The Monte Carlo Method: Versatility Unbounded In A Dynamic Computing World
Chattanooga, Tennessee, April 17–21, 2005, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2005)

DYNAMIC LOAD BALANCING OF PARALLEL MONTE CARLO TRANSPORT CALCULATIONS

Matthew O'Brien, Janine Taylor and Richard Procassini
Lawrence Livermore National Laboratory
Mail Stop L-95, P. O. Box 808
Livermore, CA 94551
mobrien@llnl.gov; taylorj@llnl.gov;spike@llnl.gov

ABSTRACT

The performance of parallel Monte Carlo transport calculations which use both spatial and particle parallelism is increased by dynamically assigning processors to the most worked domains. Since he particle work load varies over the course of the simulation, this algorithm determines each cycle if dynamic load balancing would speed up the calculation. If load balancing is required, a small number of particle communications are initiated in order to achieve load balance. This method has decreased the parallel run time by more than a factor of three for certain criticality calculations.

Key Words: Monte Carlo, particle transport, parallel computation, load balance

1 INTRODUCTION

Monte Carlo particle transport calculations can be very time consuming, especially for problems which require large particle counts or problem geometries with many zones. Calculations of this magnitude are normally run in parallel, since a single processor does not have enough memory to store all of the particles and/or zones. The approach employed in the parallel code MERCURY [1], [2] is to spatially decompose the mesh into domains, and assign individual processors to work on specific domains. This method, known as *domain decomposition*, is a form of spatial parallelism. In contrast, the easiest way to parallelize a Monte Carlo transport code is to store the geometry information redundantly on each of the processors, and assign each processor work on a different set of particles. This method is termed *domain replication*, which is a form of particle parallelism. In many cases, problems are so large that *domain decomposition* alone is not sufficient. For these problems, a combination of both spatial *and* particle parallelism is employed to achieve a scalable parallel solution.

Since particles often migrate in space and time between different regions of a problem, it is a natural consequence of domain decomposition that not all spatial domains will require the same amount of computational work. Hence, the calculation is load imbalanced. In many applications, one portion of the calculation (cycle, iteration, etc.) must be completed by all processors before the next phase can commence. If one processor has more work than any of the other processors, the less-loaded processors must wait for the most worked processor to complete its work.

In an attempt to reduce this form of particle-induced load imbalance, we have developed a strategy which allows the number of processors assigned on a domain to vary in accordance with the amount of work on that domain. The particles that are located in a given spatial domain are then divided evenly among the number of processors assigned to work on that domain, which is termed the domain's *replication level*.

This paper describes a dynamic load balancing algorithm which minimizes the computational work of the most loaded processor by off loading some of the work to other processors. The paper is organized as follows. The parallel architecture of the MERCURY Monte Carlo particle transport code is described in Section 2. This is followed by an illustration of the need for some form of load balancing in spatially-decomposed parallel calculations in Section 3. A discussion of the optimal number of processors that should be assigned to domains is found in Section 4. The various algorithms used to implement dynamic load balancing are described in Section 5, while the conclusions are given in Section 6.

2 THE ARCHITECTURE OF THE PARALLEL MONTE CARLO CODE

The code MERCURY [2] supports two modes of parallelism: *spatial parallelism* via domain decomposition, and *particle parallelism* via domain replication [1]. These methods may be used individually or in combination. Spatial parallelism involves spatial decomposition of the problem geometry into domains and the assignment of each processor to work on a different (set of) domain(s). This method is shown schematically in Figure 1, which represents a 4-way spatial decomposition of a 2-D block unstructured mesh. The red arrows indicate communication events, which are required when particles track to a facet which lies on an interprocessor boundary.

In particle parallelism, the problem geometry is replicated on each processor, and the particles are divided among each of the processors. Figure 2 shows the same 2-D mesh with 2-way domain replicated . The blue, curved arrows represent the collective “summing” communication events that are required to obtain final results from the per-processor partial results.

These methods can also be used in combination, where the problem is spatially decomposed into domains, and then within a domain, the particle load is divided among multiple processors. Each domain can be assigned a different number of processors to work on (replication level), depending on the particle work load. In Figure 3, the central domain is assigned 3 processors, since it has the highest work load. The left and right domains each have a replication level of 2, since they are the next highest loaded domains, while the top domain is not replicated, since it has the lightest particle work load.

3 THE NECESSITY FOR DYNAMIC LOAD BALANCING

The requirement for some form of active management of the particle work load in a spatially-decomposed parallel transport calculation is illustrated in Figure 4. Figure 4a shows the geometry of the double-density Godiva supercritical system, a highly-enriched uranium sphere of radius $r = 8.7407$ cm and density of $\rho = 37.48$ g/cm^3. Particles are source in at the origin and settle calculation is performed to find α eigenvalue of the system. This calculation is run on a 2-D mesh with 4-way spatial parallelism: a 2 by 2 spatial decomposition, as indicated by the black domain boundary lines in Figure 4b and 4c.

