Nuclear Mathematical and Computational Sciences: A Century in Review, A Century Anew
Gatlinburg, Tennessee, April 6-11, 2003, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2003)

DESIGN, IMPLEMENTATION AND TESTING OF MERCURY: A PARALLEL MONTE CARLO TRANSPORT CODE

Richard Procassini, Janine Taylor, Ivan Corey and John Rogers
Lawrence Livermore National Laboratory
Mail Stop L-95, P.O. Box 808, Livermore, CA 94551
spike@llnl.gov, taylorj@llnl.gov, ircorey@llnl.gov and rogers9@llnl.gov

ABSTRACT

The design, implementation and testing of a parallel Monte Carlo transport code is presented. This MERCURY code models the transport of neutrons, gammas, five light ions (and eventually electrons) through a variety of mesh types. A key requirement that has driven the design and development of MERCURY has been the ability to run on various massively parallel computing platforms. We have adopted a three-pronged approach to parallelism: spatial parallelism via domain decomposition, and particle parallelism via domain replication and task decomposition. Both distributed-memory message passing and shared-memory threading programming techniques have been employed. This approach allows us to model particle transport through very large meshes or geometries (spatial parallelism), as well as very large particle counts (particle parallelism). Following a detailed discussion of the parallel implementation of MERCURY, we assess the parallel performance of the code on two criticality problems. The parallel efficiency of the code has been determined for runs involving either spatial parallelism or particle parallelism, and for a limited set of mixed parallelism runs. This study was performed by varying the processor count for a constant particle work load, as well as a constant particle work load per processor, to determine the code's speedup and self-speedup, respectively.

Key Words: Monte Carlo transport, parallel computations

1. INTRODUCTION

Monte Carlo transport methods have long been considered "fertile ground" for parallel computations. These methods have often been referred to as being in the class of "embarrassingly parallel" algorithms. The reason for this statement is that since each particle history is treated independently of other particles, parallelization is achieved by simply dividing the particle work load across multiple processors: *particle parallelism.* Once the histories have completed, the results (tallies or edits) are then globally summed across all of the processors. This method of parallelization has been used by a number of researchers [1–3].

This rather simplistic view of Monte Carlo parallelization is only valid for cases in which the entire problem geometry is sufficiently small that it will fit within the memory that is available on a single processor. If this is not the case, as for example in the case of a three dimensional mesh problem geometry, then the problem geometry must be decomposed into subdomains: *spatial parallelism.* Spatial decomposition results in the need to communicate particles between processors when they track to an interprocessor domain boundary. Such communication is *nondeterministic* in nature, and therefore, challenging to implement in an efficient manner.

This paper describes the design, implementation and testing of a new, multidimensional, parallel Monte Carlo transport code named MERCURY [4]. This code supports both forms of parallelism, spatial and

particle, within a single simulation. This method of Monte Carlo parallelization is similar to the approach adopted by Alme, et al. [5]. The MERCURY code is intended to become a general purpose Monte Carlo code with many of the same capabilities as those available in TART [6] and COG [7], which were also developed at the Lawrence Livermore National Laboratory.

The outline of this paper is as follows. The major characteristics and capabilities of the MERCURY Monte Carlo transport code are discussed in the next section. The parallel programming model employed within MERCURY is discussed in detail in Section 3. The parallel performance of MERCURY is analyzed in Section 4. Finally, the conclusions are presented in Section 5.

2. CHARACTERISTICS AND CAPABILITIES OF MERCURY

The MERCURY Monte Carlo transport code has been developed at the Lawrence Livermore National Laboratory (LLNL) under the auspices of the Advanced Simulation and Computing Initiative (ASCI). One of the goals of the ASCI program has been the development of physics codes that can operate at high levels of efficiency on a wide variety of massively-parallel computing platforms. As such, MERCURY has been designed from the outset for parallel computations. While the code was originally intended to be a special purpose transport code, we plan to extend the capabilities of the code in order to make it suitable for use on a wide variety of transport applications.

MERCURY has been designed to model the time-dependent transport of several different types of particles through a medium (the particles move in time, but the medium is static). The code currently is capable of transporting neutrons (n), gammas (γ) and five light ions (^{1}H, ^{2}H, ^{3}H, ^{3}He and ^{4}He). Electron transport capabilities will be added in the future.

The code currently supports mesh-based problem geometries. Currently, MERCURY is capable of tracking particles through two-dimensional (structured r - z cylindrical) and three-dimensional (structured Cartesian and unstructured tetrahedral) meshes. Support for one-dimensional (radial) meshes and combinatorial geometries will be added in the future.

The current tally capabilities include cell-based scalar fluxes using a path-length estimator, cell-based energy deposits, time-dependent leakage and census spectra, and a time-dependent particle balance. Additional tally and detector capabilities will be added over the next several years.

MERCURY supports both k_{eff} eigenvalue and static α time-constant criticality calculations. Population control can be applied to any type of particle. The current capabilities include Russian Roulette and splitting.

