LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-666780

Particle Communication and Domain Neighbor Coupling: Scalable Domain Decomposed Algorithms for Monte Carlo Particle Transport

M. J. O'Brien, P. S. Brantley

January 30, 2015

NECDC
Los Alamos, NM, United States
October 20, 2014 through October 24, 2014

Disclaimer

Particle Communication and Domain Neighbor Coupling: Scalable Domain Decomposed Algorithms for Monte Carlo Particle Transport

Matthew O'Brien and Patrick Brantley
Lawrence Livermore National Laboratory, Livermore, CA

Topic 20.3: Computer Science: Performance/Scaling/Optimization.

October 20, 2014

In order to run Monte Carlo particle transport calculations on new supercomputers with hundreds of thousands or millions of processors, care must be taken to implement scalable algorithms. This means the algorithms must continue to perform well as the processor count increases. In this paper, we examine the scalability of: 1) globally resolving the particle locations on the correct processor, 2) deciding that particle streaming communication has finished, and 3) efficiently coupling neighbor domains together with different replication levels.

We have run domain decomposed Monte Carlo particle transport on up to 2^{21} = 2,097,152 MPI processes on the IBM BG/Q Sequoia supercomputer and observed scalable results that agree with our theoretical predictions. These calculations were carefully constructed to have the same amount of work on every processor, i.e. the calculation is already load balanced. We also examine load imbalanced calculations where each domain's replication level is proportional to its particle workload. In this case we show how to efficiently couple together adjacent domains to maintain within workgroup load balance and minimize memory usage.

Introduction

Mercury is LLNL's current generation Monte Carlo particle transport code. Some of Mercury's main features are:

- Solves dynamic neutron transport and criticality eigenvalue problems
- Also has charged particle and gamma transport capability
- Written in C++, python user interface, massively parallel, distributed memory MPI and shared memory OpenMP
- Parallelized via domain decomposition and domain replication

Mercury has been run domain decomposed on 2^{21} = 2,097,152 MPI processes on the IBM BG/Q Sequoia supercomputer. Many of the major parallel algorithms have been rewritten to be scalable to large processor counts. We define an algorithm to be *scalable* if it continues to perform well as the number of processors increases. An example of a scalable algorithm would have run time proportional to the logarithm of the number of processors. An example of a non-scalable algorithm would be an algorithm that has run time proportional to the number of processors. Practically speaking, as long as an algorithm consumes a small enough fraction of the

total run time, we may tolerate non-scalable algorithms if they do not affect overall performance significantly. If a non-scalable algorithm takes a significant fraction of the total runtime, then the algorithm needs to be rewritten to be scalable (if possible).

This paper discusses some of the parallel algorithms that we have rewritten to be scalable. The first section describes the parallel MPI model used in Mercury. The next section describes globally resolving particle locations on the correct processor. This can happen for "source" particles that may be created on *any* processor through sampling a user defined probability distribution function. The source particle must be communicated to the processor that owns the background domain before particle transport can take place.

The next section is on deciding when particle streaming communication has finished. The algorithm has asynchronous particle streaming communication along with processing particles and sending messages up and down a communication tree to decide if every particle that started has finished tracking.

The last section describes "domain neighbor replication coupling", which considers how to efficiently couple adjacent domains that have different replication levels (the *replication level* of a domain is how many processors are assigned to the domain to evenly divide the particle workload for that domain). Finally we present our conclusions and comment on the future challenges that face the project.

Table 1: Modern Supercomputing History at Lawrence Livermore National Laboratory.

Date	Computer	Number of Processors
1998	Blue	5,856
2001	White	8,192
2005	Purple	12,544
2004	BlueGene/L	65,636
2009	Dawn	147,456
2012	Sequoia	1,572,864

Mercury's Parallel MPI Model

Mercury has two parallel models: domain decomposition and domain replication. Mercury also has a hybrid domain decomposition + domain replication parallel model that is used for load balancing the problem.

- Domain decomposition partitions geometry.
- Domain replication
    - Distributes particles *within* a domain.
    - Load balancing: Replication level is proportional to particle workload.

Figure 1 plots the Monte Carlo particles by processor and provides an illustration of Mercury's parallel model. The problem has 6 domains and is run on 12 processors. Particles are colored by processor. Domains that have more computational work have a higher replication level, and we see more colors in those domains, representing the processors working on those particles.

