Pelegant: A parallel accelerator simulation code for electron generation and tracking

Y. Wang and M. Borland

Advanced Photon Source, Argonne National Laboratory, Argonne, IL 60439 USA

Abstract. elegant is a general-purpose code for electron accelerator simulation that has a worldwide user base. Recently, many of the time-intensive elements were parallelized using MPI. Development has used modest Linux clusters and the BlueGene/L supercomputer at Argonne National Laboratory. This has provided very good performance for some practical simulations, such as multiparticle tracking with synchrotron radiation and emittance blow-up in the vertical rf kick scheme. The effort began with development of a concept that allowed for gradual parallelization of the code, using the existing beamline-element classification table in elegant. This was crucial as it allowed parallelization without major changes in code structure and without major conflicts with the ongoing evolution of elegant. Because of rounding error and finite machine precision, validating a parallel program against a uniprocessor program with the requirement of bitwise identical results is notoriously difficult. We will report validating simulation results of parallel elegant against those of serial elegant by applying Kahan's algorithm to improve accuracy dramatically for both versions. The quality of random numbers in a parallel implementation is very important for some simulations. Some practical experience with generating parallel random numbers by offsetting the seed of each random sequence according to the processor ID will be reported.

Keywords: Computer modeling and simulation, particle accelerators, parallel computing
PACS: 07.05.Tp, 29.27.-a

INTRODUCTION

Pelegant stands for "parallel elegant," which is a parallelized version of elegant [1]. Written in the C programming language with MPICH [2][3], Pelegant has been successfully ported to several clusters and supercomputers, such as the "weed" cluster (a heterogeneous system of 100 CPUs) at the Advanced Photon Source (APS), and the Jazz cluster (350 Intel Xeon CPUs) and the BlueGene/L supercomputer (1024 dual PowerPC 440 nodes) at Argonne National Lab (ANL). Thanks to careful design in parallelization and good architecture of the serial elegant, Pelegant achieves very good performance. For example, for a simulation of 10^5 particles at the APS, including symplectic element-by-element tracking, accelerating cavities, and crab cavities, the simulation time was reduced from 14.3 days to 42 minutes on 512 CPUs of the BlueGene/L (BG/L) supercomputer. The speedup for this particular simulation was 484 with efficiency near 95%.

Here we report on the parallelization strategy and numerical issues for validating the parallel implementation against the serial program. Detailed information about how to install Pelegant, run the code, and optimize the performance can be found at [4], which also includes a list of parallelized elements in one of the appendices.

PARALLELIZATION STRATEGY

We parallelized `elegant` using the master/slaves (manager/workers) model. The time-intensive tracking parts of `elegant` are being parallelized gradually. The other parts are done (redundantly) by all the processors, which is acceptable since those processors have already been allocated to a particular `Pelegant` run. We divide the beamline elements into four classes:

1. Parallel element: only the slave processors will do the tracking. Each slave is responsible for a portion of the particles.
2. MP (multiprocessor) algorithm: the master will participate in the tracking, but it only gets the result of collective computations (e.g., sum, average) from the slaves, without doing any computations itself.
3. Uniprocessor element: must be done by master (for now) and modifies particle coordinates. An example for the present version of `Pelegant` would be wakefield elements.
4. Diagnostic: same as the uniprocessor element, but doesn't change particle coordinates. An example would be the `WATCH` element (which variously analyzes particle distribution or dumps raw particle data to a file).

A flag was added to `elegant`'s dictionary for each beamline element to identify its classification. The master is responsible for gathering and scattering particles as needed according to this classification. By adding the flag to the elements, communications can be minimized to achieve the best efficiency of parallelization. For example, it is not necessary to communicate the coordinates of particles between master and slaves when tracking through two continuous parallel elements. Similarly, we only need to gather particle coordinates from slaves to master (without subsequent scattering) when the particles go through a diagnostic element, such as a `WATCH` point with coordinate output.

In our master/slave model, the master is responsible for I/O operations and communicating with the slave processors only, i.e., it will not do the tracking for most of the elements. To run simulations efficiently, we also suggest that the user arrange all serial elements in a continuous sequence when possible, which will minimize the communication overhead for gathering and scattering particles. This will be unnecessary in the future when all of the elements are parallelized.

By default, `Pelegant` is built in such a way that it does load balancing after each pass through the accelerator. This is particularly important when the user does not have exclusive use of the nodes. When running `Pelegant` in an environment where only one user is allowed to run a job on a computer node at a time, `Pelegant` can be built (using a compiler flag) to redistribute particles (workload) only if the particle number is changed. `Pelegant` also has the ability to allocate workload according to the real speed of a processor. This is crucial to achieve best performance on a cluster consisting of CPUs of different speeds, such as APS's weed cluster.