While the code will eventually support a wide variety of general external sources, the current capability is somewhat limited. At present, we support only monoenergetic, time-independent external sources. In addition, the code is capable of modeling reaction-based sources, such as from thermonuclear reactions, on a per cell basis.

The Monte Carlo All Particle Method (MCAPM) library [8] provides the code's interface to the nuclear cross sections. This library provides the cross-section sampling and collisional kinematics support within MERCURY. While the MCAPM library uses the LLNL developed Evaluated Nuclear Data Library (ENDL) format to store the nuclear data, both the ENDL and ENDF/B-5 data bases are currently

supported, and the translation of the ENDF/B-6 data base into ENDL format is nearing completion. Since the MCAPM library originally supported only multigroup data, MERCURY was developed using a multigroup energy treatment. Recent advances in the development of MCAPM have provided support for continuous-energy cross sections. In addition, a multiband treatment of multigroup cross sections is planned for MCAPM. Modification to MERCURY are planned in order to take advantage of these additional capabilities within MCAPM.

3. THE MERCURY PARALLEL PROGRAMMING MODEL

The requirement that MERCURY perform efficiently on a variety of parallel computing platforms has motivated a three-pronged parallel programming model. MERCURY provides a flexible parallel computing environment by allowing the user to employ each of these forms of parallelism individually or in any combination.

3.1. Parallelization Methods

The first form of parallelism supported by MERCURY is *Domain Decomposition*, which involves the spatial partitioning of the problem geometry (mesh) and the assignment of subdomains to processors. This method allows the code to transport particles through very large meshes. Domain decomposition is a form of *spatial parallelism*, which has been implemented via the use of message-passing techniques on distributed memory computers. Any particle that tracks to a zonal facet which lies on the boundary of a subdomain must be communicated to the processor containing the adjacent subdomain in order to continue the tracking of that particle. The implication is that the non-deterministic nature of Monte Carlo transport *computations* in the context of a spatially partitioned mesh results in non-deterministic *communication* patterns among adjacent spatial subdomains. This method is illustrated in Fig. 1, where the mesh has been decomposed into four spatial subdomains each containing sixteen zones. The red arrows indicate the paths for particles that are communicated between adjacent spatial subdomains.

The second form of parallelism supported by MERCURY is *Domain Replication*, which involves the distribution of the particle load across redundant copies of a spatial domain. This method allows the code to transport large numbers of particles. Domain replication is a form of *particle parallelism*, which has also been implemented via the use of message-passing techniques on distributed memory computers. Since particles are tracked independently on each copy of the domain, this method requires no communication during the particle tracking phase of the calculation. For this reason, this method has often been termed “embarrassingly parallel”. However, the results of each independent calculation must be summed across (communicated among) all replications to obtain the final result. Examples of the tally quantities that are summed in MERCURY include scalar fluxes, energy deposits and isotope mass depletions/accretions. Fig. 2 illustrates this method, where the entire domain has been replicated to produce two independent calculations. The blue arrows indicate the paths for summing “reduction” communications between the redundant domains.

The domain decomposition *and* domain replication methods may be used in combination to achieve flexible parallelism schemes, as shown in Fig. 3. While this figure indicates an uniform level of domain replication for each workgroup, such a choice is not mandated. The technique that we have implemented to achieve load balanced calculations involves varying the level of domain replication by workgroup. This technique may be applied in either a static sense (as shown below in Table I), or may evolve dynamically in

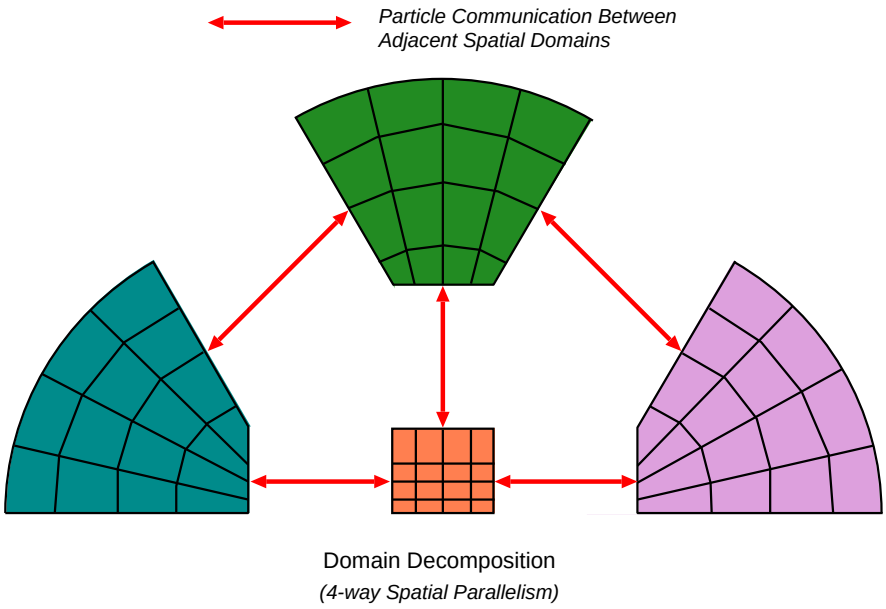

Figure 1. Schematic representation of domain decomposition to achieve four-way spatial parallelism. The red arrows indicate the paths for particles that are communicated between adjacent spatial sub-domains.