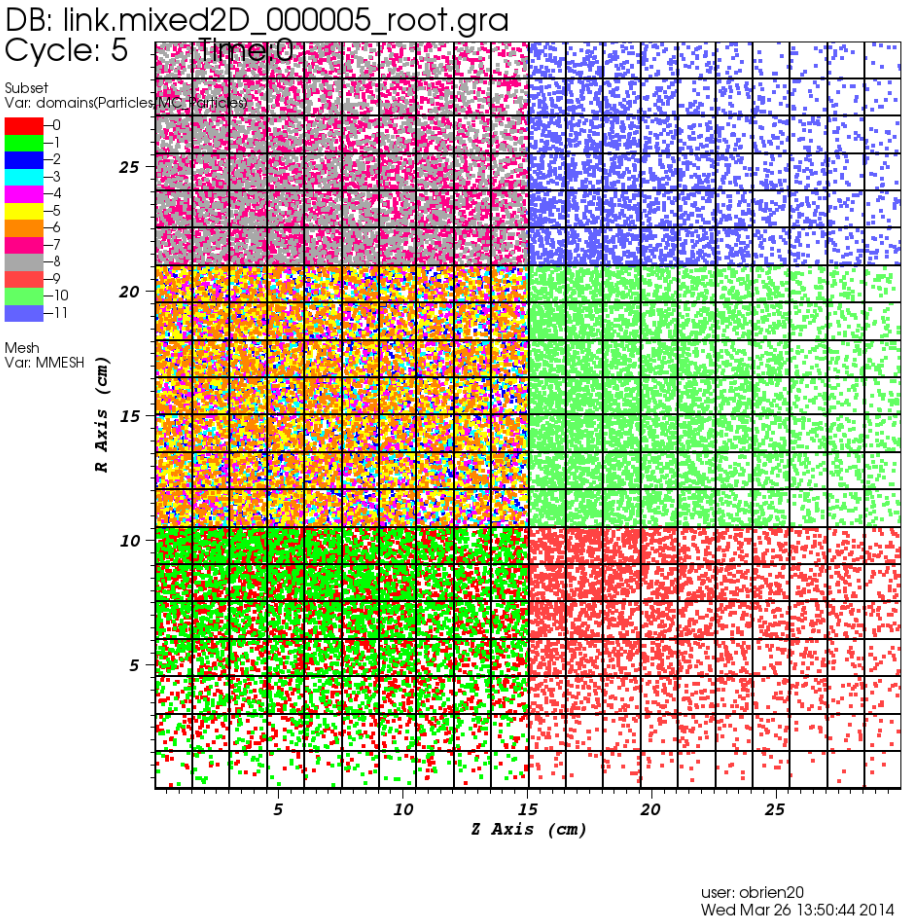


Figure 1: Particles are colored by processor. Some domains have more processors than others, depending on the domain's workload.

Globally Resolving Particle Locations On The Correct Processor

This section examines the question of how to communicate particles to the correct processor if they were created on another processor. This is not particle *streaming* communication where particles travel from one domain to an adjacent domain. Here we are considering creating a particle whose coordinate was created by sampling from a user defined probability distribution function (PDF). All processors sample particles from the PDF and the coordinate may end up anywhere in the problem, not necessarily on the domain that the processor owns. So particles need to be communicated to the processor that owns the background domain for that particle.

Here is the algorithm for communicating particles to the correct processor:

- Each processor has the bounding box of *every* domain.
- If a processor does not own a particle, send the particle to all ***candidate*** domains (the particle is within the domain's bounding box).
- Instead of linearly searching through all bounding boxes, store the bounding boxes in a space partitioning tree, enabling faster searches.

Improved Searching for Candidate Domains:

- Old algorithm linearly searched through all of the bounding boxes of every domain.
- New algorithm stores the domain bounding boxes in a space partitioning tree for faster searches. (Disadvantage: still need to ***store all*** of the bounding boxes of every domain)
- Partition 1 dimension at a time. A bounding box becomes an interval in 1 dimension.

```
Search Algorithm():
      if leaf node:
            search through intervals
      else:
            if coord < partition:
                  search left
            else:
                  search right
```

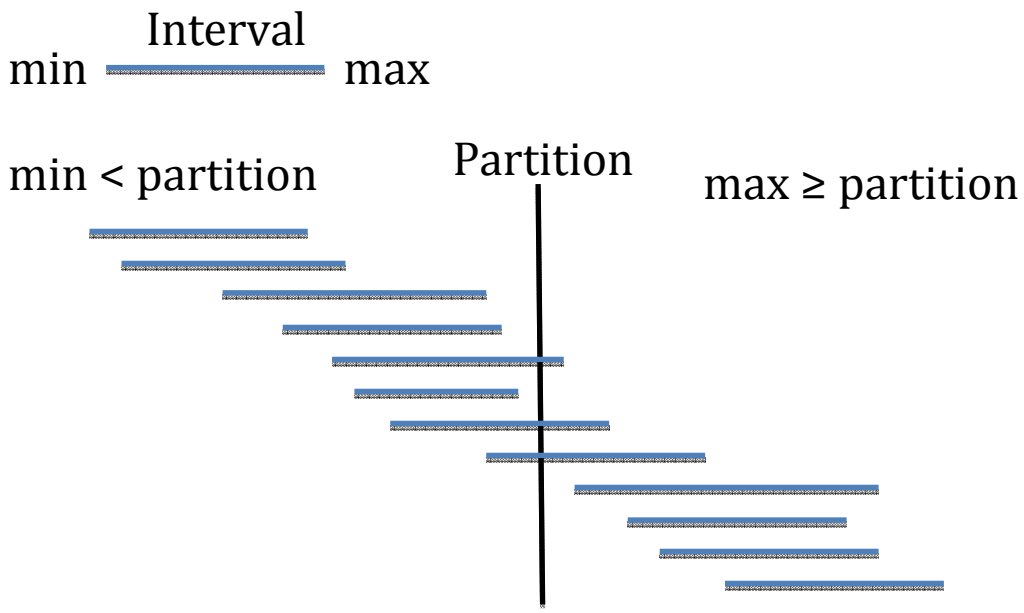


Figure 2: Illustration of dividing 1 dimensional intervals by a partition. Store intervals in "left" child if min < partition and store intervals in "right" child if max ≥ partition. Some intervals will be stored in both "left" and "right" children.

