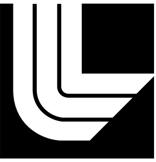


UCRL-PROC-212594

Load Balancing of Parallel Monte Carlo Transport Calculations

R. J. Procassini, M. J. O'Brien, J. M. Taylor

May 27, 2005

Joint Russian-American Five-Laboratory Conference on Computational Mathematics and Physics
Vienna, Austria
June 19, 2005 through June 23, 2005

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Load Balancing Of Parallel Monte Carlo Transport Calculations

R.J. Procassini, M. J. O'Brien and J.M. Taylor

Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551

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

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. Several parallel execution modes techniques are employed in the parallel code MERCURY (Procassini, et al., 2003) (Procassini and Taylor, 2005). The first mode involves spatial decomposition of the geometry into domains, and assignment of individual processors to work on specific domains. This method, known as *domain decomposition*, is a form of spatial parallelism. The second mode, which is 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-loaded processor to complete its work.

In an attempt to reduce this form of particle-induced load imbalance, a technique has been developed which allows the number of processors assigned to a domain, known as the domain's *replication level,* to vary in accordance with the amount of work on that do-

main. This technique requires the use of both spatial and particle parallelism. The particles that are located in a given spatial domain are divided evenly among the processors assigned to work on that domain, known as the domain's *work group*.

This paper describes a dynamic load balancing algorithm which minimizes the computational work of the most loaded processor by off loading part 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 the next section . This is followed by a discussion of a problem that illustrates the need for some form of load balancing in spatially-decomposed parallel calculations. A discussion of the optimal number of processors that should be assigned to the domains is then presented. The various algorithms used to implement dynamic load balancing are described, followed by the conclusions.

The Architecture Of The MERCURY Parallel Monte Carlo Code

The Monte Carlo transport code MERCURY (Procassini and Taylor, 2005) supports two modes of parallelism: *spatial parallelism* via domain decomposition, and *particle parallelism* via domain replication (Procassini, *et al.*, 2003). These modes 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 replication. The blue, curved arrows represent the collective "summing" communication events that are required to obtain final results from the per-processor partial results.

These modes 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 (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.

The Requirement 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 sourced at the origin and a 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

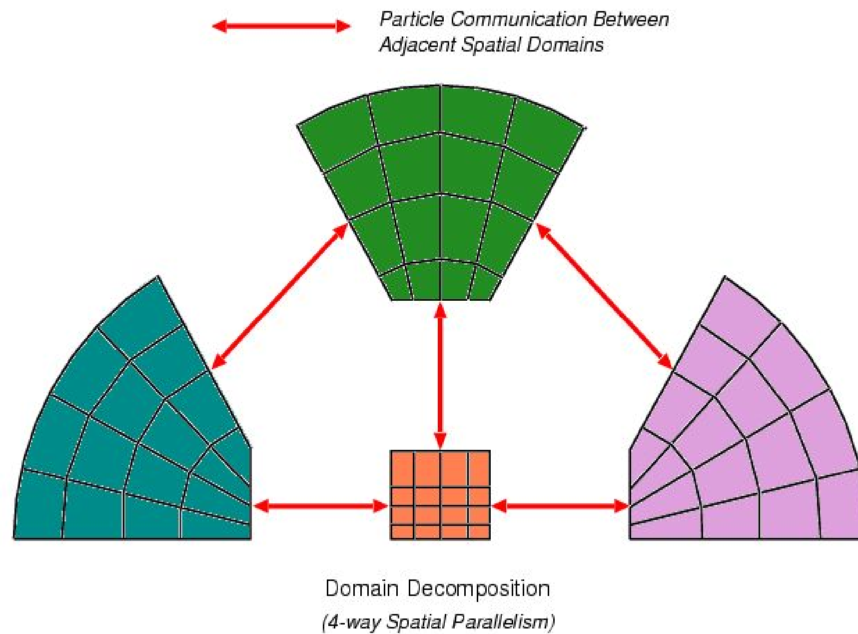


Figure 1. Spatial parallelism in MERCURY is achieved via domain decomposition (spatial partitioning) of the problem geometry. Particles must be transferred to adjacent domains when they reach a domain boundary. The communication of particle buffers between adjacent spatial domains is indicated via the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

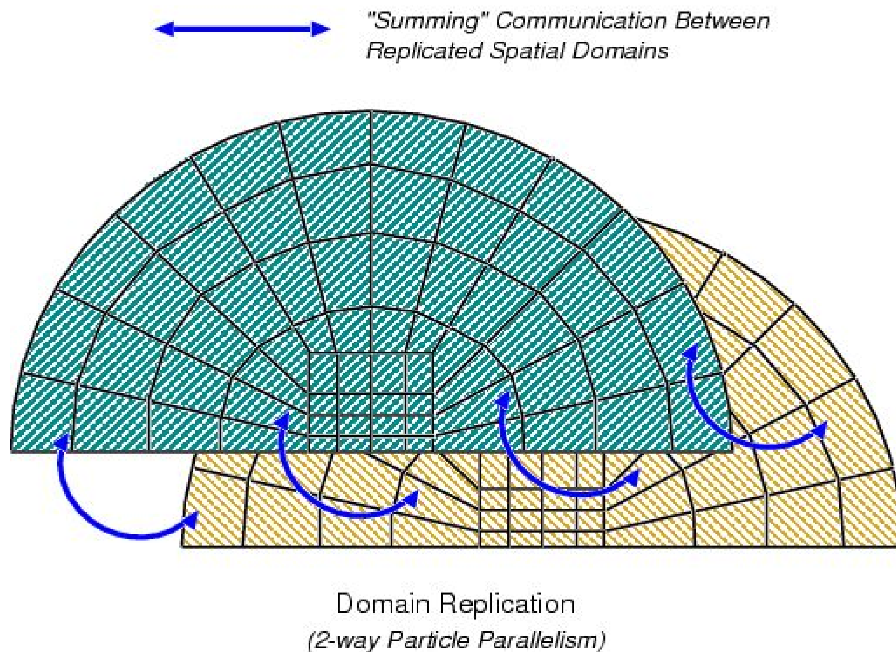


Figure 2. Particle parallelism in MERCURY is achieved via domain replication (multiple copies) of the problem geometry. The particle workload is distributed across the copies of the domain. The summing communication of partial results is indicated by the blue, curved arrows.

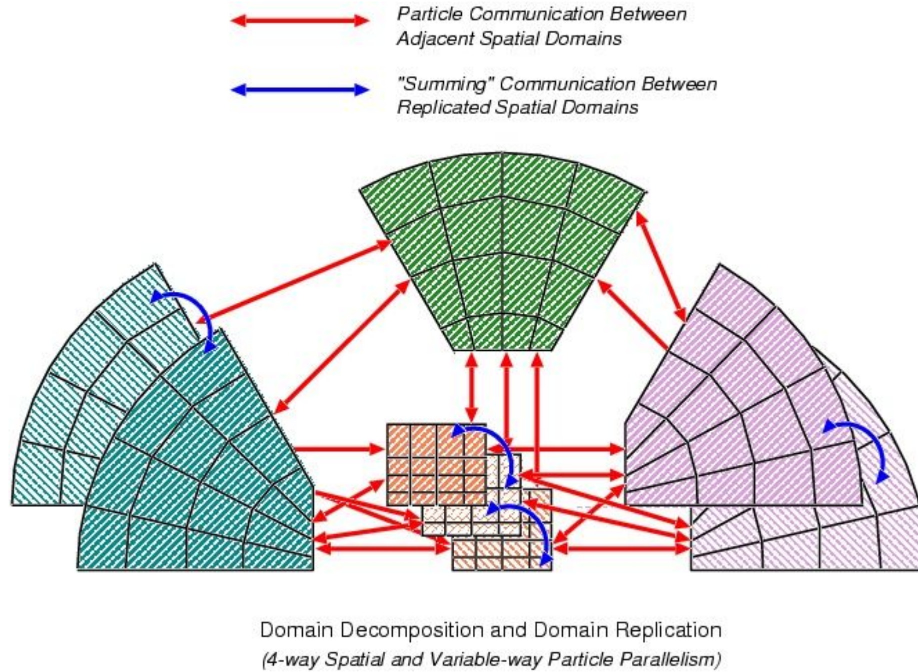


Figure 3. Diagram illustrating the combination of spatial parallelism (domain decomposition) and particle parallelism (domain replication). The central domain has 3 processors assigned to it since it has the largest computational work load.