response to uneven particle migration to a subset of the spatial domains (not presented here). Our static approach to load balancing is similar to that employed by Alme, et al. [5]. In an attempt to further balance particle loads, we communicate particle buffers to each processor of an adjacent workgroup in round robin fashion, as indicated by the multiple red communication paths shown in Fig. 3.

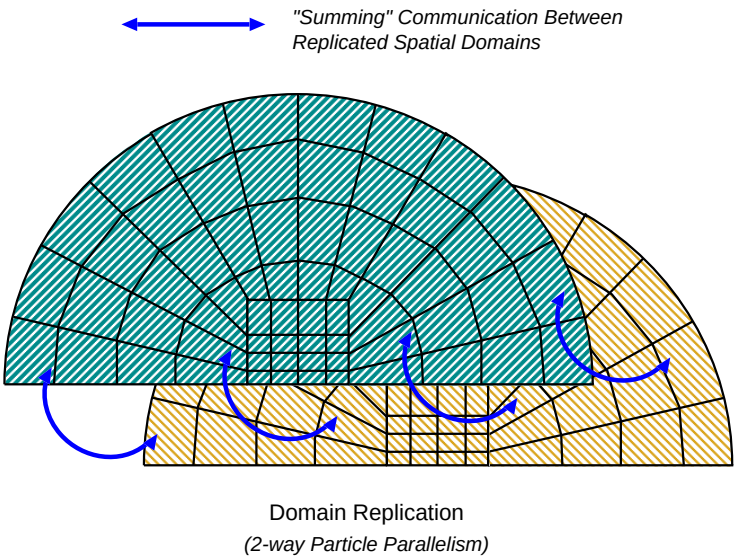

Figure 2. Schematic representation of domain replication to achieve two-way particle parallelism. The blue arrows indicate the paths for summing "reduction" communications between the redundant domains.

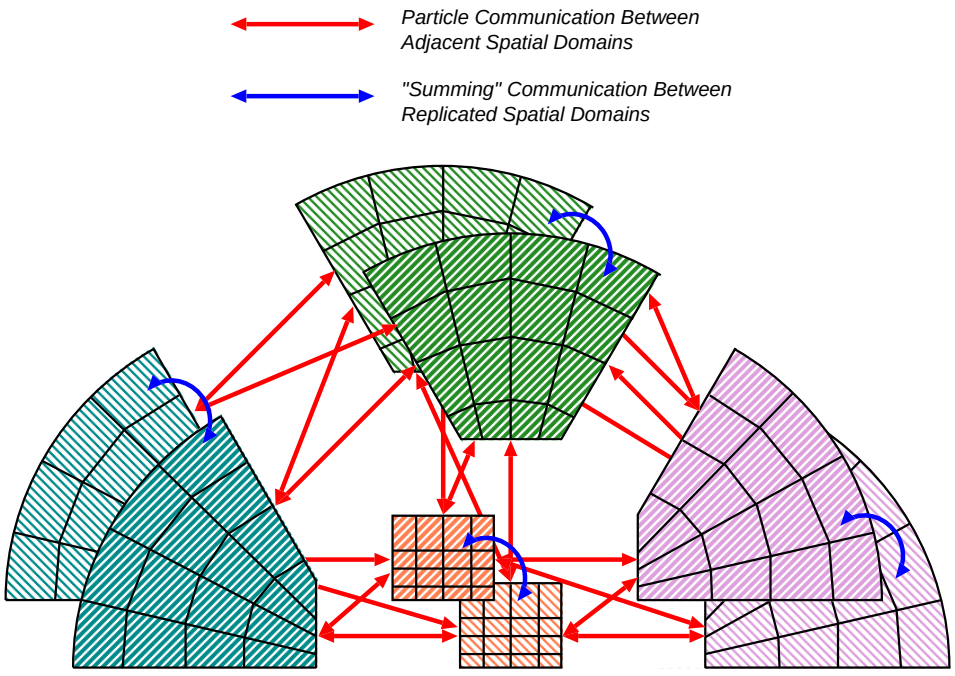

Figure 3. Schematic representation of domain decomposition and replication to achieve eight-way parallelism: four-way spatial and two-way particle parallelism.

The third form of parallelism supported by MERCURY is *Task Decomposition*, wherein the main particle loop is decomposed by assigning tasks (particles) to threads on shared memory computers. This represents another form of *particle parallelism.* This has been implemented in the code through the use of OpenMP [9] pragmas. This method may be used in isolation, or in conjunction with the other parallelization techniques.