Figure 1 shows, in a crab cavity simulation, that `Pelegant` achieves very good efficiency, which is defined as the the speedup divided by the number of processors. For example, if the number of processors is less than or equal to 512, the efficiency is more

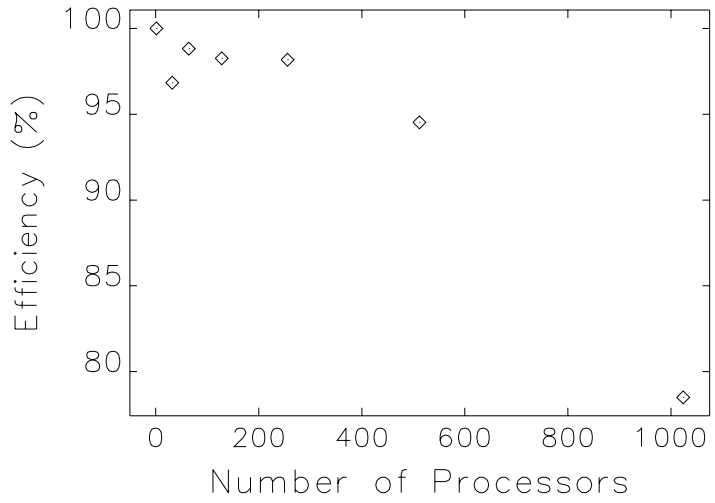

FIGURE 1. The efficiency of `Pelegant` for the crab cavity simulation (100,000 particles) on the BlueGene/L supercomputer at ANL.

than 90%. For 1024 processors, the efficiency goes down to 78% due to communications overhead and serial I/O bottlenecks.

NUMERICAL ISSUES FOR VALIDATION

Validating a parallel program against a uniprocessor program with the requirement of bitwise identical results is notoriously difficult [5]. Some of the problems we may meet include different ordering of summations and non-scalable random number generators. We ran a regression test of nearly 100 cases and validated the results of `Pelegant` with the serial `elegant`. Discrepancies are not uncommon in our comparison due to the reasons above. Although the simulation results with the discrepancies should conform to IEEE 754 within some tolerance, more consistent results can be expected with more accurate numerical algorithm, such as Kahan's summation formula [6], which has been employed in both serial and parallel versions of `elegant`. Random numbers also play an important role in `elegant`. In our regression tests, some of the parallelized beamline elements involve simulations with random numbers. We report our experience of implementing a parallel random generator in the second part of this section.

Kahan's Summation Formula

As the computers we use today have finite precision, the rounding errors of floating-point arithmetic operations can not be avoided. For those elements needing collective operations, e.g., a very long sum, the results of `Pelegant` and `elegant` could be

different. This is because the floating-point arithmetic is not associative, and the parallel version will not keep the same order of operations as the serial version. This makes validation by comparing the result numerically very hard. Figure 2 illustrates the difference between these two versions.

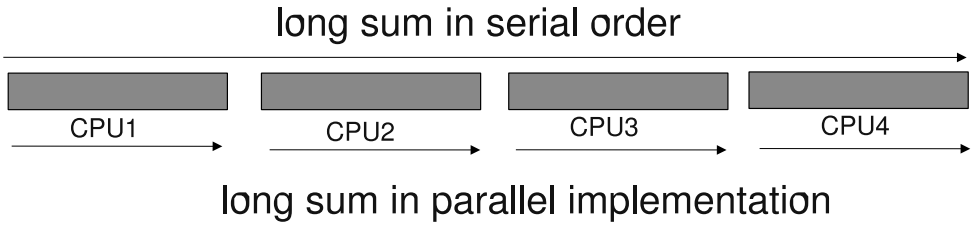

FIGURE 2. Different order for a long sum between parallel and serial versions.

One possible solution to get better agreement is to improve the accuracy of both versions. We implemented Kahan's summation formula [6] for the very long sum operations in the simulations. The numerical results of parallel and serial versions then agree very well. Suppose that we want to compute $\sum_{j=1}^{j=N} X_j$: with the traditional summation algorithm where we just add the summand one by one, the error could be as bad as $N\varepsilon\sum_{j=1}^{j=N}|X_j|$, where ε is the machine precision. For double precision, $\varepsilon = 10^{-16}$. The N here can be understood as the number of simulation particles, which in `elegant` can be very large, thus making the results inaccurate.