Figures 4b and 4c compare two different ways of distributing 16 processors to 4 spatial domains. The first approach (Figure 4b) is to uniformly assign 4 processors to each domain. This configuration does not take into account the actual work load of the domain, so it is less efficient (60% parallel efficiency) than an approach that considers the domain work load when deciding

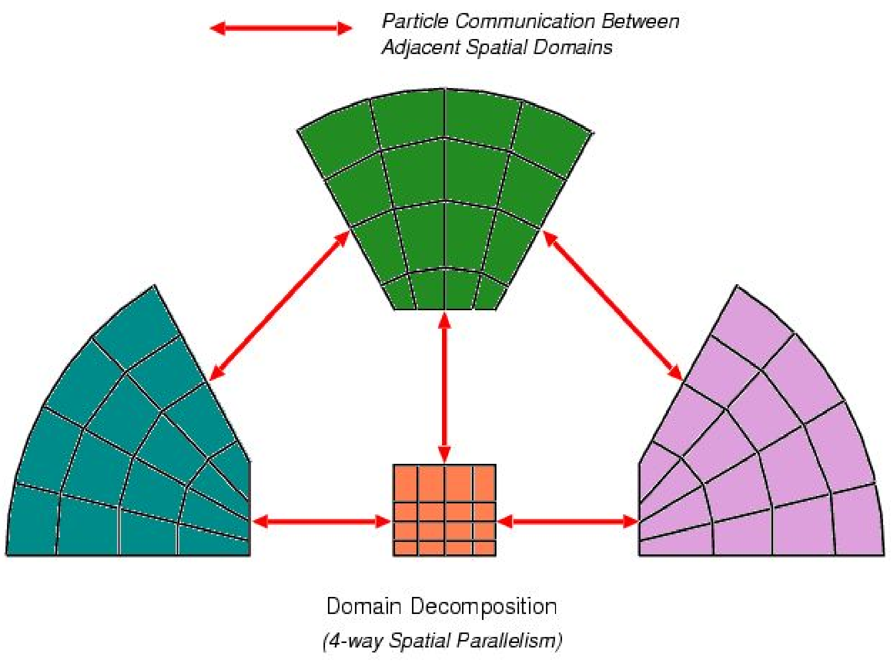

Particle Communication Between Adjacent Spatial Domains
Domain Decomposition
(4-way Spatial Parallelism)

Figure 1. Spatial parallelism in MERCURY is achieved via domain decomposition (spatial partitioning) of the problem geometry. Particles must be transferred to adjacent domains when they reach a domain boundary. The communication of particle buffers between adjacent spatial domains is indicated via the red arrows.

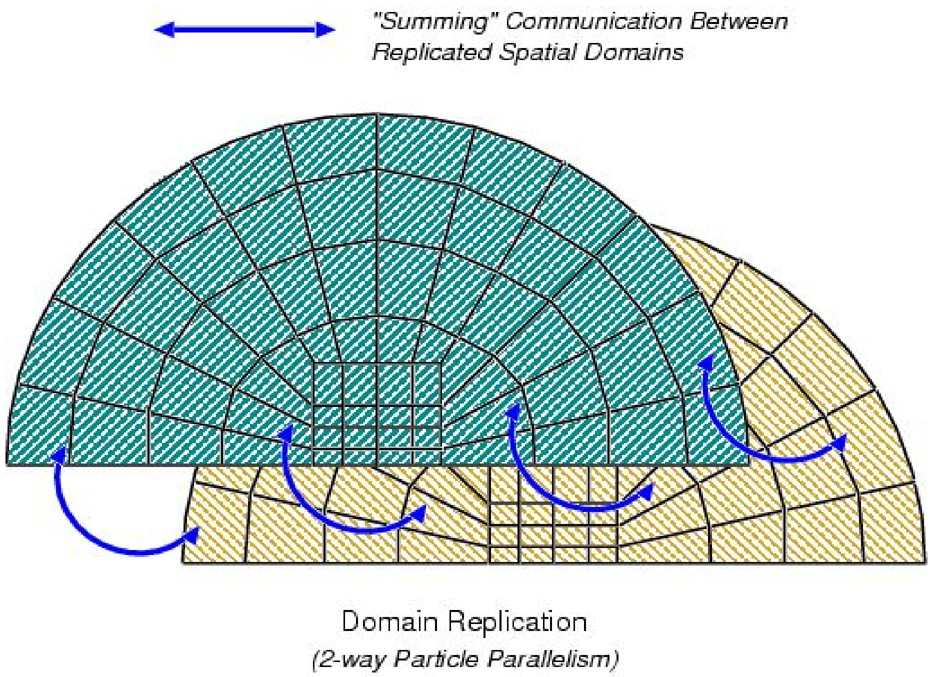

"Summing" Communication Between Replicated Spatial Domains
Domain Replication
(2-way Particle Parallelism)

Figure 2. Particle parallelism in MERCURY is achieved via domain replication (multiple copies) of the problem geometry. The particle workload is distributed across the copies of the domain. The summing communication of partial results is indicated by the blue, curved arrows.