One of the problems that is typically encountered in shared memory programs is the need to atomically lock a shared data structure when each processor is writing to it, in order to avoid data corruption or an indeterminate state of the data. The process of locking data structures incurs a large performance penalty, both due to the time required to initiate and release the locks, as well as the serialization that results when many processors are attempting to write to the data structure at about the same time. In an effort to avoid the use of locks, we have adopted a task redundancy algorithm. This method uses one copy of the "shared" data structure for each task. Once the specific portion of the parallel calculation has completed, we simply sum the partial results for each task. Note that this summing operation is local to the shared-memory node: no additional communications are required.

3.2. Processor Roles

The use of a message passing paradigm for the implementation of the first two types of parallelism has produced a three-level processor hierarchy in which individual processors are assigned varying levels of responsibility. The three processor roles are that of *Worker, Foreman* and *Boss.*

A Worker processor is charged with transporting particles on one of the replicated spatial subdomains. Worker processors are only used when particle parallelism is employed via domain replication.

A Foreman processor performs all of the duties that are assigned to a Worker, and also controls the operations of the Worker processors in its *work group*. A Foreman processor is responsible for the transport of particles through all of the subdomains that are assigned to the work group. Foreman processors are used when spatial parallelism is employed via domain decomposition, with or without domain-replication-based particle parallelism. A Foreman is also responsible for performing duties beyond those that are assigned to Workers. These additional duties include the writing and reading (in parallel) of restart and graphics dumps, as well as acting as a conduit for data that is sent between the Worker and Boss processors.

The sole Boss processor performs all the duties that are assigned to a Foreman (and thus a Worker), and is also responsible for the overall operation of the code. The Boss processor is used for spatial parallelism via domain decomposition, and/or particle parallelism via domain replication. The additional duties that the Boss undertakes are the reading of the input file, the writing of edit files, and the calculation of all critical (serial) sections of the code.

3.3. Reproducibility of Parallel Calculations

In an effort to ensure the reproducibility of parallel calculations, we have adopted a hierarchical random number scheme. A single, common *simulation* random number seed (state) is defined and is available on each of the processors. *Domain* and *source* seeds are then spawned from the simulation seed for each domain and/or source in sequential order. These domain and source seeds are used to spawn the *particle* seeds of all particles created in that domain/source. Once a particle has been assigned its individual seed, it uses that seed to obtain all the random numbers necessary to track the particle for its lifetime. Along with the coordinates, velocities, weight, etc., this random number seed is a defining quantity for each particle. The particle carries its seed along as it tracks through cells and between spatial domains. This seed is used to spawn the seeds of secondary daughter particles resulting from collisions.

This hierarchical method ensures the reproducibility of parallel calculations which employ particle parallelism via domain replication and/or task decomposition. This method also minimizes, but does not eliminate, the possibility of irreproducible parallel calculations which employ spatial decomposition. The reproducibility of such calculations can not be guaranteed for more than two spatial domains. This is due to the non-deterministic, asynchronous nature of the communications used to move particles between adjacent spatial domains on a multiple instruction, multiple data (MIMD) parallel computer. Consider a three domain problem in which each domain communicates with the other two. In one run, it is possible that a particle buffer from domain A to domain C will arrive prior to the buffer from domain B, while a different race condition is possible in a second run, where the buffer from domain B arrives prior to that from domain A. Therefore, the particles are not processed in the same order in the two runs, such that the accumulation of any tally quantity may lead the simulation down different paths. Note that it is possible to ensure reproducibility in spatially decomposed calculations if one employs synchronous communications. However, this leads to a significant reduction in the parallel efficiency of the calculations.

4. THE PARALLEL PERFORMANCE OF MERCURY

The parallel performance of MERCURY is analyzed for calculations of two criticality problems: a k_{eff} eigenvalue calculation of the Planet critical assembly and a static α time-constant calculation of the

gedanken double-density Jezebel system. Each of these systems was modeled in two-dimensions (using an axisymmetric, cylindrical r - z geometry mesh) for a series of parallel computations with processor counts in the range $2 \leq N_{proc} \leq 64$. Sequences of calculations were performed which employed only domain decomposition (Sequence A) or domain replication (Sequence B), and a third which employed a combination of both methods (Sequence C). Sequences A and B were each run in two modes: a constant-total work load of $N_{part} = 2 \times 10^5$ particles (Mode 1), in an effort to study speedup, and a variable work load of $N'_{part} = 2 \times 10^4$ particles per processor (Mode 2), in an effort to study self-speedup.

