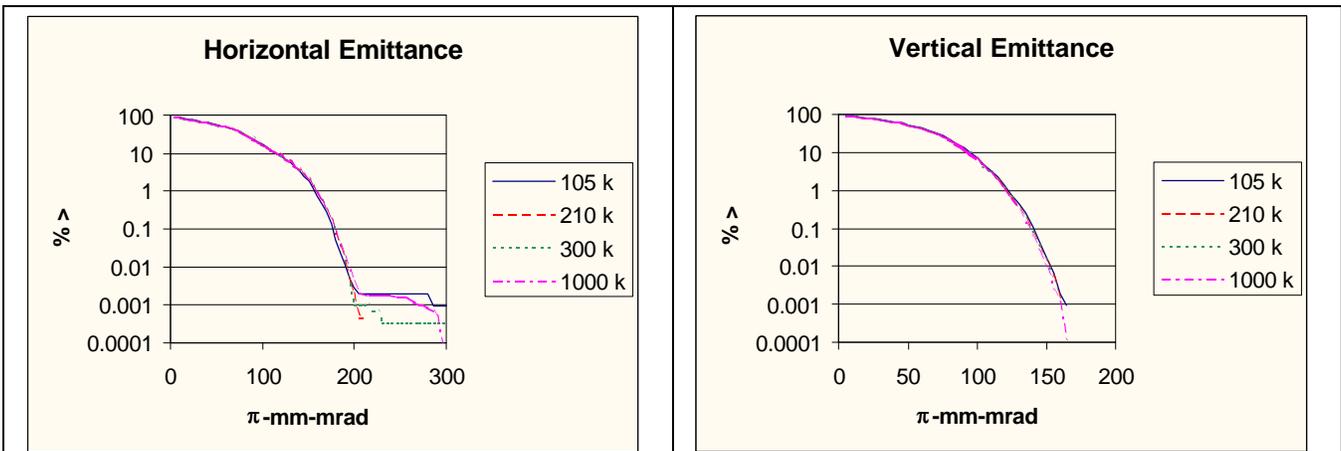


Figure 4. Impact of increased number of macro-particles for a complete injection scenario. The 210k, 300k and 1000k macro-particle cases are parallel runs.

## Timings

First some simple timings were done to study the parallel efficiency. These runs were performed on the SNS Wonderland Beowulf cluster using 533 MHz alpha chip processors, connected by a 100 Mbs switch on a private network. The parallel efficiency is defined to be:

$$h_{//} = \frac{t_{serial}}{t_{//} \times N_{cpu}},$$

where  $t_{serial}$  is the serial CPU time, and  $t_{//}$  is the (wall clock) time to do the same calculation in parallel with  $N_{cpu}$  parallel processing CPUs. We investigate the sensitivity of this efficiency to the number of macro-particles used and to the FFT grid as shown in Table 1. Figure 5 shows the timing results for the 128x128 transverse grid size example. Also the cases for 1 CPU are serial run counterparts. This example calculation consists of pushing a macro-particle herd for one turn, using 480 lattice elements/turn. Only linear lattice elements are used here, and there is one longitudinal space charge update included. As can be seen, the parallel computation is > 90% efficient if there are > 10-20 particles/cell.

With fewer particles, the calculation becomes less efficient because the computation spends relatively more time in inter process communication and in calculating the FFT. We investigate these inefficiencies by artificially masking certain parts of the transverse space charge calculation, and seeing the impact of computation time. These runs do not produce physically realistic results, but do shed light on which parts of the parallel implementation are using the most time. An inefficient case from Table 1 is used for this example (128x128 grid with 125k macro-particles and 4 cpus). First we artificially mask the synchronization of the grid used across CPUs (i.e. skip this part of the parallel calculation). This has a minimal impact on the total computation time, indicating that this message passing is not causing significant parallel inefficiencies. Next we mask the synchronization of the charge distribution across the CPUs. This has a large impact, increasing the parallel efficiency from 61% to 78%, i.e. eliminating almost half of the parallel inefficiency. Finally we mask all the message passing in the transverse space charge calculation, and note that the parallel efficiency increases to 80%. There is still a 20% inefficiency for this case, compared to the “ideal time” derived by dividing the serial CPU time by the number of processors used. This residual inefficiency is inherent in the parallel calculation method used; namely each processor calculates the FFT, and is spending a larger fraction of its time performing the FFT calculation, rather than pushing the macro-particles, as the parallel case has fewer macros /CPU than the serial case. Thus even if the inter-process communication is infinitely fast, there is still an inherent parallel inefficiency associated with the parallel implementation adopted here. But the scheme appears to be fast enough to benefit use of modest sized clusters (say < 16-32 nodes) for cases with large numbers of macro-particles.

Table 1. Sensitivity of the parallel computation efficiency to the transverse grid size, the number of macro-particles and the number of CPUs used.