To investigate the performance of the search algorithms, we use a spherical surface source test problem as shown in Figure 3. We run a scaling study by increasing the number of domains from 400 to 102,400.

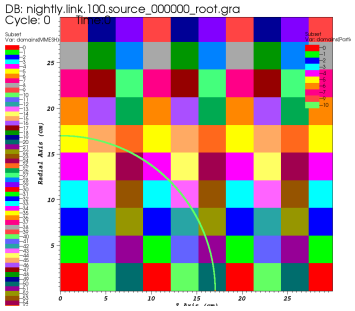


Figure 3: Green circle shows the position of the spherical surface source. Background geometry is colored by domain.

Figure 4 shows the scaling timing results. We are comparing the new *Tree Search* method with the old *Linear Search* method. The new method scales much better. The old linear search method has to search through all of the domains to find the candidate domains that contain the particle's coordinate. The tree search method can ignore more of the domains by ruling them out as impossible intersections.

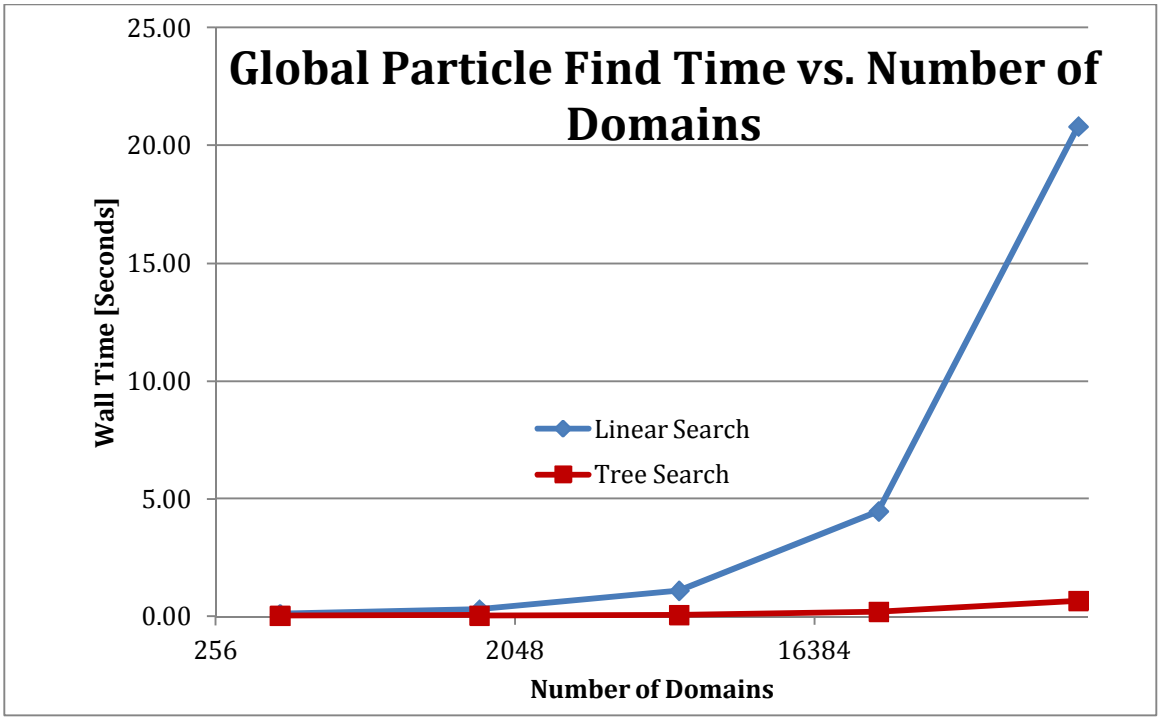


Figure 4: Scaling study results for new *Tree Search* compared to old *Linear Search*.

Table 2 shows the data from Figure 4 along with the speedup. For 102,400 domains, the Tree search is 30 times faster than the linear search!

Table 2: Timing data from scaling study.

Domains	Linear Search [seconds]	Tree Search [seconds]	Speedup
400	0.11	0.05	2.3
1,600	0.31	0.06	5.5
6,400	1.12	0.09	12.5
25,600	4.48	0.22	20.4
102,400	20.81	0.69	30.3

Figure 4 and Table 2 are reporting on the *total* time for comparing the two algorithms. The total time has 2 components, initialization time (to construct the data structure for searching) and search time (the time to actually search through the data structure).