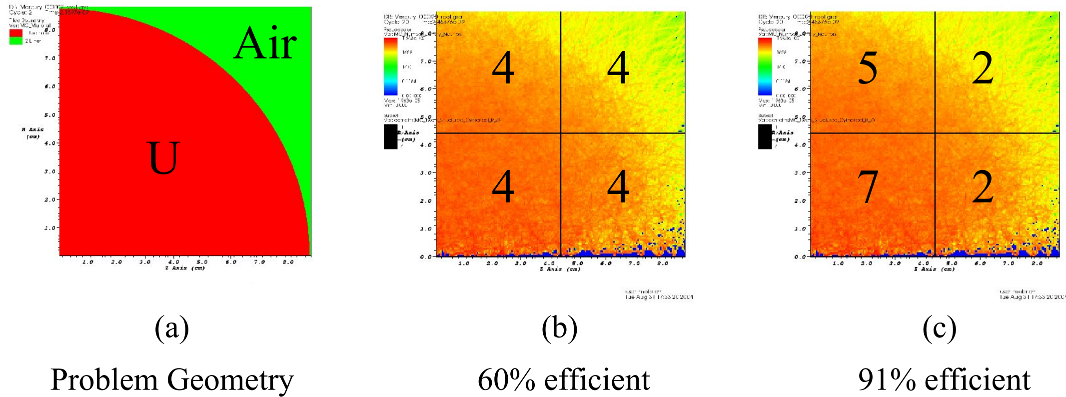
Figures 4b and 4c compare two different ways of distributing 16 processors to 4 spatial domains. The first approach (Figure 4b) is to uniformly assign 4 processors to each domain. This configuration does not take into account the actual work load of the domain, so it is less efficient (60% parallel efficiency,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

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:

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

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 dramatically slow down the entire calculation. Since the calculation run time is inversely proportional



(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.

to the parallel efficiency, the goal of load balancing is to maximize the parallel efficiency and minimize the run time to run the problem. The parallel efficiency of the calculation changes as the problem evolves over time, or as the replication level of the domains changes.

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).

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 (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

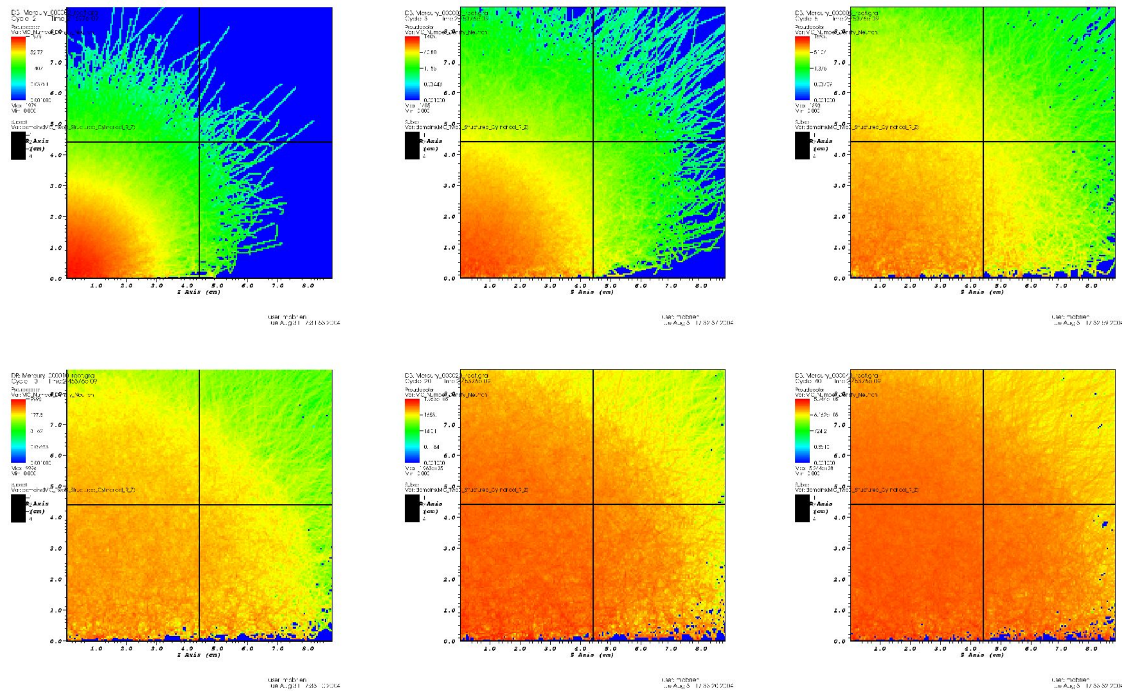


Figure 5. Pseudocolor plot of particle number density at several times during the simulation. Redder areas indicate more computational work. Clearly, there is an uneven workload over time. The black lines indicate the domain boundaries.

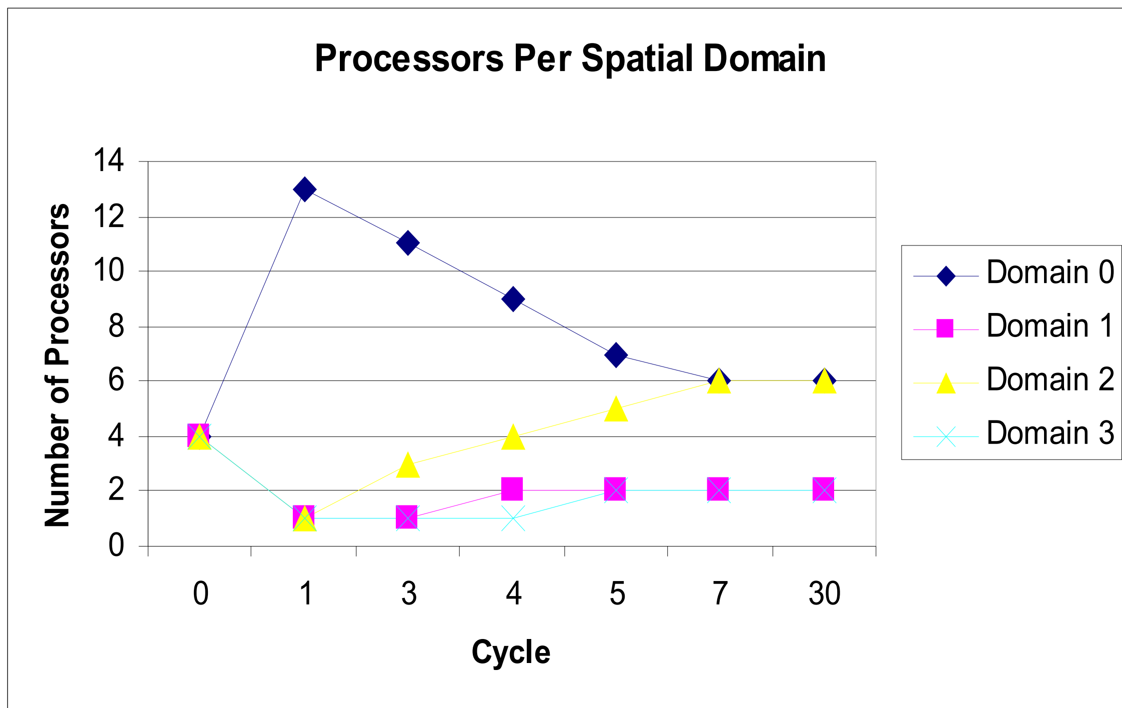


Figure 6. Variation of the replication level (number of assigned processors) of each domain as a function of time.