<u>64 x 64 grid</u>			<u>128 x 128 grid</u>		
<u>N-cpu</u>	<u>time</u> <u>(sec)</u>	<u><math>\eta_{//}</math></u>	<u>N-cpu</u>	<u>time</u> <u>(sec)</u>	<u><math>\eta_{//}</math></u>
<u>200 k macros</u>			<u>1 M macros</u>		
1	226	1.000	1	1303	1.000
2	124	0.91	2	701	0.93
4	59	0.96	4	348	0.94
7	47	0.69	7	220	0.85
<u>100 k macros</u>			<u>500k macros</u>		
1	124	1.000	1	610	1.000
2	62	1.000	2	359	0.85
4	36	0.86	4	204	0.75
7	30	0.59	7	124	0.70
<u>50k macros</u>			<u>250 k macros</u>		
1	54.2	1.000	1	312	1.000
2	36	0.75	2	192	0.81
4	21	0.65	4	104	0.75
7	23	0.34	7	83	0.54
			<u>125 k Macros</u>		
			1	160	1.000
			2	94	0.85
			4	66	0.61
			7	66	0.35

Table 2. Impact of various components of the transverse space charge message passing on the parallel inefficiencies. The case with a 128x128 grid, 125k macros, and 4 CPUs is used.

Case	Time (sec)	<u><math>\eta_{//}</math></u>
Full parallel calculation	66	0.61
Mask the TSC grid synchronization	63	0.63
Mask the charge distribution synchronization	51	0.78
Mask all transverse space charge message passing	50	0.8
Ideal parallel case	40*	1.0

\* this is simply taken to be the serial CPU time/4

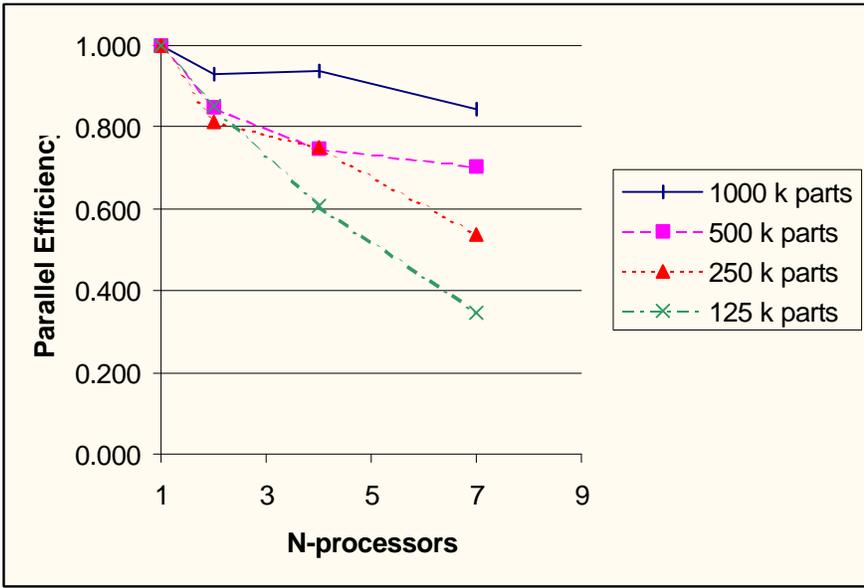
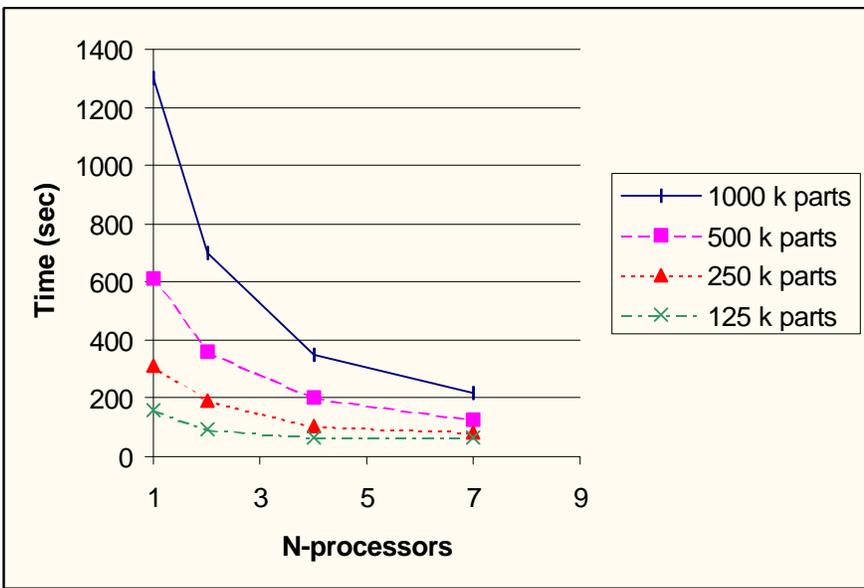


Fig. 5 Timings and efficiencies for a single turn using a 128x128 transverse grid and 480 steps per turn. Note the poor parallel performance with < 250,000 macro-particles.

## References

- 1- ORBIT - A Ring Injection Code with Space Charge , J. Galambos, S. Danilov, D. Jeon, J. Holmes, D. Olsen, ORNL, Oak Ridge, TN; J. Beebe-Wang, A. Luccio, BNL, Upton, NY, PAC99, <http://ftp.pac99.bnl.gov/Papers/Wpac/THP82.pdf>
- 2- Parallel Virtual Machine, see <http://www.epm.ornl.gov/pvm/>
- 3- ORBIT User Manual – see <http://www.ornl.gov/~jdg/APGroup/Codes/Codes.html>