Total Time = Initialization Time + Search Time

Figure 5 examines each of these components independently. The initialization time for the tree algorithm is slower than the linear algorithm (orange curves) since we must construct a k-d tree out of the domain bounding boxes. But once the data structures are constructed, searching the tree is much faster than the linear search (purple curves).

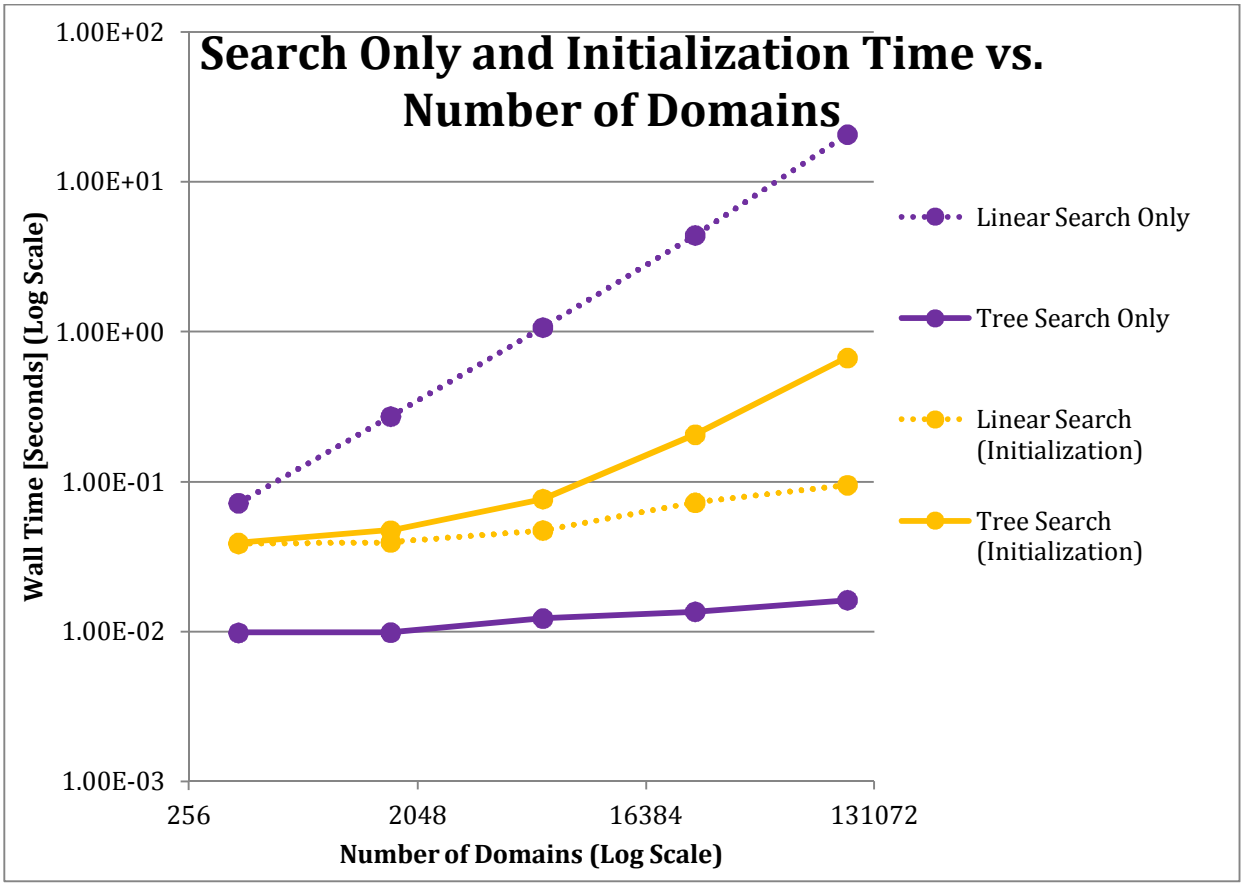


Figure 5: Scaling study results for initialization and search phases of calculation.

Table 3 shows the detailed timing breakdown of the search phase only and the initialization phase only. At 102,400 domains, the search only phase is 1,274 times faster for the tree algorithm compared to the linear search algorithm. But it is more expensive to construct the tree than the linear data structure, so the *total* speedup is only 30X.

Table 3: Detailed timing data from scaling study.

Domains	Linear (total)	Tree (total)	Speedup (total)	Linear Search Only	Tree Search Only	Speedup (Search Only)	Linear (Initialization)	Tree (Initialization)
400	0.11	0.05	2.3	0.07	0.01	7.3	0.04	0.04
1,600	0.31	0.06	5.5	0.27	0.01	27.6	0.04	0.05
6,400	1.12	0.09	12.5	1.07	0.01	87.3	0.05	0.08
25,600	4.48	0.22	20.4	4.41	0.01	324.1	0.07	0.21
102,400	20.81	0.69	30.3	20.71	0.02	1273.8	0.10	0.67

Deciding That Particle Streaming Communication Has Finished

Each processor is processing the particles that it owns and asynchronously sending/receiving MPI messages to/from adjacent domains when particles reach domain boundaries. It is difficult to know when the calculation has finished. A processor may have finished *all* of the particles it owns, but it does not know if it will receive more incoming particles. One way to know that the calculation has completed is when the total number of particles started (summed over all processors) equals the total number of particles finished (summed over all processors). The idea is to try to maximize the overlap of communication and computation so the calculation can run efficiently.