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.

The computational work performed by each processor $W(p)$ is approximately equal to 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:

(a) Facet Crossing
(b) Collision
(c) Thermalization
(d) Census
(e) Energy-group Boundary Crossing

The computational work performed on each processor, represented by a single 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.

Description of the Load Balancing Algorithms

Determining When to Load Balance

The dynamic load balance algorithm is designed to be executed *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):

$$S = \frac{\varepsilon_C}{\varepsilon_{LB}} \qquad [2]$$

The computer 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 run time for the next cycle (τ'):

$$\tau' = \tau_{Phys} \cdot S + \tau_{LB} \qquad [3a]$$

$$\tau = \tau_{Phys} \qquad [3b]$$

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

$$\text{if } (\tau' <' 0.9 \cdot \tau)$$

```
{
    DynamicLoadBalance();                                        [4]
}
```

Determining The Optimal Domain Replication-Level

This section describes the algorithm used to determine the number of processors that should be assigned to work on each domain, also known as the domain's replication level. This algorithm uses (a) the number of processors (N_P), (b) the number of domains (N_D) and (c) the work load per domain (W_i) in order to determine the optimal domain replication level (P_i) that minimizes the particle work load on the most worked processor. The result is a parallel calculation which is (reasonably) load balanced.

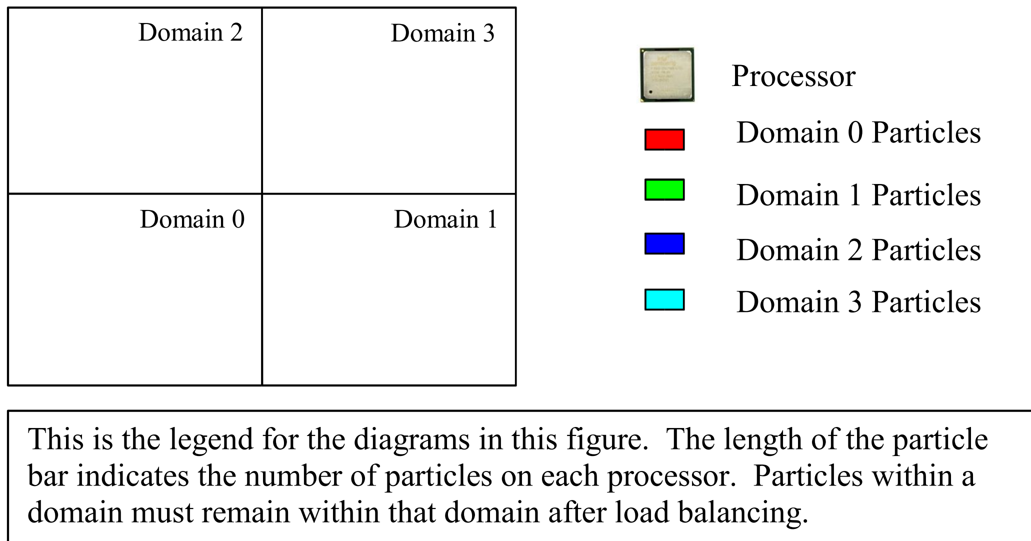
This algorithm is similar to what a manger of a company might use when assigning employees to different projects. In this scenario, N_P is equivalent to the total number of employees, N_D is the number of projects, W_i is the total work for project *i*, and P_i is the number of employees working on project *i*. To start, each project is assigned an equal number of employees. The process then continues in an iterative manner, by finding the project with the most work per employee and assigning another employee to the project, until there are no more employees available. A simple proof by mathematical deduction on P_i shows that this algorithm minimizes the particle work load on the most worked processor.

The Particle Communication Algorithm

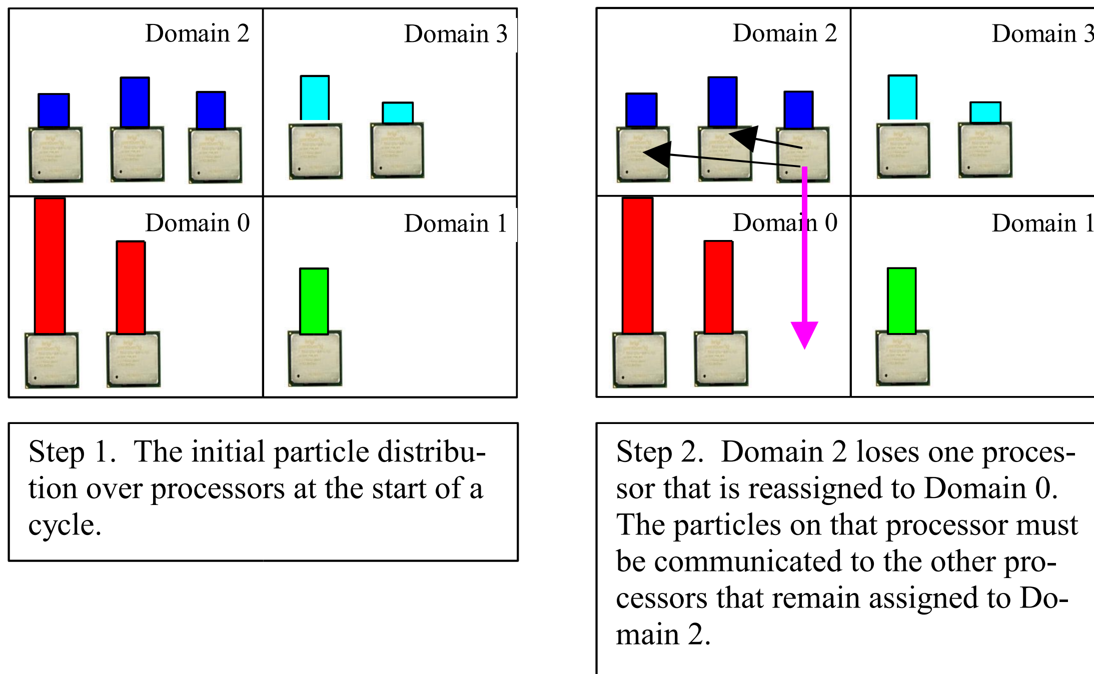
Once the per-domain particle work load has been used to determine the optimal number of processors to assign to each domain, particles must be communicated between processors in order to move from the current *(load imbalanced)* state, to the desired *(load balanced)* state. This is accomplished by (a)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