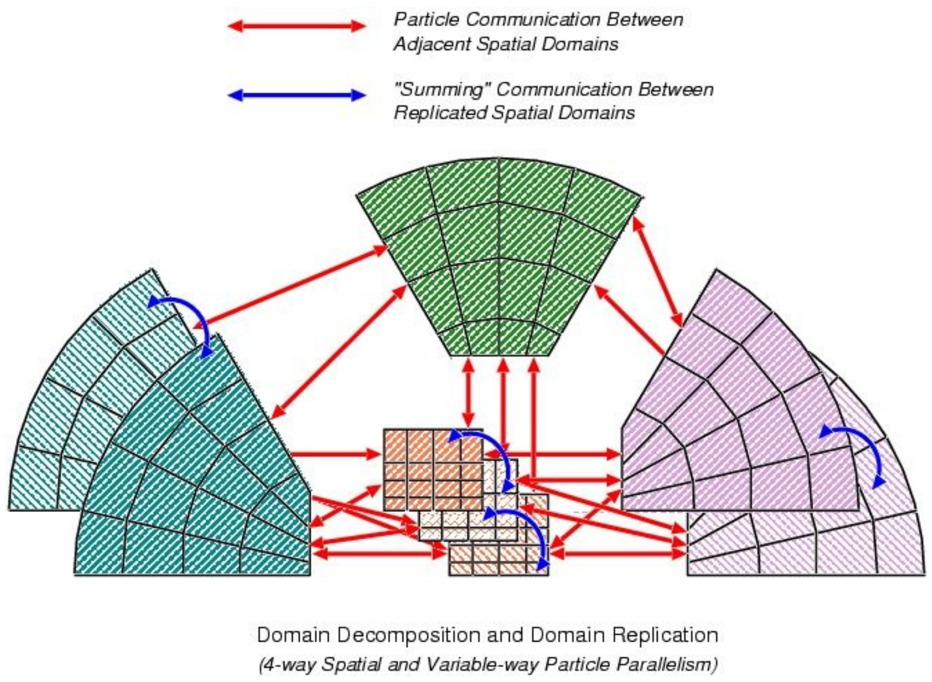

Particle Communication Between Adjacent Spatial Domains
"Summing" Communication Between Replicated Spatial Domains
Domain Decomposition and Domain Replication
(4-way Spatial and Variable-way Particle Parallelism)

Figure 3. Diagram illustrating the combination of spatial parallelism (domain decomposition) and particle parallelism (domain replication). The central domain has 3 processors assigned to it since it has the largest computational work load.

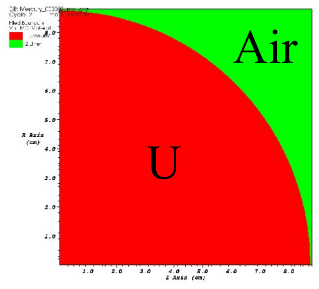
how many processors should be assigned to each domain. The second configuration (Figure 4c) assigns processors to domains based on the work load of the domain. As a result, the parallel efficiency of this calculation is much higher (91%).

As used here, the *parallel efficiency* is defined to be the average computational work over all processors, divided by the maximum computational work on any processor:

$$\varepsilon = \frac{\overline{W}(p)}{\widehat{W}(p)} = \frac{\left(\frac{1}{N_p}\right)\sum_{p=1}^{N_p}[W(p)]}{max_{p=1}^{N_p}[W(p)]} \tag{1}$$

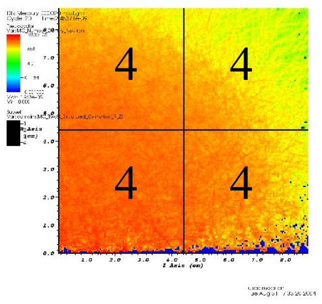
where ε is the parallel efficiency, $W(p)$ is the computational work associated with processor p, N_p is the number of processors, $\overline{W}(p)$ is the computational work averaged over all of the processors and $\widehat{W}(p)$ is the computational work on the most loaded processor.

Note that the parallel efficiency is inversely proportional to the maximum work load of any processor, so having even a single processor that is over worked can really slow down the calculation. Since the calculation wall time is inversely proportional to the parallel efficiency, the goal of load balancing is to maximize the parallel efficiency and minimize the wall time to run the problem. The parallel efficiency of the calculation changes as the problem evolves over time, or as the number of processors assigned to work on a domain changes.