The results from calculation Sequences A and B are shown in Figs. 4 and 5, respectively. In each of these figures, part (a) displays the parallel speedup

$$S \equiv \frac{t_1}{t_{N_{proc}}} \tag{1}$$

and part (b) displays the parallel efficiency

$$\varepsilon \equiv \frac{t_1}{N_{proc} \cdot t_{N_{proc}}} \tag{2}$$

where t_1 and $t_{N_{proc}}$ are the execution times on 1 and N_{proc} processors, respectively. These calculations were performed on the ASCI White computer at the Lawrence Livermore National Laboratory (LLNL), an 8192 processor IBM SP2 system composed of 512 16-way symmetric multiprocessors (SMPs) that are connected by a high-speed network. Due to memory limitations for very-high particle count calculations (not described herein), it was decided that all calculational sequences would extrapolate the one-processor run time t_1 from the two-processor run time t_2, assuming linear speedup over that range.

Note that none of the calculational sequences have employed task decomposition, although this method was available for use on the SMPs of ASCI White. These sequences were designed to study the performance of the message-passing parallelism within the code. It should be pointed out that the implementation of the MPI [10] message passing library on ASCI White employs shared memory within an SMP, and as such, provides very efficient communications on a node. Early studies of the code's performance on ASCI White have shown that domain replication (message passing) is more efficient than task decomposition (shared memory threading) for achieving particle parallelism.

The results presented in Fig. 4 for Sequence A indicate that the parallel efficiency of the domain decomposed calculations begins to degrade dramatically for $N_{proc} > 2$, and begins to approach an asymptotic value for $N_{proc} \simeq 64$: $S = 20.0 - 21.6$ and $\varepsilon = 0.31 - 0.34$. Note that these results are relatively independent of the run mode: adding more work per processor did *not* dramatically improve the efficiency of these calculations. The black lines in the figures indicate linear speedup.

These results, while not encouraging, are not surprising. As the problem domain is subdivided into more and more subdomains, the total number of communications paths between the adjacent spatial subdomains increases with a multiplier on the subdomain count that approaches four (in two dimensions). Stated another way, as the *number* of subdomains increases, the *size* of each subdomain decreases, thereby increasing the likelihood that a particle will completely cross a subdomain during its lifetime and require a communication to move it to the adjacent subdomain on a different processor. Since the communications patterns in a spatially decomposed system cannot be determined *a priori*, one must provide an algorithm that dynamically checks for completion of the communications between adjacent spatial subdomains. As the number of potential communications paths increases with subdomain count, the current "all done"

algorithm can require several queries by the Boss processor of the message sent-received balance for each processor. Small delays are enforced between queries in order to allow outstanding messages to complete before the next query. This algorithm will be analyzed in the near future in an effort to improve the parallel performance of the spatially decomposed method.

The results for Sequence B, presented in Fig. 5, indicate that the parallel efficiency of the domain decomposed calculations scale reasonably well with increasing processor count. The performance of Run Mode 1 at $N_{proc} = 64$ is reasonable ($S = 47.3 - 48.3$ and $\varepsilon = 0.74 - 0.76$), especially since the per processor particle count is only $N'_{part} \simeq 3125$. The efficiency improves dramatically as the particle work load per processor is increased: Mode 2 displays better parallel performance than Mode 1. The reason for the observed behavior is simple: for a fixed processor count, the computations cost increases as the particle work load per processor increases, while the summing communications cost remains fixed. Hence, the fraction of time spent communicating decreases for increasing particle count, leading to the increased speedup and efficiency shown in Figs. 5(a) and 5(b). The performance of Run Mode 2 at $N_{proc} = 64$ is quite good: $S \simeq 59$ and $\varepsilon \simeq 0.92$.

The results for the combined decomposition and replication runs of Sequence C are presented in Table I. These results were obtained from k_{eff} eigenvalue simulations of the Planet critical assembly on $N_{proc} = 64$ with $N_{part} = 2 \times 10^5$ particles. The number of spatial subdomains was increased, and the number of replications of each subdomain (replication level) was decreased, by a factor of two for each subsequent simulation. These results indicate that the best performance is obtained with a small number of spatial domains ($N_{dom} = 2$) and a large replication level ($\mathcal{R} = 32$). As the domain count is increased beyond this small number of domains, the run time begins to increase. These results were not unexpected in light of the performance shown in Figs. 4 and 5. However, it is interesting to note that the performance improves as the domain count is increased above $N_{dom} \geq 16$. This behavior was truly unexpected.

Table I. Parallel Performance of Decomposition and Replication

Number of Domains	Replication Level	Run Time [sec]
2	32	227.94
4	16	375.83
8	8	372.57
16	4	347.28
32	2	339.95

The execution time per particle per cycle (iteration), also known as the *grind time*, is shown in Fig. 6 for k_{eff} eigenvalue simulations of the Planet critical assembly. The grind times are presented as a function of the processor count ($2 \leq N_{proc} \leq 64$) as obtained on three parallel computers at LLNL: ASCI Blue-Sky (a 5856 processor IBM SP2 system composed of 1464 4-way SMPs), ASCI White and the Emperor cluster (a 176 processor Linux Networx system composed of 88 2-way SMPs). The domain decomposition method was utilized for these parallel calculations. This figure indicates that for particle counts of $N_{part} = 1 \times 10^6$, MERCURY is able to perform each iteration of the k_{eff} calculation in $\sim 222, 110$ and 60 seconds for $N_{part} = 8$ processors of Sky, White and Emperor, while for $N_{part} = 64$ processors, the corresponding times are $\sim 34, 17$ and 9 seconds.