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

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

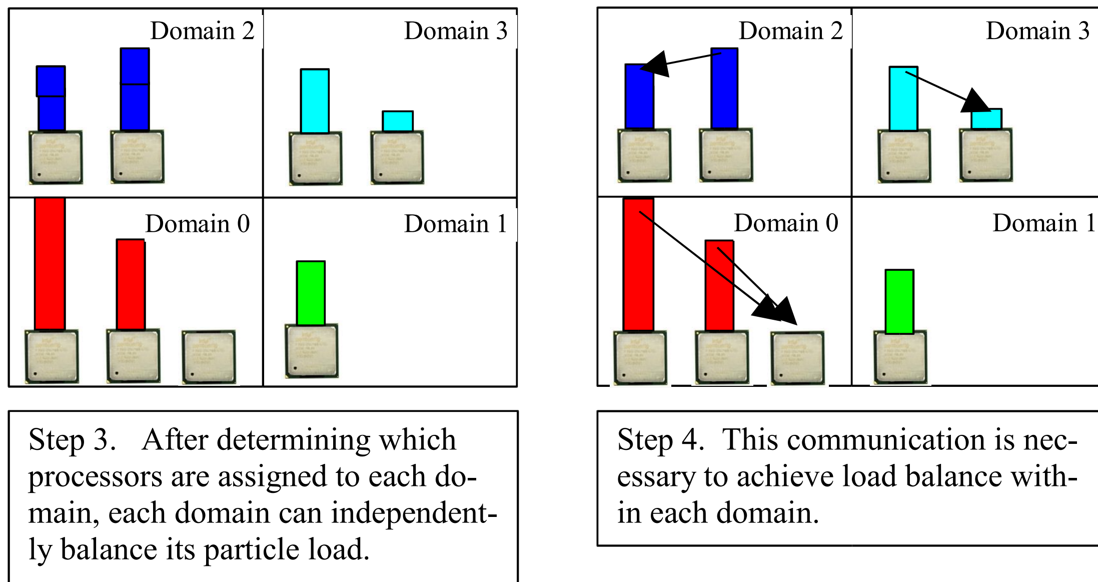


(*a*)

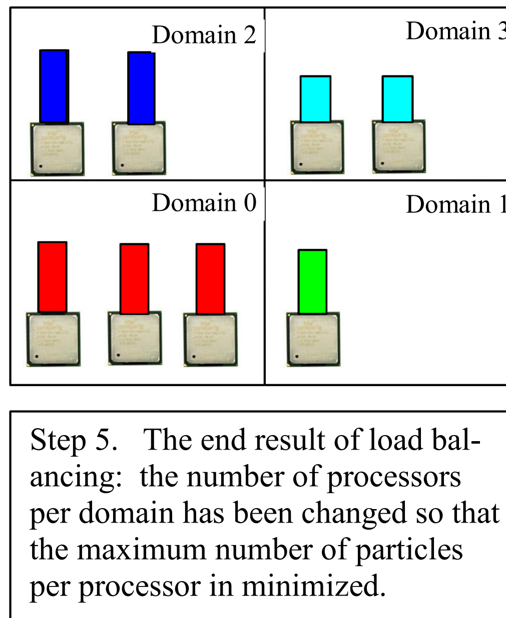


(*b*)

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



(c)



(d)

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

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 $(\bar{C})$ 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. The procedure repeats until all of the stacks consist of the average 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. Once the 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: $\bar{C} = \left(\frac{1}{M_d}\right)\sum_{i=1}^{M_d} C_i$.

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, 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 results in one processor having $\bar{C}$ particles, such that the algorithm 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.

Results

The efficacy of dynamic load balancing in the context of parallel Monte Carlo particle transport calculations is tested by running one criticality problem and one sourced problem. These problems are chosen because they exhibit substantial of particle-induced dynamic load imbalance during the course of the calculation. Each of these problems is time dependent, and the particle distributions also evolve in space, energy and direction over many cycles. Two calculations were made for each of these problems, with the dynamic load balancing feature either disabled or enabled.

Criticality Test Problem

The criticality problem chosen for this test is one of the benchmark critical assemblies compiled in the *International Handbook of Evaluated Criticality Safety Benchmark*

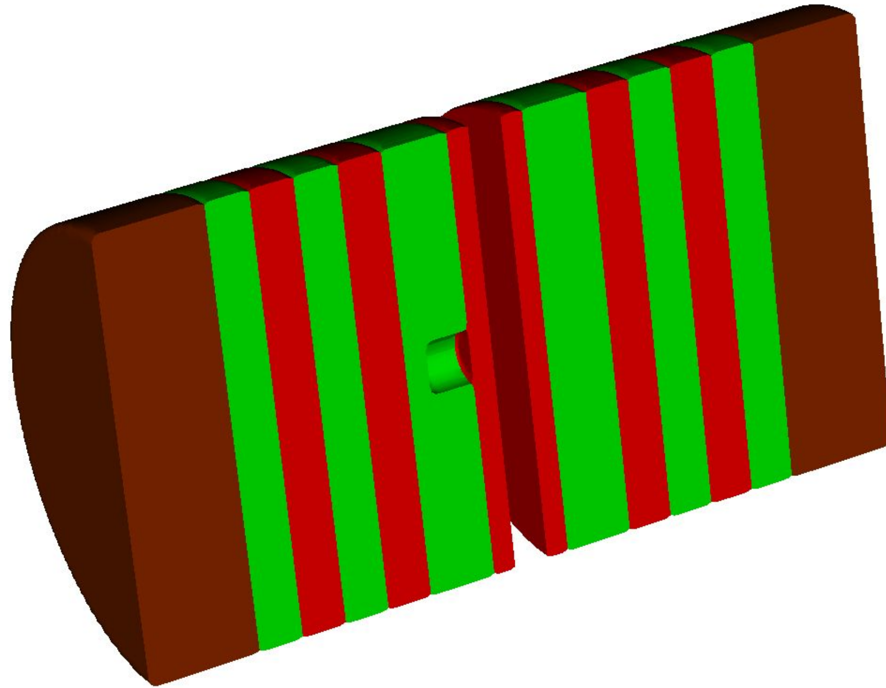


Figure 8. Geometry of the critical assembly test problem HEU-MET-FAST-017. The green regions are highly enriched uranium, while the red and maroon regions are two different forms beryllium.

Experiments (ICSBEP, 2004). This particular critical assembly is a known as HEU-MET-FAST-017: a right-circular cylindrical system comprised of alternating layers of highly-enriched uranium and beryllium, with beryllium end reflectors. The assembly is $L = 35.31$ cm in length and has a radius of $r = 9.995$ cm, as shown in Figure 8. The central cavity contains a neutron source, and the two halves of the assembly are separated by a $\delta = 1.52$ cm air gap.

This problem was run on a 2-D $r - z$ mesh that was spatially decomposed in 8 domains, axially along the axis of rotation. Parallel calculations were made using 16 processors of the ASC White machine (an IBM SP-2 parallel computer with 16-way symmetric multiprocessor nodes) at the Lawrence Livermore National Laboratory (LLNL). These calculations were run with $N_p = 1\times10^6$ particles, using a “pseudo-dynamic” algorithm that iterates in time to calculate both the k_{eff} and α eigenvalues of the system.

The nature of particle induced load imbalance in this calculation is clearly seen in Figure 9. Pseudocolor plots of the particle number density are shown at six cycles during the evolution of the time iteration algorithm. Redder areas indicate a greater particle density, and hence a larger work load, than do blue areas. The domain boundaries are indicated by the black lines in Figure 9. The particles are initially sourced into the problem at the center of the source cavity, as shown in the Cycle 1 plot. As the cycles progress, the particles transport through all of the domains, but it is clear the the heterogeneous nature of this assembly results in uneven particle densities at all cycles.