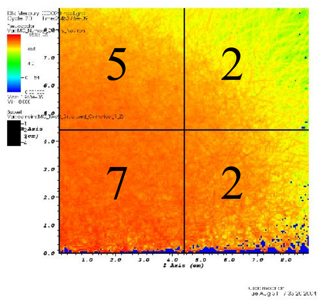
(a)

Problem Geometry



(b)

60% efficient



(c)

91% efficient

Figure 4. The double-density Godiva criticality problem: (a) the problem geometry, (b) a constant, uniform assignment of processors to domains with 4 processors assigned to each domain has a parallel efficiency of only 60% efficient, (c) a dynamic, varying assignment of processors to domains based on the work per domain is 91% efficient. Figures 4b and 4c are pseudo color plots of particle number density, where redder areas represent more computational work.

What is the reason for this large disparity in parallel efficiencies as one changes the replication level of the domains? Figure 5 illustrates the dynamic nature of the particle work load as the problem evolves over time. Time increases to the right, and then down, in the figure. These are pseudo color plots of the particle number density per zone at 6 cycles during the calculation. Initially all of the work is on the lower-left domain (Domain 0), since the particles were sourced in at the origin. As time evolves, the particles migrate to the other domains, first to the upper-left (Domain 2) and lower-right (Domain 1) domains, and then to the upper-right domain (Domain 3). This explains why the *static* replication shown in Figure 4c (7, 2, 5, 2) out performs the processor assignment shown in Figure 4b (4, 4, 4, 4).

4 THE OPTIMAL NUMBER OF PROCESSORS PER DOMAIN

The figures in the previous section clearly show that the computational work load in a parallel Monte Carlo transport calculation changes over the course of the problem. This implies a change in the work load of any given domain. As a result, the number of processors assigned to work on a domain (the replication level) should respond according to the work load of that domain.

Figure 6 is a graph showing the dynamic nature of the work load from cycle to cycle in the double-density Godiva problem. The calculation was run with 4 domains on 16 processors. The calculation begins with a uniform assignment of processors to domains: each domain has 4 processors working on its particles. After the first cycle, the code responds to the large number of (sourced) particles in Domain 0 by assigning 13 processors to it (3 processors are reassigned from each of Domains 1,2 and 3). As time evolves, the work load per domain changes, leading the code to redistribute the number of processors working on each domain. At the end of the simulation, there are 6 processors working on Domains 0 and 2, while 2 processors are working on Domains 1 and 3.