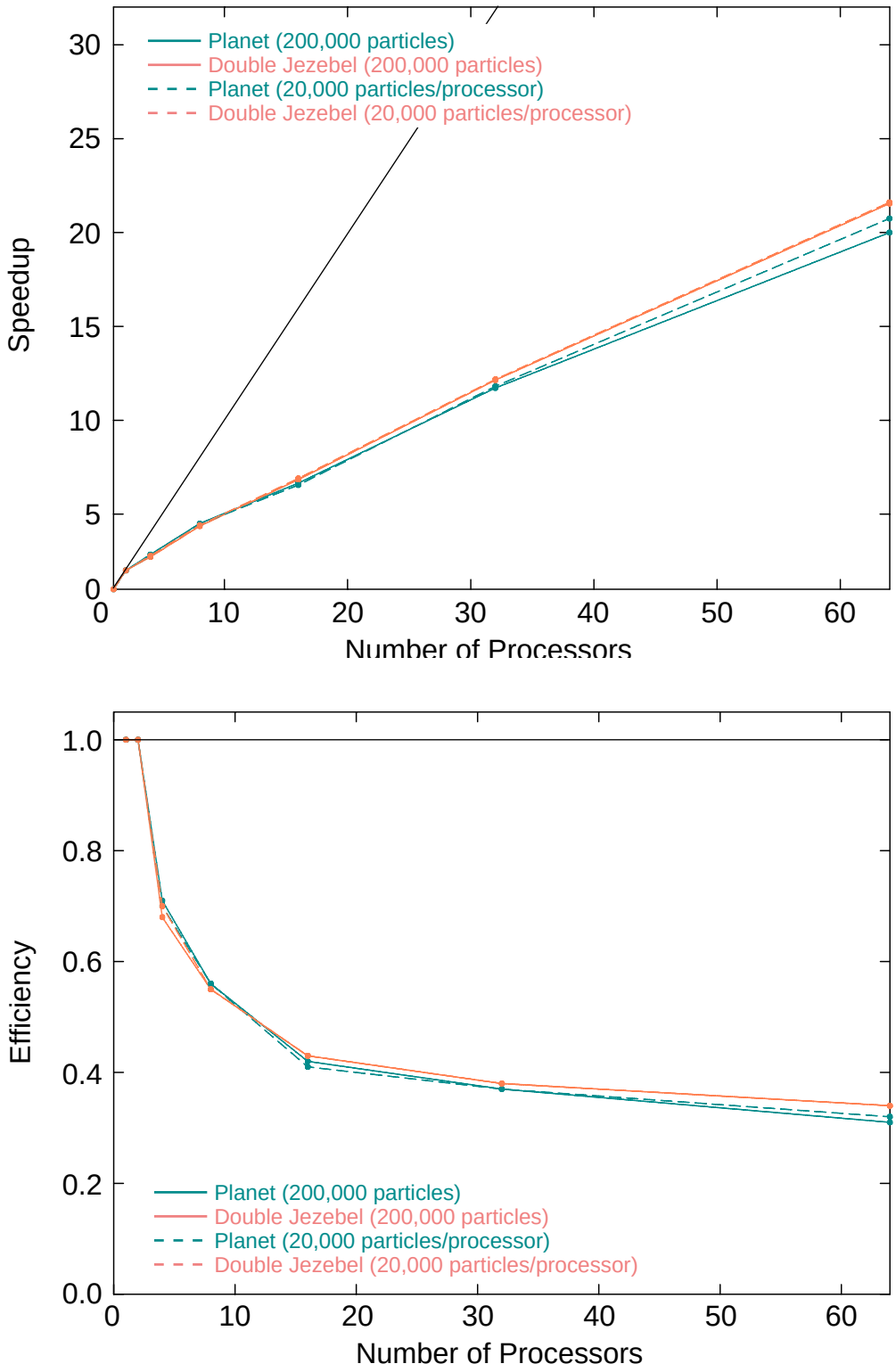

Figure 4. The (a) parallel speedup and (b) parallel efficiency of calculations of the Planet and double-density Jezebel systems which employ *only* spatial domain decomposition (*spatial parallelism*).