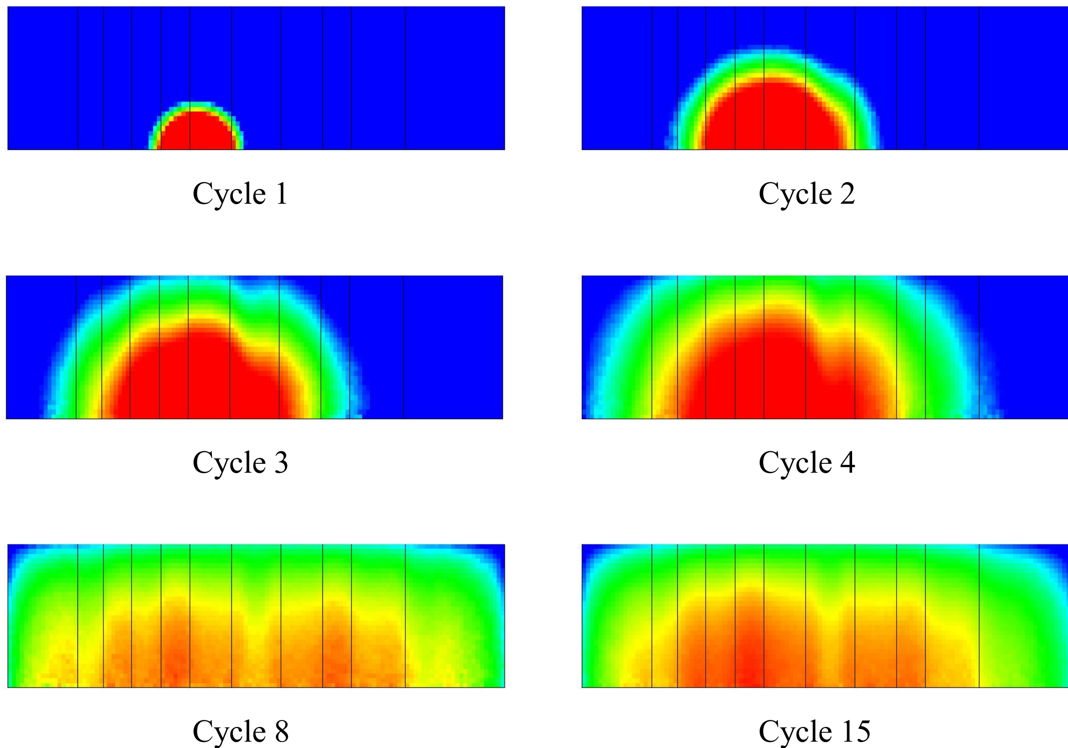
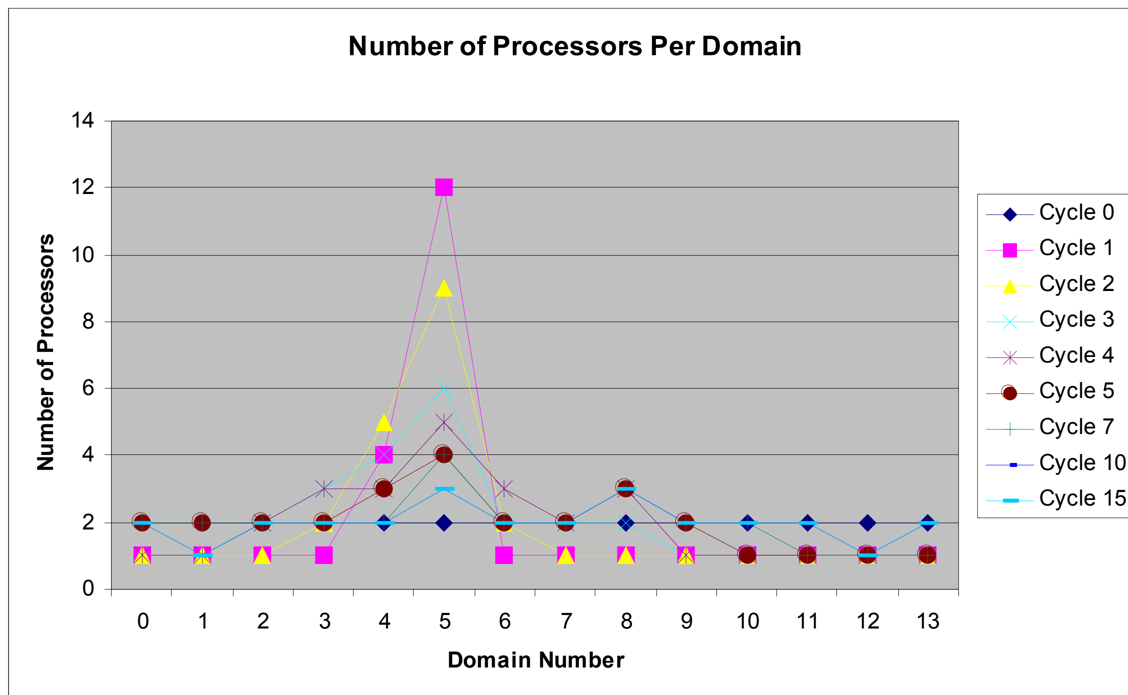


Figure 9. Pseudocolor plots of the particle number density for six cycles during the evolution in the critical assembly test problem. The particle density increases as the color changes from blue to red. The domain boundaries are indicated by the black lines.

The run in which the dynamic load balancing feature was disabled had a uniform, static replication level (work group) of two processors assigned to each domain. When this feature is enabled, the size of the work group assigned to each domain varies in accordance with the work load per domain, as shown in Figure 10. The initial load imbalance (see the Cycle 1 plot in Figure 9) results in 6 processors being assigned to work on the domain containing the source cavity (Domain 3) during cycle 1: a 3-to-1 max-to-mean domain processor count. As the work load evens out over time, the replication level becomes less peaked at the center of the system, such that by cycle 25, the central Domains 2 and 3 are assigned 3 processors, while the peripheral Domains 0 and 7 are assigned 1 processor, with the remaining domains being assigned 2 processors: a 1.5-to-1 max-to-mean ratio.

The non-load-balanced and load balanced per-cycle run times and the parallel efficiencies are presented as a function of the cycle count in Figure 11. These figures shows only the first 14 iterative cycles. When the load balancing feature is enabled, the per-cycle run times are reduced by more than a factor of 2 for Cycle 2, and the reduction in run time is a factor of 1.3 at Cycle 14. Similarly, the parallel efficiency is increased by a factor of 2.4 at Cycle 2, and then falls off to a factor of 1.2 at Cycle 14. The cumulative run time is reduced by 39% when the load balancer is enabled, as shown in Table 1.



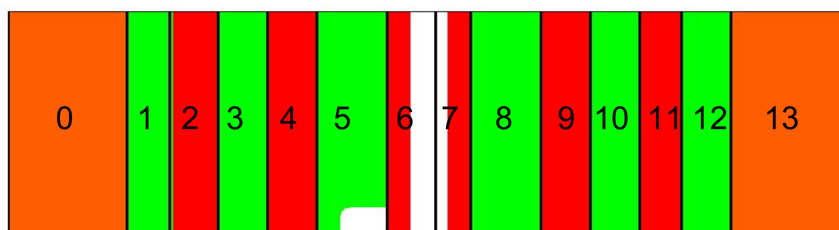
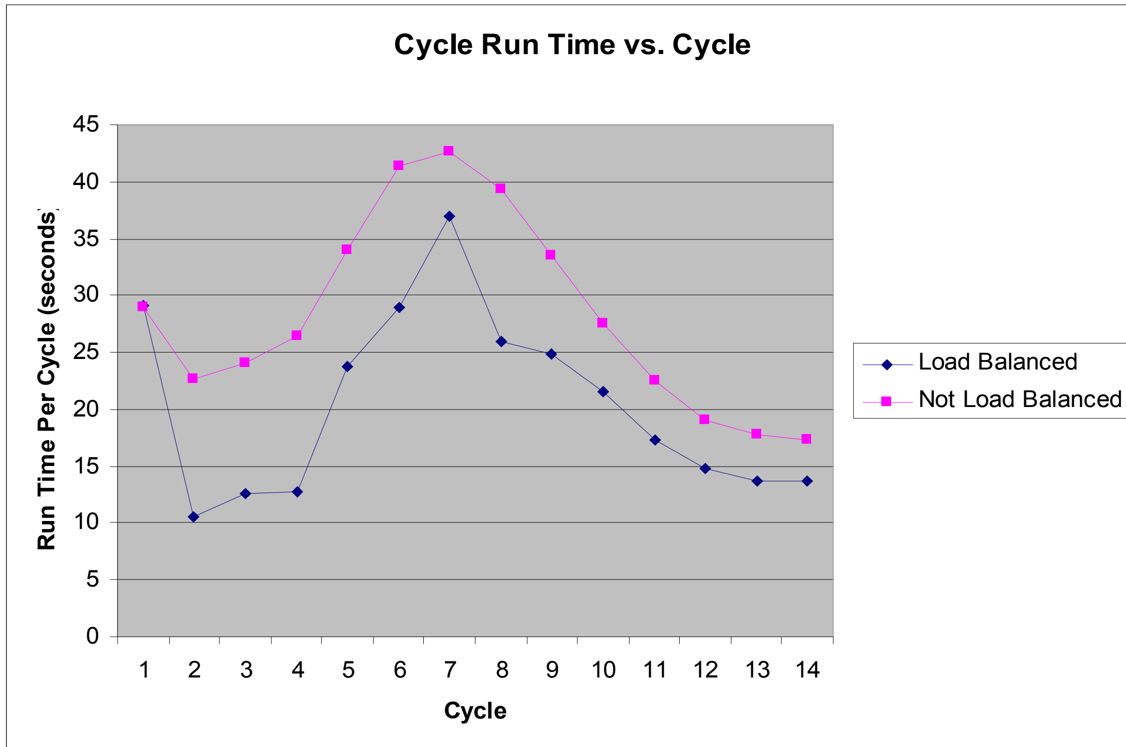