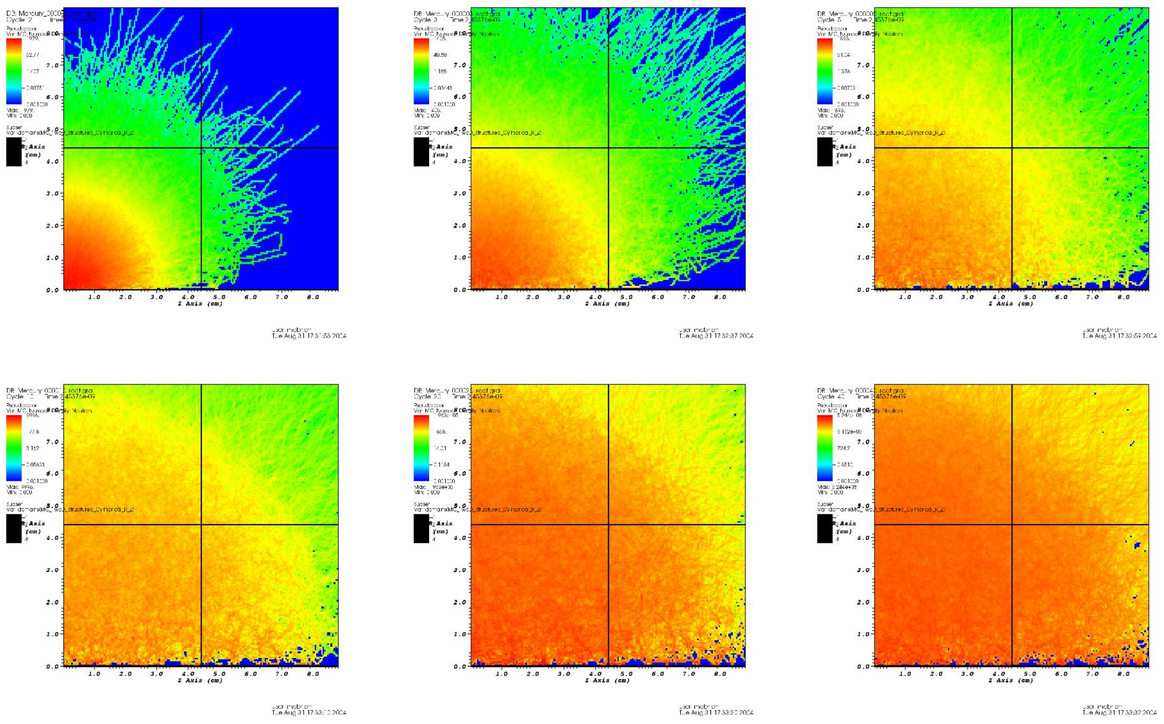


Figure 5. Pseudocolor plot of particle number density at several times during the simulation. Redder areas indicate more computational work. Clearly, there is an uneven workload over time. The black lines indicate the domain boundaries.

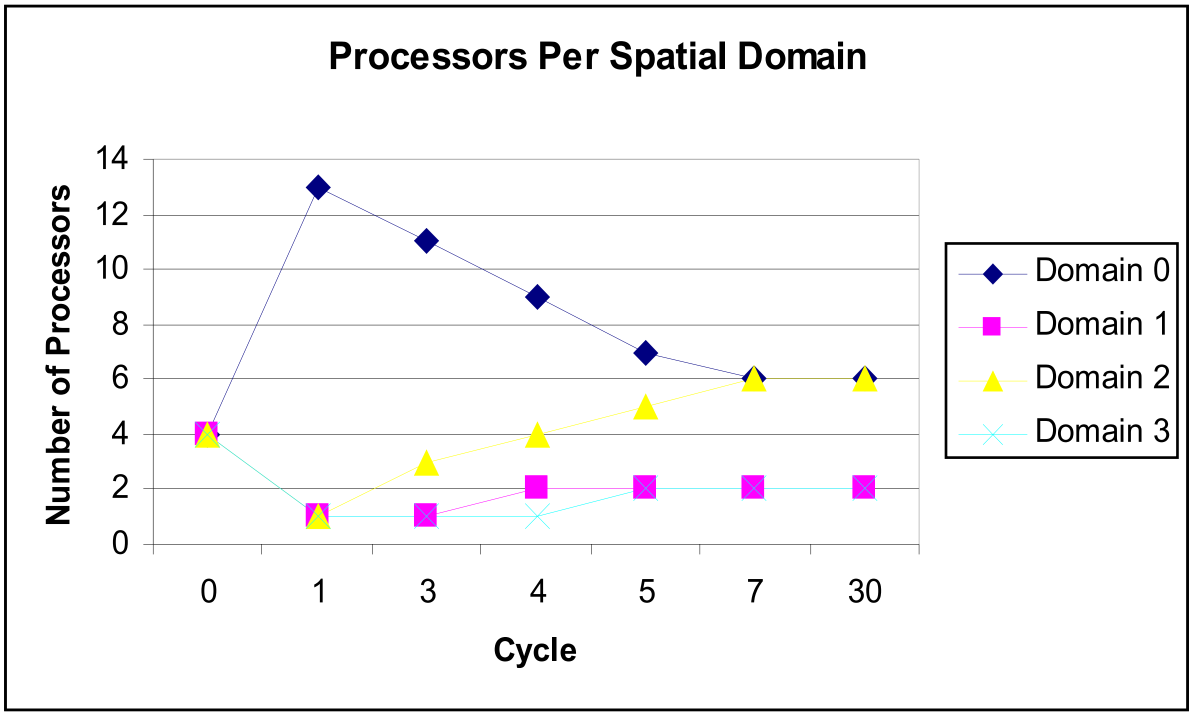


Figure 6. Variation of the replication level (number of assigned processors) of each domain as a function of time.

The computational work performed by each processor $W(p)$ is approximately the number of particle *segments* that occurred on each processor during the previous cycle. A segment is defined to be one of the following particle events:

- Facet Crossing
- Collision
- Thermalization
- Census
- Energy-group Boundary Crossing

The computational work performed on each processor, represented by 1 integer per processor, is then globally communicated, such that each processor knows the work load of all other processors. Since the domain that each processor is currently assigned to is known, it is straightforward to determine the most worked domain. The code then predicts what the parallel efficiency *would* be *if* a redistribution of processors was to take place at the current time. This prediction is used in to determine *when* to perform a dynamic load balance operation.