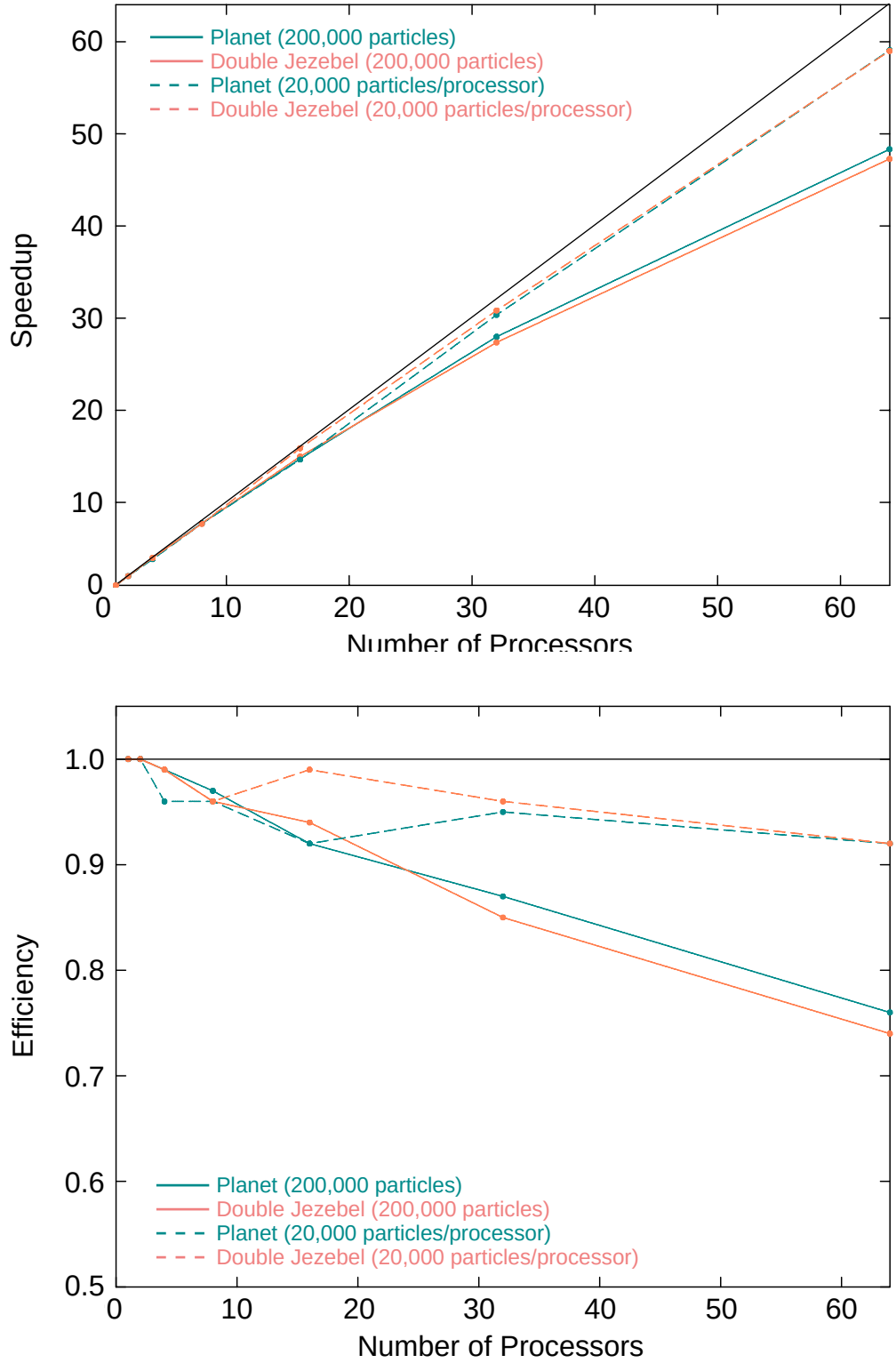

Figure 5. The (a) parallel speedup and (b) parallel efficiency of calculations of the Planet and double-density Jezebel systems which employ *only* domain replication (*particle parallelism*).