We are using an algorithm based on Brunner and Brantley, 2009. The idea of the algorithm is to use non-blocking reduce and broadcast operations concurrently as particles are tracking. We have found that this algorithm works well for load balanced problems, but load imbalanced problems can introduce difficulties. A parent and child in the communication tree may have different workloads which may cause the child to send too many MPI messages to the parent, exhausting internal MPI resources. We have tried to ameliorate this by introducing a user-settable parameter: *Issend_Period*, which controls how often we use MPI_Issend (note the extra "s") instead of MPI_Isend. MPI_Issend() enforces synchronization between the sender and receiver (receiver must have started receiving before MPI_Issend() completes), whereas MPI_Isend() can return a request handle that has completed as long as MPI has buffered the message, making no claim at all about the receiver.

Figure 6 shows scaling study results for an idealized, perfectly load balanced problem run up to 2^{21} = 2,097,152 domains (each run has: number of domains = number of processors).

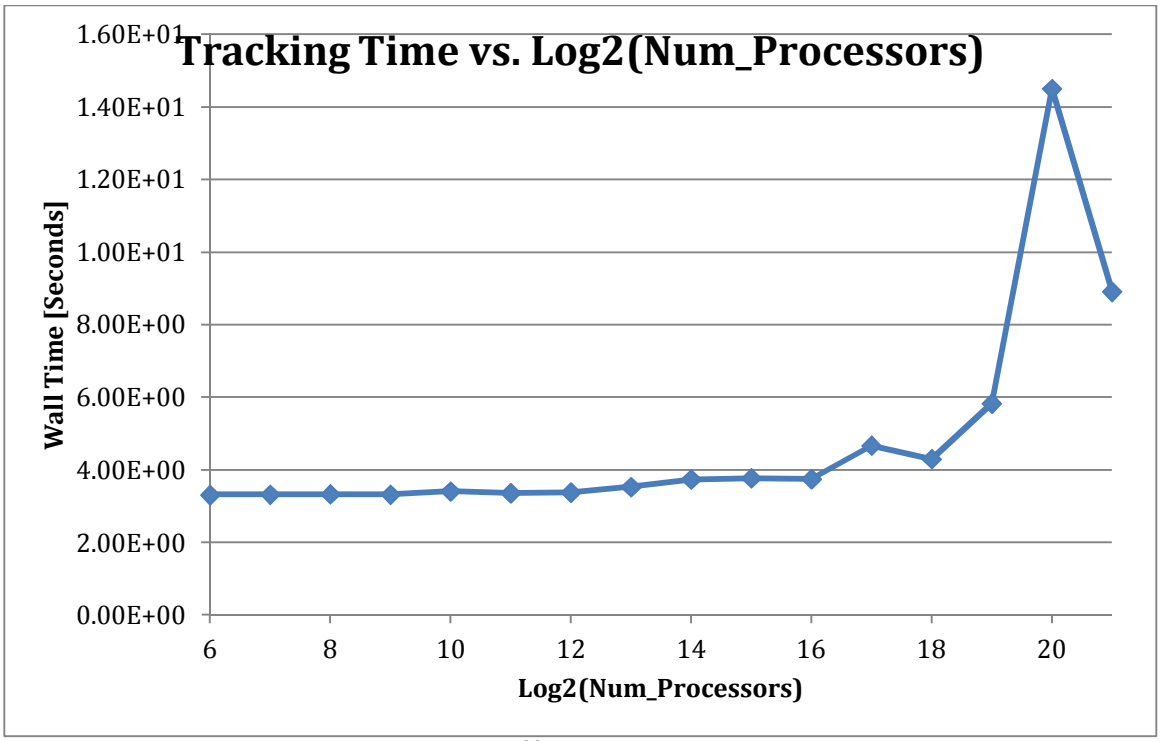


Figure 6: Weak scaling study up to 2^{21} = 2,097,152 processors for a load balanced test problem.

We have also implemented a very simple blocking test for done algorithm that just calls MPI_Allreduce() to compute the total number of particles that started tracking and the total number of particles that finished tracking. When these two numbers are equal (summing over all processors), then the calculation has finished. This algorithm is beautifully simple, but MPI_Allreduce() is a blocking algorithm which does not allow particle streaming communication or particle tracking to take place until the algorithm returns. MPI 3.0 has non-blocking collective calls, so we use MPI_Iallreduce() when it is available. The Sequoia supercomputer does not use MPI 3.0 by default so we have not used the non-blocking MPI_Iallreduce() on Sequoia yet. If the non-blocking particle start count equals the non-blocking particle finished count, then we call the blocking MPI_Allreduce() to make sure the counts agree. Particle transport is done when the blocking counts agree.