5 DESCRIPTION OF THE LOAD BALANCING ALGORITHMS

5.1 The Algorithm for Determining When to Load Balance

The algorithm is designed to perform a dynamic load balance operation *only if* it will result in a faster overall calculation. This criterion can be checked inexpensively each cycle. The code calculates the current parallel efficiency (ε_C), as well as what the parallel efficiency *would* be if the code was to redistribute processors right now (ε_{LB}). The ratio of these two efficiencies defines the speedup factor (S):

$$S = \frac{\varepsilon_C}{\varepsilon_{LB}} \tag{2}$$

The wall time required to execute the previous physics cycle (τ_{Phys}) and the time required to perform the load balance operation itself (the communication cost of distributing the particles to other processors, τ_{LB}) define the predicted wall time for the next cycle (τ'):

$$\tau' = \tau_{Phys} \cdot S + \tau_{LB} \tag{3a}$$

$$\tau = \tau_{Phys} \tag{3b}$$

The algorithm used to determine if a dynamic load balancing operation is worthwhile compares the predicted wall time of the next cycle run *with* and *without* a load balance operation:

```
if  $(\tau' <' 0.9 \cdot \tau)$
{
        DynamicLoadBalance();                                   (4)
}
```

5.2 The Domain Replication Level Algorithm

This section describes the algorithm which is used to determine the number of processors that should be assigned on each domain. The goal is to minimize the particle work load on the most worked processor. The data input to, and quantities calculated by, this algorithm are:

Input:

i. N_P is the total number of processors

ii. N_D is the total number of domains, where $N_P \geq N_D$

iii. W_i is the computational work associated with domain i, where $i = 1,\ldots,N_D$

Output:

i. P_i is the number of processors assigned to domain i, subject to the following constraints:

- $P_i \geq 1$ for $i = 1,\ldots,N_D$: Each domain is assigned at least one processor.
- $\sum_{i=1}^{N_D} P_i = N_P$: The sum of all processors assigned to all domains must equal the total number of processors.
- $\max_{i=1}^{N_D}\left(\frac{W_i}{P_i}\right)$ is minimized: The maximum work per processor is minimized.

This algorithm is similar to what a manger of a company might employ when assigning employees to different projects. In this scenario, N_P is equivalent to the total number of employees, N_D is the number of projects, W_i is the total work for project *i*, and P_i is the number of employees working on project *i*. To start, each project is assigned 1 employee. The process then continues in an iterative manner, by finding the project with the most work per employee and assigning it another employee, until there are no more employees available.

A straightforward algorithm can be used to accomplish this task:

```
(1) Initialize $P_i = 1$ for $i = 1,\ldots,N_D$
(2) for ( $i = N_D;\ i < N_P;\ i{+}{+}$ )
    {
        (a) Find $m$ such that $\left(\frac{W_m}{P_m}\right) \geq \left(\frac{W_k}{P_k}\right)$ for all $k = 1,\ldots,N_D$
        (b) Increment $P_m$: $P_m{+}{+}$
    }
```

A simple proof by mathematical deduction on P_i shows that this algorithm minimizes $\max_{i=1}^{N_D}\left(\frac{W_i}{P_i}\right)$.

5.3 Communication Algorithm

Once the per-domain particle work load is used to determine the optimal number of processors to assign to each domain, particles must be communicated between processors in order to move from the current, *load-imbalanced* state, to the desired *load-balanced* state. This is accomplished by finding the *changes* in particle count that must be communicated, followed by sending that small set of changes in order to achieve load balance.

The operation of this algorithm is illustrated in Figure 7. The input required by the algorithm is the current state of (a) the number of processors assigned to work on each domain, (b) the number of particles on each processor, and (c) the number of processors that *should* be working on each domain *after* load balancing. The algorithm then shuffles processors and communicates particles in order to achieve a load balanced state (see Figure 7, Step 4).

The Particle-Communication Load Balancing Algorithm

An easy way to think about this algorithm is to imagine M_d stacks of quarters, where each stack of quarters can be a different height. Say the *i*-th stack of quarters has C_i quarters in it. The goal of the algorithm is find a small set of transfers of quarters from one stack to another such that, in the end, each stack of quarters is about the same height. The general idea is to move quarters from the tallest stack to the shortest stack, such that after the move, one of the stacks will have the *average* number of quarters in it. Once a stack of quarters has the average number of quarters in it, it no longer participates in the transfers, since it already has the target number of quarters.

In the case of the MERCURY load balance algorithm, the number of stacks of quarters corresponds to the number of processors, while the height of each stack of quarters corresponds the number of particles on that processor. Particles must be communicated between processors such that each processor ends up with about the same number of particles on it after the load balancing operation completes.