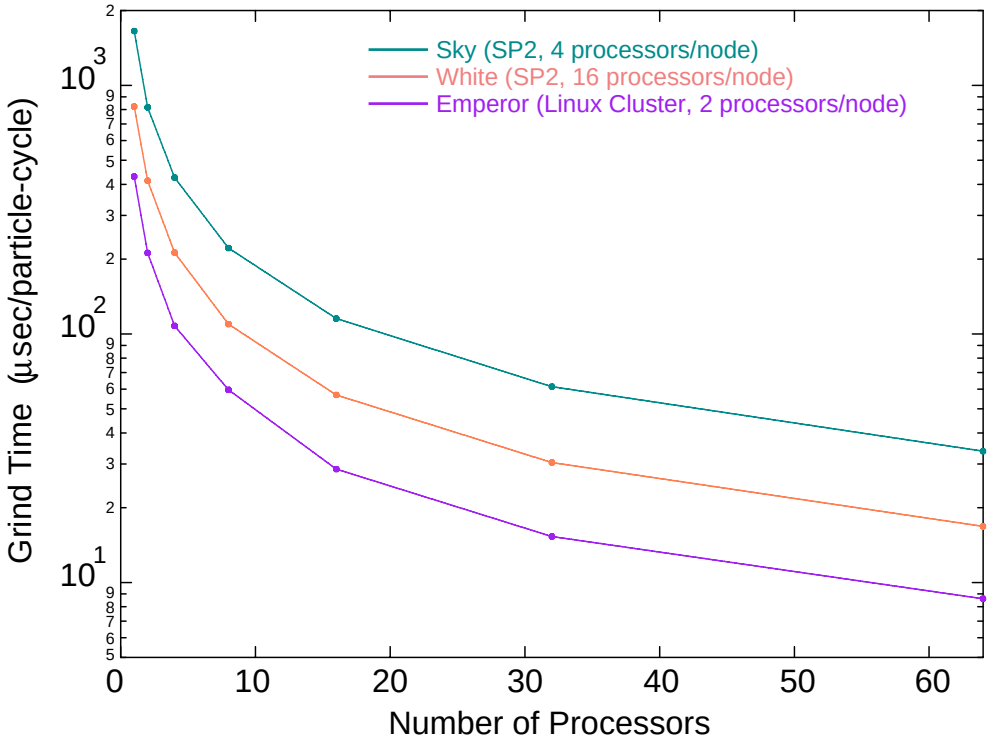

Figure 6. The *grind time* for the k_{eff} eigenvalue simulations of the Planet critical assembly as a function of processor count on three parallel computers.

5. CONCLUSIONS

We have described the parallel programming model for the new Monte Carlo transport code MERCURY, and presented the results from an initial parallel performance study. From its inception, MERCURY has been designed and developed to perform transport calculations on the massively-parallel computing platforms of the ASCI initiative. In order to provide a flexible computing capability on various parallel computers, we have implemented a three pronged approach to parallelism: decomposition of the problem domain (*spatial parallelism* via message passing), replication of individual subdomains (*particle parallelism* via message passing) and task decomposition of the main particle loop (*particle parallelism* via shared-memory threading).

The results from the initial parallel performance (scaling) study of MERCURY indicates that, for the relatively small two-dimensional criticality problems studied, the parallel efficiency of the domain replication method (Sequence B) is superior to that of the domain decomposition method (Sequence A). In addition, we discovered that as the work load (particle count) per processor was increased (Mode 2 relative to Mode 1), the performance of the domain decomposition method did not change, while the performance of the domain replication method increased noticeably. When these parallelization techniques were applied in combination on a fixed number of processors (Sequence C), we found that the performance of the code is best for a small numbers of domains with a large replication level. This result is not surprising based upon the results from Sequences A and C. However, we were surprised to see that the performance improved at large domain counts with small replication levels.

ACKNOWLEDGMENTS

The authors wish to thank the reviewers for providing valuable suggestions which have improved the quality of this paper. This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract Number W-7405-ENG-48.

REFERENCES

[1] C-H. Ma, "Implementation of a Monte Carlo Code on a Parallel Computer System", *Parallel Computing*, **20**, 991 - 1005, 1994.

[2] T-Z. Wan and W.R. Martin, "Parallel Algorithms for Monte Carlo Photon Transport", *Transactions of the American Nuclear Society*, **53**, 285 - 286, 1986.

[3] F. Schmidt, W. Dax and M. Luger, "Experiences with Parallelization of Monte Carlo Problems", *Progress in Nuclear Energy*, **24**, 141 - 151, 1990.

[4] R.J. Procassini and J.M. Taylor, "MERCURY User's Guide: Version a7", Lawrence Livermore National Laboratory, Livermore, CA, USA, 9 December 2002.

[5] H.J Alme, G.H Rodrigue and G.B. Zimmerman, "Domain Decomposition Models for Parallel Monte Carlo Transport", *Journal of Supercomputing*, **18**, 5 - 23, 2001.

[6] D.E. Cullen, "TART-2000: A Coupled Neutron-Photon 3-D, Combinatorial Geometry, Time-Dependent Monte Carlo Transport Code" Report UCRL-ID-126455, Revision 3, Lawrence Livermore National Laboratory, Livermore, CA, USA, 22 November 2000.

[7] R. Buck, D. Clark, S. Hadjimarkos and E. Lent, "COG User's Manual (Second Edition): A Monte Carlo Neutron, Photon and Electron Transport Code", Lawrence Livermore National Laboratory, Livermore, CA, USA, 4 July 1994.

[8] P.S. Brantley, C.A Hagmann and J.A. Rathkopf, "MCAPM-C Generator and Collision Routine Documentation", Report UCRL-MA-141957, Lawrence Livermore National Laboratory, Livermore, CA, USA, 22 November 2000.

[9] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface: Version 1.0", Document Number 004-2229-001, 1998.

[10] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard: Version 1.1", 1995.