We have run small scaling studies with the MPI 3.0 non-blocking collective call MPI_Iallreduce. The results look promising. We plan on running a large scaling study using the MPI_Iallreduce algorithm. If that algorithm scales well, then we will switch to that as our default algorithm. This algorithm is very simple to implement, and we hope that it will scale similarly to the more complicated "hand coded" Brunner and Brantley algorithm. So we hope to have the best of both worlds: simplicity and good performance.

Neighbor Replication Coupling

If two adjacent domains have the same replication level, then there is a simple and efficient way to couple them together: have each replica of one domain couple to exactly one replica of the other domain. But if the replication levels are unequal for adjacent domains, then it

is more challenging to come up with a memory efficient and load balanced coupling of the adjacent domains.

Figure 7 shows two adjacent domains with each domain replicated 4 times. Figure 7(b) shows a "sparse" way to couple the replicas together and Figure 7(c) shows a full "crossbar" coupling between the domains. The crossbar coupling is load balanced for uneven replication levels, but it requires a large amount of memory.

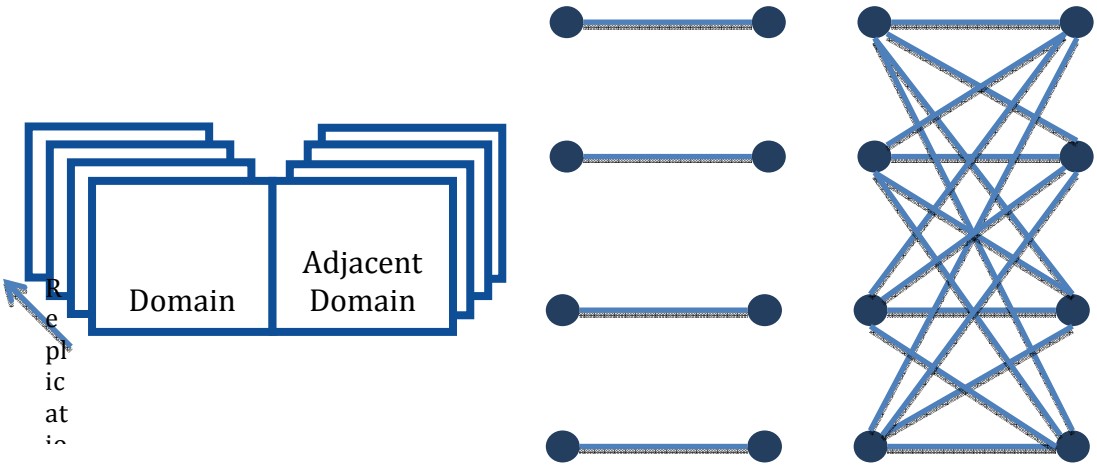


Figure 7: (a) Domain replication coupling. (b) Sparse. (c) Crossbar.

Within cycle neighbor induced load imbalance can happen for poorly constructed sparse communication graphs. Figure 8 shows this where one domain is replicated 5 times and an adjacent domain is replicated 4 times. In the case of uneven domain replication coupling, it is possible to have two domains communicating all of their particles to a single adjacent domain, which ends up giving twice as much work to *one* replica of the adjacent domain, which causes *within cycle* neighbor induced load imbalance.

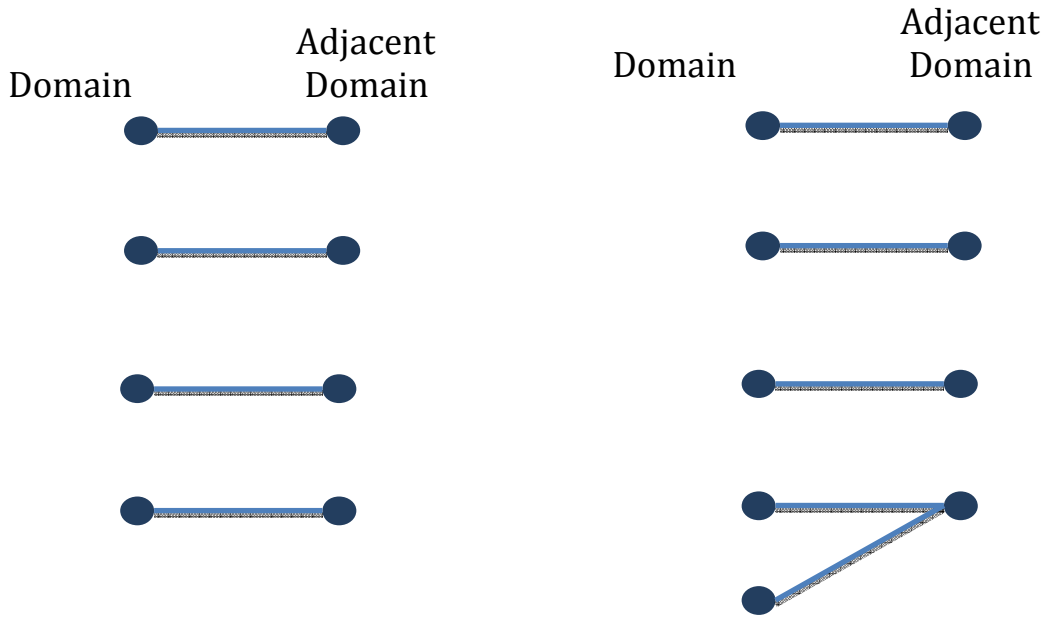


Figure 8: (a) Equal domain replication coupling and (b) unequal domain replication coupling.