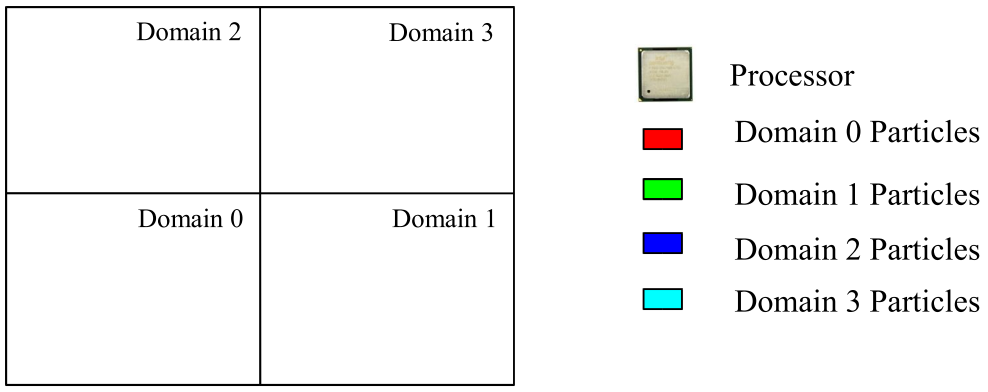
The input data required by, and output data produced by, the particle communication algorithm is as follows:

Input:

i. M_d is the number of processors working on domain *d*
ii. C_i is the number of particles on processor *i*, where $i = 1, \ldots, M_d$

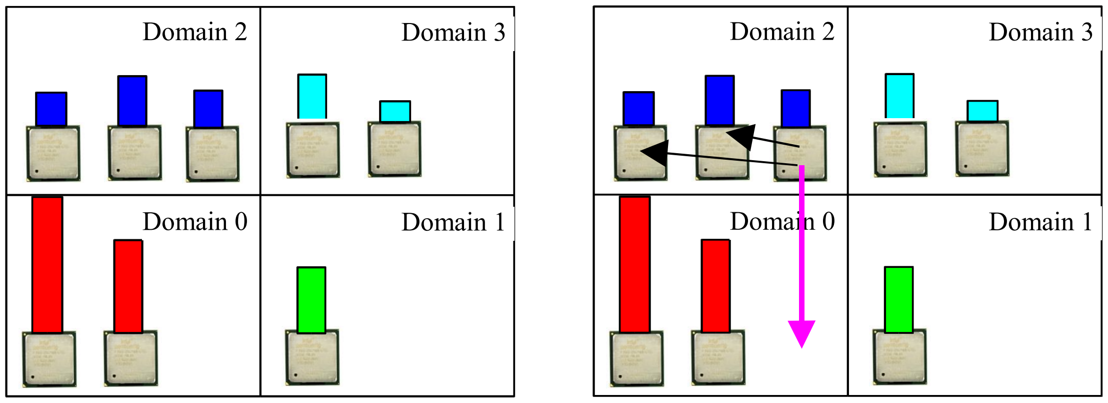
Output:

i. N_E is the number of communications needed to achieve load balance (the number of *edges* in communication graph), where $N_E < M_d$
ii. *COMM$_j$.from* is the number of communications required *from* processor *j*
iii. *COMM$_j$.to* is the number of communications required *to* processor *j*
iv. *COMM$_j$.amount* is the number of particles to be sent *from* processor *j*



This is the legend for the diagrams in this figure. The length of the particle bar indicates the number of particles on each processor. Particles within a domain must remain within that domain after load balancing.

(a)

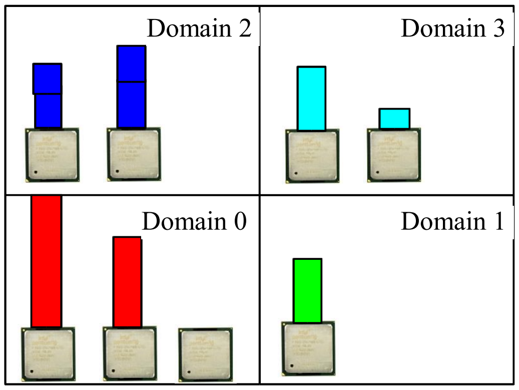


Step 1. The initial particle distribution over processors at the start of a cycle.

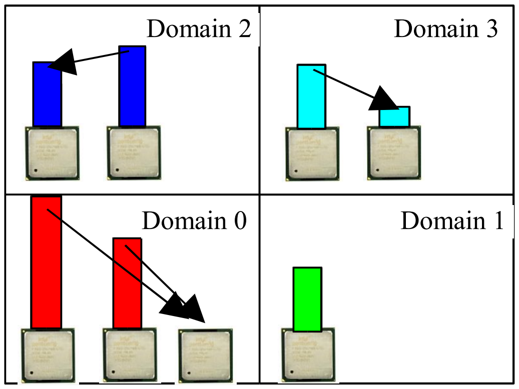
Step 2. Domain 2 loses one processor that is assigned to Domain 0. The particles on the processor that is assigned to Domain 0 must be communicated to the processors that remains assigned to Domain 2.

(b)

Figure 7. A graphical representation of the dynamic load balancing algorithm for Method 3: (a) the legend that describes the various steps of the algorithm, (b) the first and second steps of the algorithm.