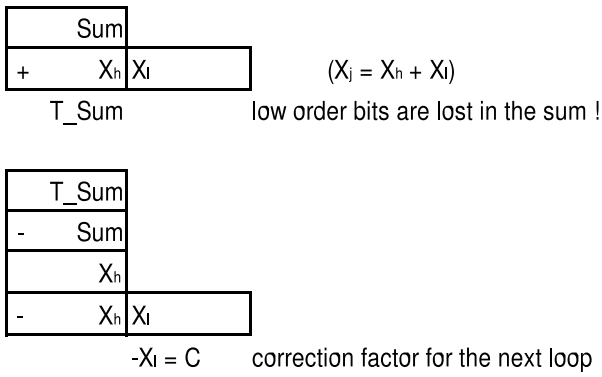

FIGURE 3. An intuitive explanation of the Kahan's algorithm.

In contrast, the error bound for the Kahan summation formula is just $2\varepsilon\sum_{j=1}^{j=N}|X_j|$. The formal proof of this error bound can be found in [6]. An intuitive explanation of how the Kahan summation formula works is illustrated in Figure 3. We consider each summand X_j as two separate parts: the high-order bits X_h and the low-order bits X_l. When a summand is added to the *Sum*, the low-order bits will be lost because of limited machine precision. To recover the low-order bits, we first compute the high-order bits by subtracting *Sum* from *T_Sum*. Then $X_j(= X_h + X_l)$ is subtracted from X_h to get the lost low bits, which will become the correction factor for the next loop. There is not much difference between the parallel and serial versions of Kahan's formula, except the

correction factor for the last step on each processor should be collected together as the correction factor of the final result.

Figure 4 shows the discrepancies of average momentum in the cases with and without Kahan's algorithm. From the solid line in the figure, we can find that the results of `Pelegant` and `elegant` are almost identical when Kahan's algorithm is applied, while a discrepancy of 10^{-9} is found without Kahan's algorithm. Although the difference is trivial for this one-turn simulation, the accumulated errors after thousands of turns could be very large, which could not only cause inaccurate numerical results, but would also make the validation procedure (against serial version) extremely difficult. The application of Kahan's algorithm in both `elegant` and `Pelegant` improves the accuracy significantly and makes the validation procedure much easier.

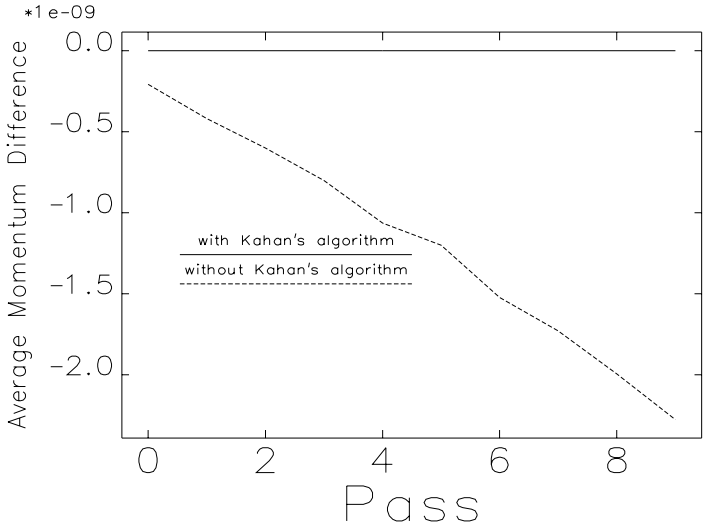

FIGURE 4. The discrepancies of average momentum with and without Kahan's algorithm in a crab cavity simulation.

Parallel Random Generator

A scalable random number generator is very important for correct parallel implementation. "Poor random number generators are like bugs in a software program," says David Ceperley, a physicist at NCSA and the University of Illinois at Urbana-Champaign. "They can distort your results and sometimes cost you hours of computing time." There are several parallel random number generator packages available. One of these, the Scalable Parallel Pseudo Random Number Generators Library (SPRNG) [7], has been widely used in the literature because of its high quality. Although it is convenient to take advantage of an available parallel random generator, we would like to parallelize the existing random generator in the serial `elegant` because of the special requirements for this particular parallel implementation:

1. There are four different random sequences in the simulation software. Some of

them are required to have the same sequence on different processors, for example the random generator for the input beam or for magnet errors. Others need to have different sequences on different processors, for example the random generator for elements that scatter the beam.

2. The random number sequence for the serial version should not change, in order to make it convenient for verifying the result with the previous versions of `elegant` numerically.
3. A simple interface was needed that did not require too many changes in the original structure of `elegant`.

As a result of these considerations, we did not use SPRNG. However, we choose SPRNG as the reference to evaluate the quality of our own parallel random number implementation.