Figure 9 shows that crossbar coupling has better parallel efficiency than sparse coupling. This problem has a fixed number of foremen, 32, so by increasing the number of processors we give the load balancer more worker processors to do a better job of load balancing. Parallel efficiency should *increase* with number of processors. Parallel efficiency does increase for the Crossbar algorithm, but it decreases with the Sparse algorithm. This is an indication of the within cycle neighbor induced load imbalance.

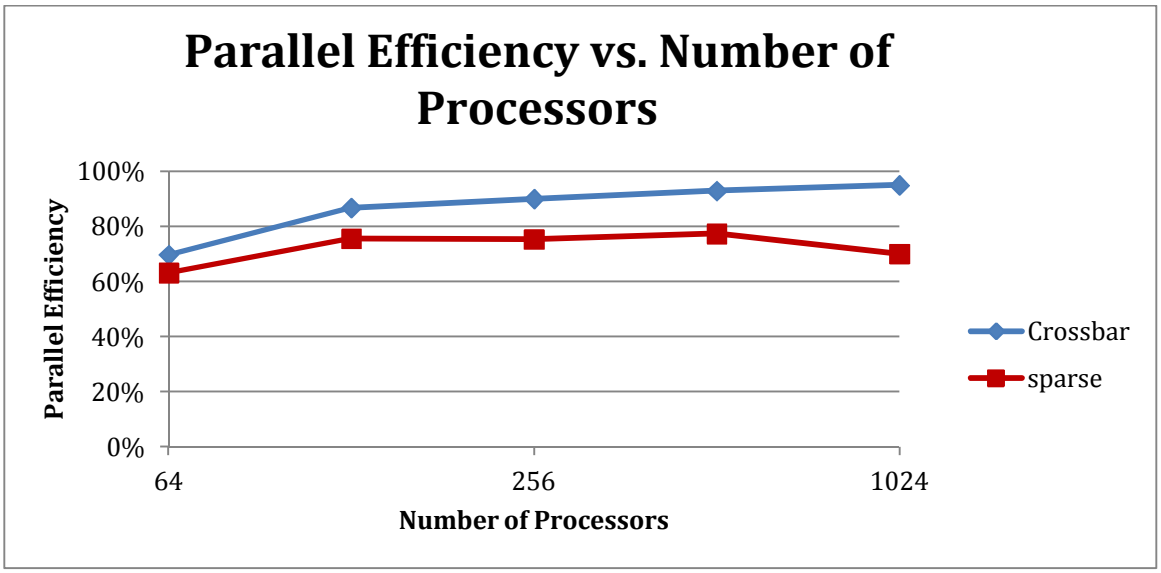


Figure 9: Parallel efficiency scaling study comparing Crossbar and Sparse neighbor replication coupling.

Figure 10 shows the crossbar coupling requires an ever increasing amount of memory as we increase the replication level in this test problem. The average memory usage for the sparse algorithm is acceptable, but the max memory usage for the sparse algorithm is still scaling poorly.