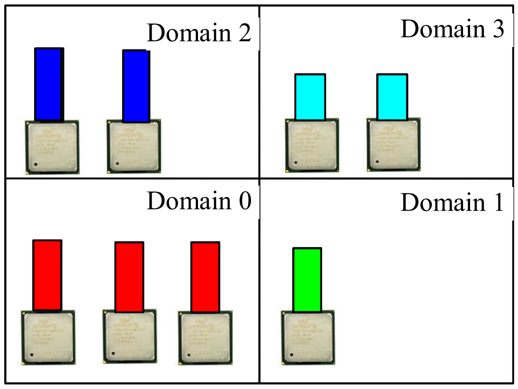


Step 3. After the code determines which processors are assigned to each domain, each domain can balance its particle load.



Step 4. This is the communication necessary to achieve load balance within each domain.

(c)



Step 5. The end result of load balancing: The number of processors per domain has been changed so that the maximum number of particles per processor in minimized.

(d)

Figure 7 (continued). A graphical representation of the dynamic load balancing algorithm for Method 3: (c) the third and fourth steps of the algorithm, (b) the fifth step of the algorithm.

Once the output communication graph has been executed, the number of particles per processor is the same for all processors assigned to work on that domain, modulo a few particles if there is a remainder from the division: $\bar{C} = \left(\frac{1}{M_d}\right)\sum_{i=1}^{M_d} C_i$.

The particle-communication load balancing algorithm is as follows:

```
(1) Initialize $N_E = 0$
(2) while (not load balanced)
    {
        (a) Find i,min such that $C_{i,min} \le C_i$ for all $i = 1,\dots,M_d$
        (b) Find i,max such that $C_{i,max} \ge C_i$ for all $i = 1,\dots,M_d$
        (c) Send particles from i,max to i,min such that either $C_{i,min}$ or $C_{i,max}$ becomes $\bar{C}$
            (Send particles from the processor with the most particles to the processor with
            the least particles, bringing one of them to the average number of particles).
        (d) $COMM_{N_E}.from = i,max$
        (e) $COMM_{N_E}.to = i,min$
        (f) $COMM_{N_E}.amount = min(C_{i,max} - \bar{C},\ \bar{C} - C_{i,min})$
        (g) $C_{i,max}$ -= $COMM_{N_E}.amount$
        (h) $C_{i,min}$ += $COMM_{N_E}.amount$
        (i) Increment the number of edges: $N_E$++
    }
```

This is a very natural load balancing algorithm. Every processor C_i is either *over* the average $\bar{C}$, or *under* the average $\bar{C}$. If a processor's particle count C_i already equals $\bar{C}$, then it does *not* need to participate in load balancing, since it already has the desired number of particles. If C_i is *over* the average, then the processor *sends* particles to other processors, and its particle count is reduced to $\bar{C}$. In contrast, if C_i is *under* the average, then the particle *receives* particles from other processors, increased its particle count to $\bar{C}$. As a result, a processor is either sending or receiving particles, but not both in the same cycle.

The goal is that all processors will end up with the average number of particles per processor. At each iteration of the loop, the particles are sent from the processor with the most particles to the processor with the least particles, and one of those processors will end up with $\bar{C}$ particles. Each iteration of the loop results in one processor having $\bar{C}$ particles, such that the loop can be iterated at most M_d times. The result is a very sparse communication graph that is used to achieve load balance, which is important since communication can be expensive on modern, parallel computing platforms.

6 CONCLUSION

The particle work load in a spatially-decomposed, parallel Monte Carlo transport calculation has been shown to be dynamic and non-uniform across domains. This particle-induced load imbalance results in a reduction of the computational efficiency of such calculations. In an effort to overcome this shortcoming, the MERCURY Monte Carlo code has been extended to include a dynamic particle load balancing algorithm. The method uses a variable number of processors that are assigned to each domain (replication level) in an attempt to balance the number of particles per processor. The algorithm includes logic that determines the optimal number of processors per domain, when to perform a dynamic redistribution of processors to domains, as well as how to perform the load balancing particle communications between processors. This method has been applied to a criticality calculation, where it has yielded more than a three-fold increase in the parallel efficiency.

7 ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy at the Lawrence Livermore National Laboratory under Contract Number W-7405-Eng-48.

8 REFERENCES

1. R. J. Procassini, J. M. Taylor, I. R. Corey and J. D. Rogers, "Design, Implementation and Testing of MERCURY: A Parallel Monte Carlo Transport Code", *2003 Topical Meeting in Mathematics and Computations*, American Nuclear Society (2003).

2. R. J. Procassini and J. M. Taylor, *Mercury User Guide (Version b.6)*, Lawrence Livermore National Laboratory, Report UCRL-TM-204296 (2004).