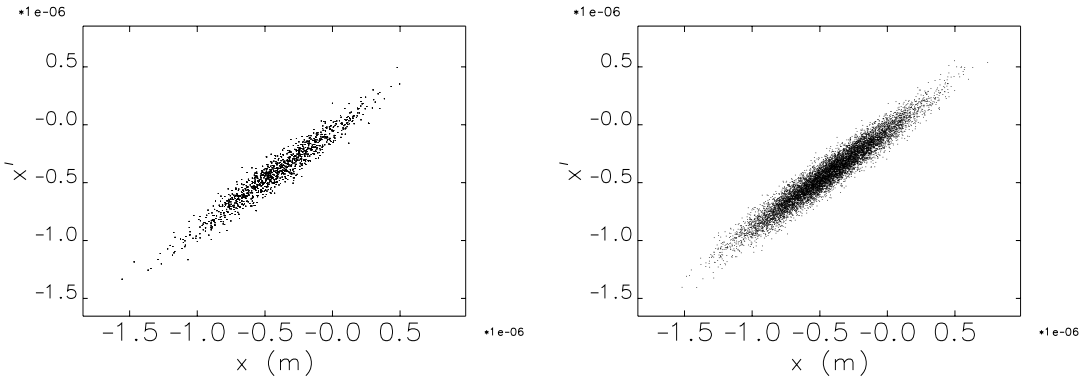

FIGURE 5. The phase space after a canonical kick sector dipole magnet on ten processors. Picture on the left: without a parallel random number generator; picture on the right: with a parallel random number generator.

If one naively converted a serial code into a parallel one, the same random number sequence would be repeated across all the processors. The result is the same as if they had run the job on a single processor with a much smaller sample size. This is because all of the random generators have the same seed, which is usually given by the user or the developer. For a parallel implementation, we need to find the appropriate way to make different processors have different seeds. To satisfy the requirement that the seed of the random number generator must be a large odd number, we offset the seed of all the slave processors by two times its ID number from the seed of the master processor. This makes all the processors have different random sequences without too many changes in the interface. Figure 5 shows the phase space after particle tracking through a canonical kick sector dipole magnet with quantum excitation. The picture on the left is the case where all of the processors repeat the same random sequence. The picture on the right is the result when our parallel random generator was used. The phase-space result from `Pelegant` with a parallel random generator agrees very well with the result of `elegant`. We also did some tests comparing the correlation of the different random sequences generated on different processors with the SPRNG package. The correlation coefficients in our implementation are between 1e-3 and 1e-2, which is

the same order as the SPRNG package. In a two-dimensional visualization result, we observed both parallel random sequences distributed uniformly in the specified area.

CONCLUSIONS AND FUTURE WORK

Many of the time-intensive elements in `elegant` were parallelized, in particular, those that do not involve collective interaction of particles. This reduced the simulation time for some practical problems very significantly. The parallel version produces results that conform to the tested and established serial version. This goal was achieved by applying Kahan's algorithm in both `elegant` and `Pelegant` to improve accuracy, as well as by making a correct implementation of a parallel random generator appropriate to `elegant`'s architecture. `elegant`'s extensive regression test suite was used for this purpose. Several physicists at APS have benefited already from the current evolution of `Pelegant`. A public version will be available for download at: http://www.aps.anl.gov/Accelerator_Systems_Division/Operations_Analysis/software.shtml. In the future, we will continue to parallelize elements for the simulation of collective effects, including wakefields and resonant impedances. We also plan to take advantage of parallel I/O in MPI-2 [3] to improve the performance for multi-million particle simulations.

REFERENCES

1. M. Borland, "elegant: A Flexible SDDS-Compliant Code for Accelerator Simulation," Advanced Photon Source LS-287, September 2000.
2. W. D. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Process, Cambridge, MA, 1999.
3. W. D. Gropp, E. Lust, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Process, Cambridge, MA, 1999.
4. Y. Wang, M. Borland, and R. Soliday, "User's Manual for `Pelegant`," 2006. http://www.aps.anl.gov/Accelerator_Systems_Division/Operations_Analysis/oagSoftware.shtml.
5. W. D. Gropp, "Accuracy and Reliability in Scientific Computing," Chapter in *Accurate and Reliable Use of Parallel Computing in Numerical Program*. SIAM, 2005.
6. D. Goldberg, "What every computer scientist should know about floating-point arithmetic," ACM Computing Surveys, 23(1):5–48, March 1991. http://docs.sun.com/source/806-3568/ncg_goldberg.html.
7. "Scalable Parallel Pseudo Random Number Generators Library," http://sprng.cs.fsu.edu/.