Figure 10. Evolution of the number of processors assigned to each domain (replication level) at six cycles during the iterative calculation. The initial load imbalance results in 6 processors being assigned to work on the domain containing the source cavity during cycle 1. Later, as the work load evens out, the replication level becomes less peaked at the center of the system.

Table 1. Critical Assembly Test Problem Cumulative Run Times

	Problem Run Time (sec)		
Cycle Range	**Not Load Balanced**	**Load Balanced**	**Speedup**
1 to 4	102.2	65.0	1.57
1 to 14	397.1	286.4	1.39



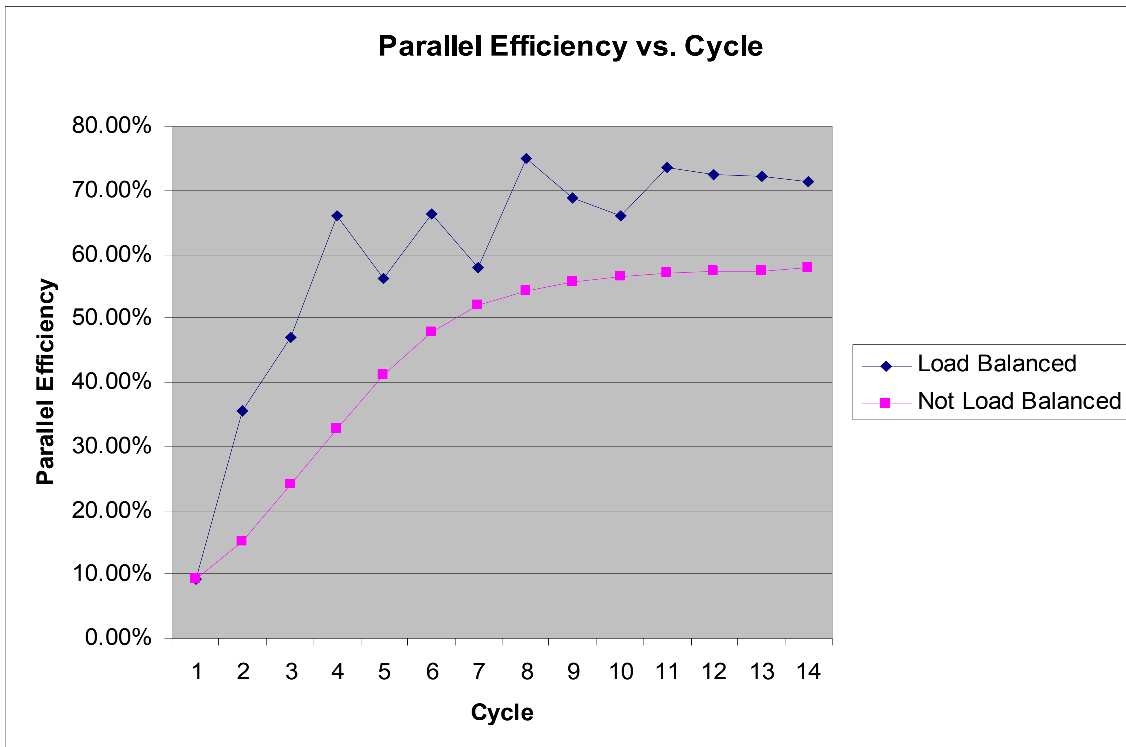


Figure 11. Per-cycle run time and parallel efficiency as a function of cycle from the calculations of the critical assembly test problem. The non-load-balanced results are shown in pink, while the load balanced results are shown in blue.

Sourced Test Problem

The time-dependent sourced problem chosen for this test is a spherized version of a shielding configuration that has been considered as a candidate for the neutron shield surrounding a fusion reactor. The shield consists of alternating layers of stainless steel and borated polyethylene, as shown in Figure 12. The radius of the inner steel sphere is $r = 35.56$ cm, and the thickness of each of the the other shells is $\delta = 5.08$ cm. Monoenergetic $E = 14.1$ MeV particles are sourced into the center of the system from an isotropic point source. During each of the the first 200 cycles, $N'_p = 1 \times 10^5$ particles are injected into the system. The source is then shut off, and the particles continue to flow through the shield for the next 800 cycles. The size of the time step is $\Delta t = 1 \times 10^{-8}$ sec.

This problem was also run on a 2-D $r - z$ mesh that was spatially decomposed in 4 domains, 2 domains along each of the axes. Parallel calculations were made using 16 processors of the White machine. The main tally for these calculations is the time history of the particles leaking across the outer steel layer into the air.

The evolution of the particle positions is shown for six cycles in the scatter plots of Figure 13. The particles are color coded according to their kinetic energy: red is 14.1 MeV, green is 1×10^{-5} MeV and cyan is 1×10^{-8} Mev. The interior (exterior) shell boundaries are shown in blue (red), while the domain boundaries are shown in black. It is clear from these figures that the majority of the particle work load “flows” from the bottom-left domain (Domain 0) early in time (Cycle 2), to the domains on the periphery (Domains 1 and 2) later in time (Cycles 301 through 1001).

The run in which the dynamic load balancing feature was disabled had a uniform, static work group of four processors for each domain. Enabling the load balancer gave work groups whose sized varied with the work load per domain, as shown in Figure 14. The initial load imbalance (see the Cycle 2 plot in Figure 13) results in 13 processors being assigned to work on the domain containing the source cavity (Domain 3) during cycle 1: a 3.25-to-1 max-to-mean domain processor count. As the work load evens out over time, especially once the source is turned off, the replication level becomes less peaked at the center of the system. By cycle 300, the near-peripheral Domains 1 and 2 are assigned 4 and 5 processors, while the center Domains 0 is assigned 6 processors, with the far-peripheral Domain 3 being assigned 1 processor: a 1.5-to-1 max-to-mean ratio.

The per-cycle run time and the parallel efficiency of the non-load-balanced and load balanced calculations are shown as a function of the cycle count in Figure 15. During the first 200 cycles, while the source is turned on, the figures indicates that the run time is high and the efficiency is low for the non-load-balanced calculation. By enabling the load balancing feature, these per-cycle run times are reduced by more than a factor of 2, and the efficiency is increased by more than a factor of 2.5. Once the source is turned off, the benefit of load balancing is not as substantial, however, the efficiency of the load-balanced calculation still exceeds that of the non-load-balanced calculation by a factor of 1.3 to 1.8. This trend is also seen in the data presented in Table 2.

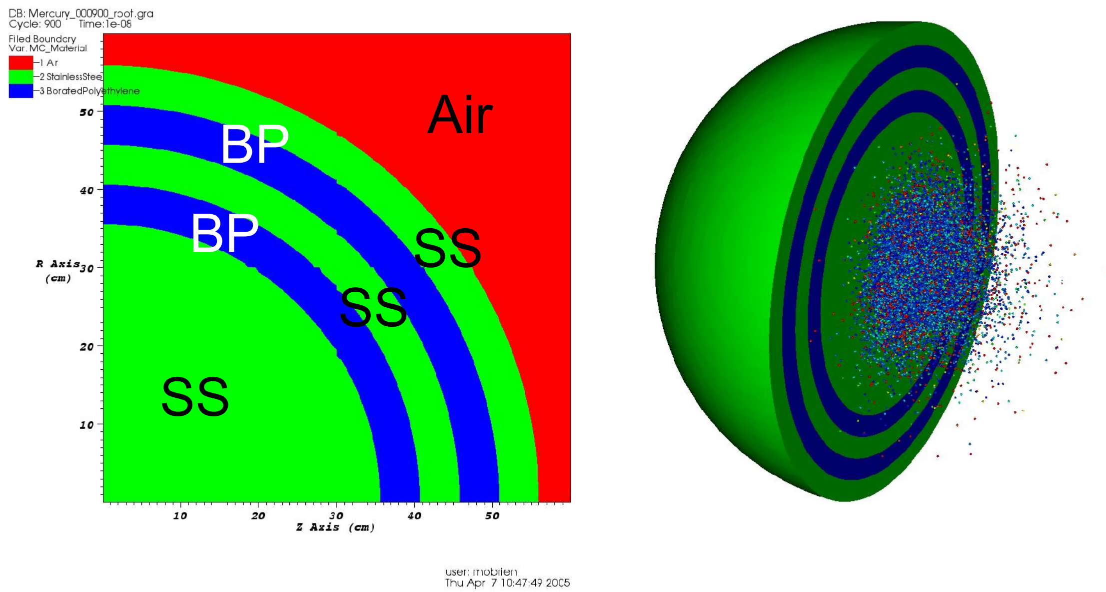


Figure 12. Geometry of the spherical-shield sourced test problem. The green areas are stainless steel and the blue areas are borated polyethylene. The dots on the right-hand image represent particles which are color coded by kinetic energy (blue is low, red is high).

Table 2. Spherical Shield Test Problem Cumulative Run Times

	Problem Run Time (sec)		
Cycle Range	**Not Load Balanced**	**Load Balanced**	**Speedup**
1 to 201	1355	615	2.20
1 to 1001	2221	1404	1.58

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 the parallel calculation of one criticality and one sourced problem, where it has yielded more than a two-fold increase in the parallel efficiency.

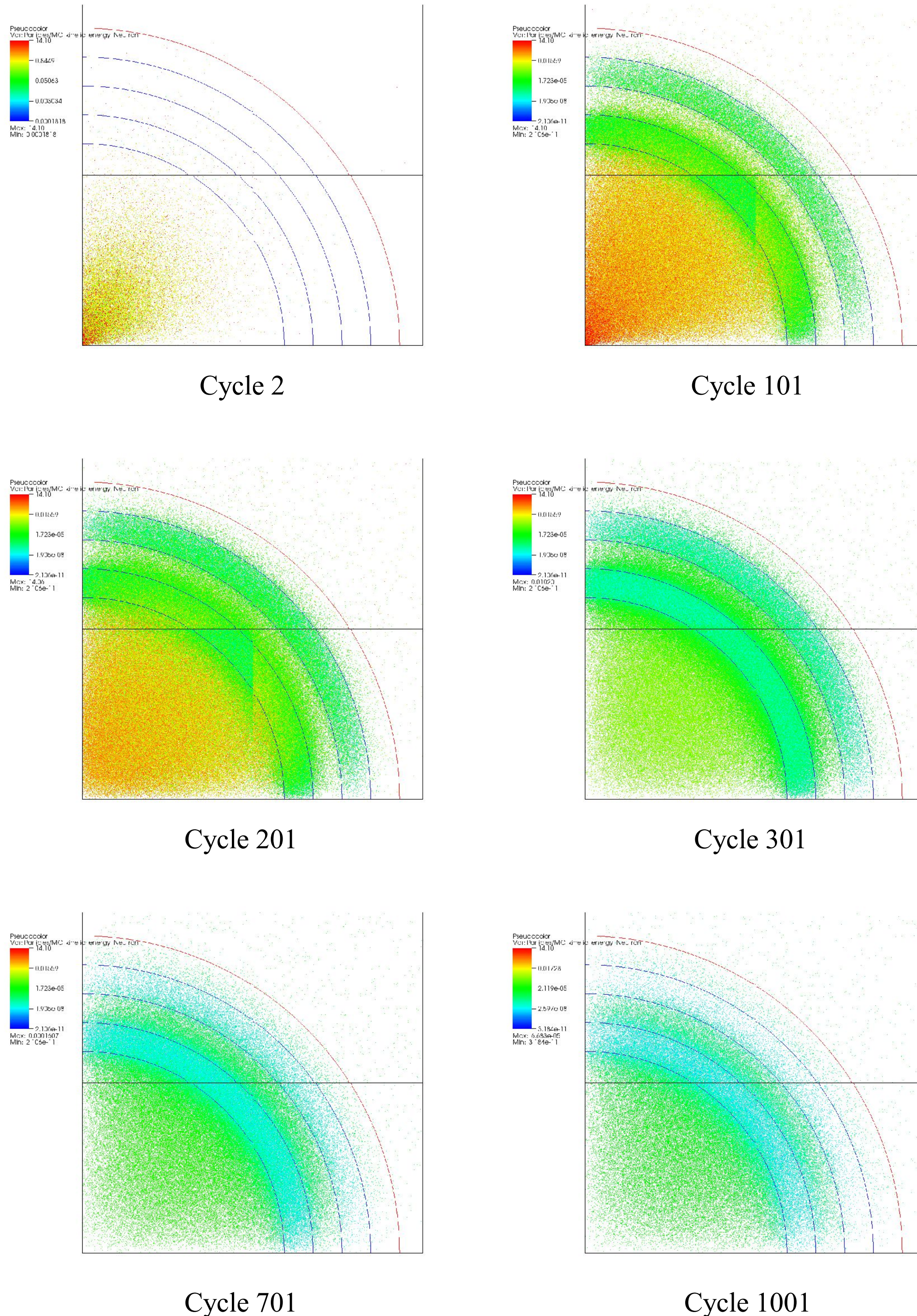


Figure 13. Scatter plots of the particle positions for six cycles during the evolution of the spherical shield test problem. The boundaries of the spherical shells are shown in blue or red, while the domain boundaries are indicated by the black lines.

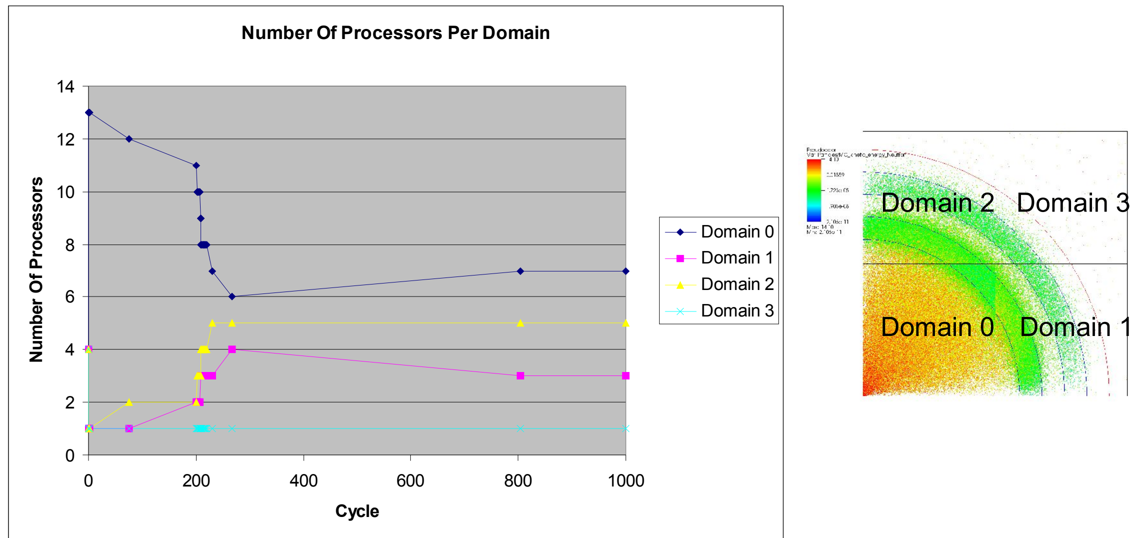


Figure 14. Evolution of the number of processors assigned to each domain (replication level) at six cycles during the time dependent calculation. The initial load imbalance results in 13 processors being assigned to work on the center domain during cycle 1. Later, as the work load evens out, the replication level becomes less peaked at the center of the system.

Acknowledgments

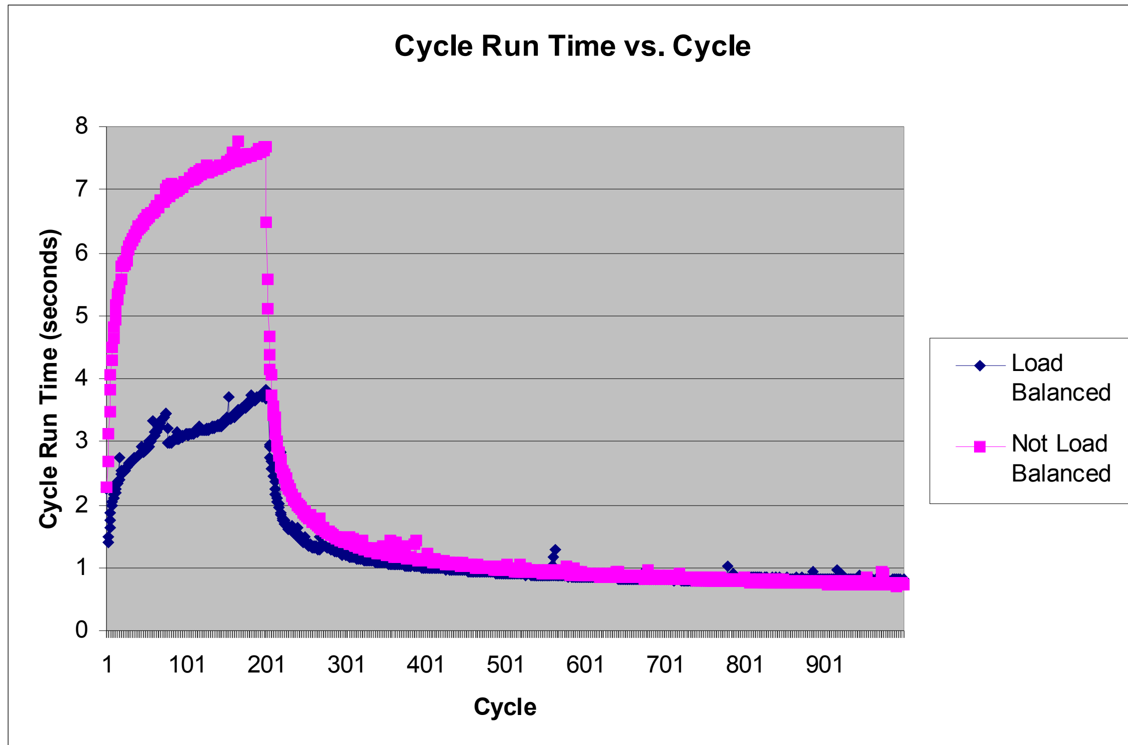
This work was performed under the auspices of the U.S. Department of Energy at the Lawrence Livermore National Laboratory under Contract Number W-7405-Eng-48.

References

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

Procassini, R. J. and Taylor, J. M., *Mercury User Guide (Version b.8)*, Lawrence Livermore National Laboratory, Report UCRL-TM-204296 (2005).

The International Critical Safety Benchmark Evaluation Program, *International Handbook of Evaluated Criticality Safety Benchmark Experiments*, NEA Nuclear Science Committee, Nuclear Energy Agency, Report NEA/NSC/DOC(95)03 (2004).



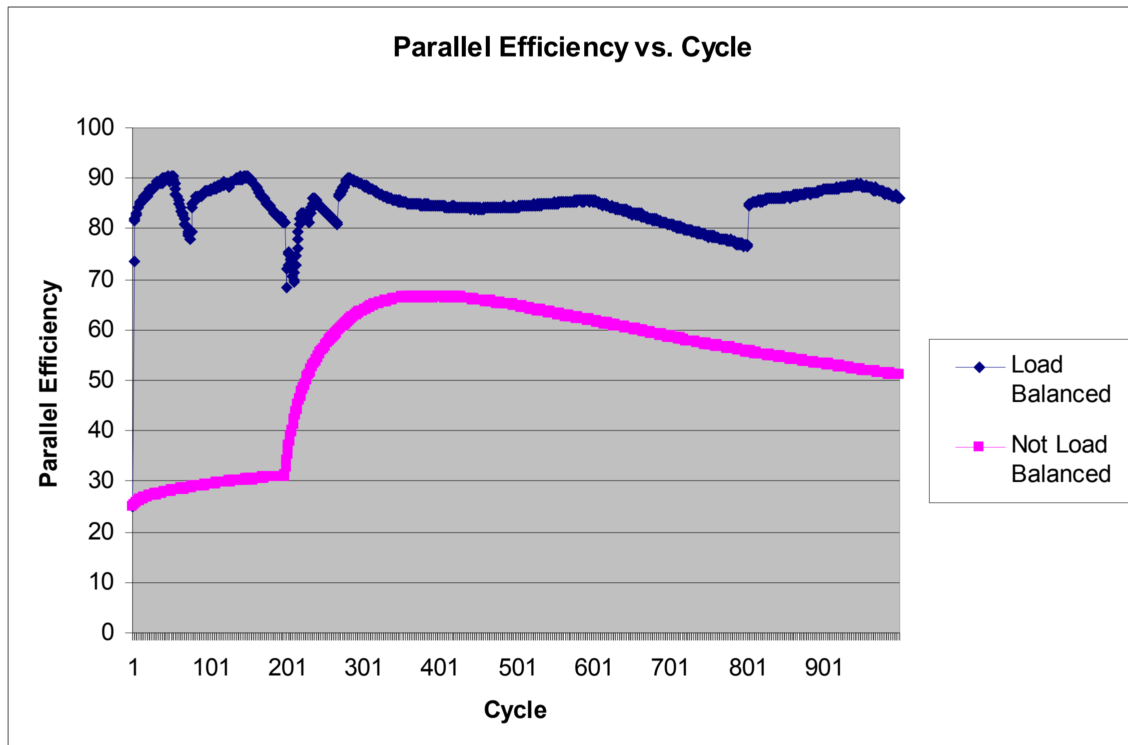


Figure 15. Per-cycle run time and parallel efficiency as a function of cycle from the calculations of the spherical shield test problem. The non-load-balanced results are shown in pink, while the load balanced results are shown in blue.