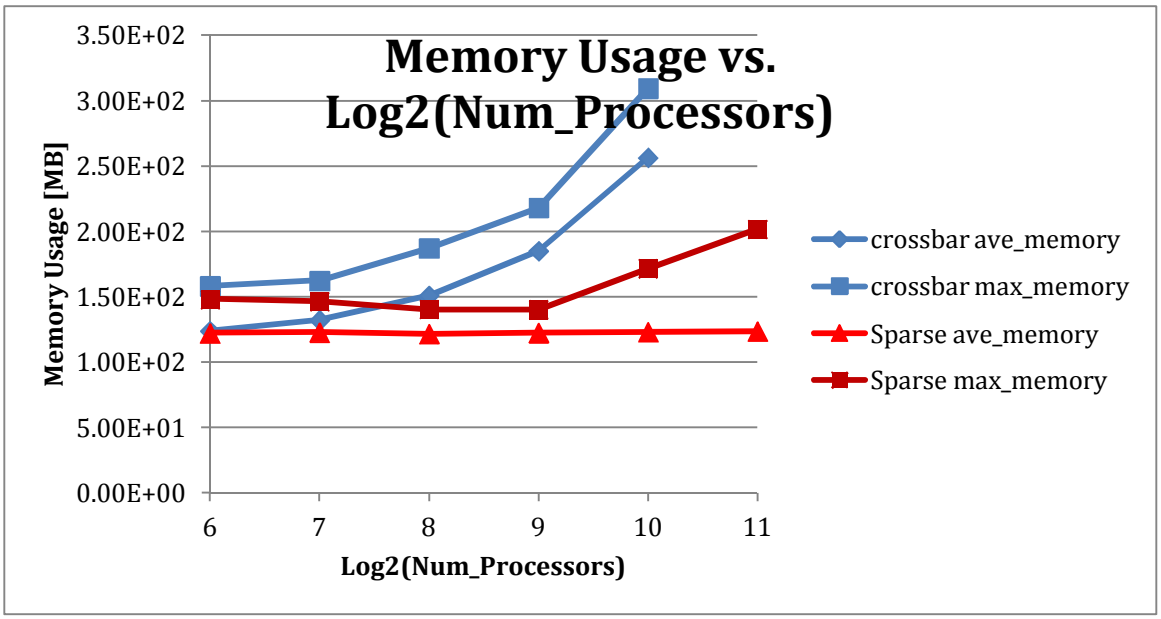
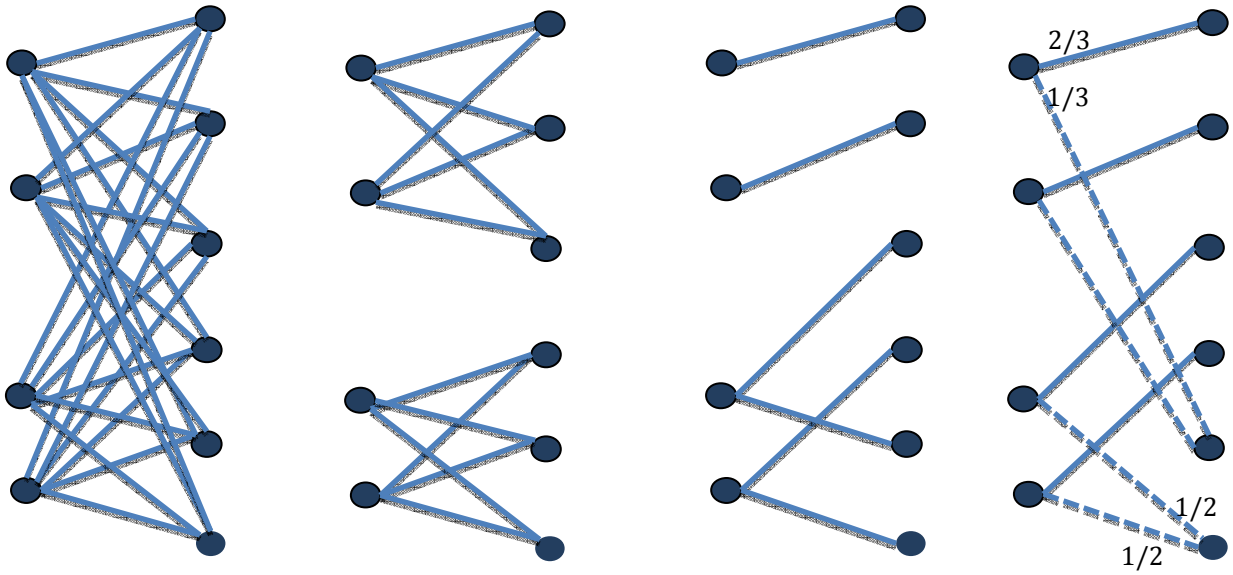


Figure 10: Memory usage scaling study comparing Crossbar and Sparse neighbor replication coupling.

So our goal is to develop an algorithm that minimizes memory usage (the number of connection edges) and is 100% load balanced when sending particles to adjacent domains. This requires sending an equal number of particles to each replica (which implies 100% load balanced within the group of replicated processors).

Figure 11 shows a comparison of four different algorithms for neighbor replication coupling. The first three algorithms all use *uniform* outgoing edge weights. That is, they simply "round robin" the outgoing particles evenly over all of the outgoing edges. The *Variable Weight* algorithm uses non-uniform outgoing weights that represent the probability that a particle gets sent along each edge. The outgoing weights are calculated so that the total incoming weight to each adjacent domain is equal, so we have 100% load balance efficiency within each workgroup. Note that Figure 11(d) shows *outgoing* edge weights. The sum of the *incoming* edge weight should be equal for all replicas on the left side (∑incoming=1.5), and equal for all replicas on the right side (∑incoming=2/3).



(a) Crossbar (b) Divide and Conquer (c) Sparse (d) Variable Weight

Figure 11: Four different neighbor replication methods.

Variable weight improves parallel efficiency and reduces memory usage. The variable weight algorithm is very similar to the recursive Euclidean Algorithm for computing the *greatest common divisor GCD* of two numbers (m, n). m is the replication level of one domain and n is the replication level of an adjacent domain. Without loss of generality, assume $m \leq n$. If m divides n evenly, then just assign n/m outgoing edges from m to n. But if m does not evenly divide n, then we have to deal with the remainder and fractional weight edges.

Figure 12 shows the parallel efficiency for the 4 neighbor replication methods. We see that the Crossbar and Variable Weight methods have the highest parallel efficiency.

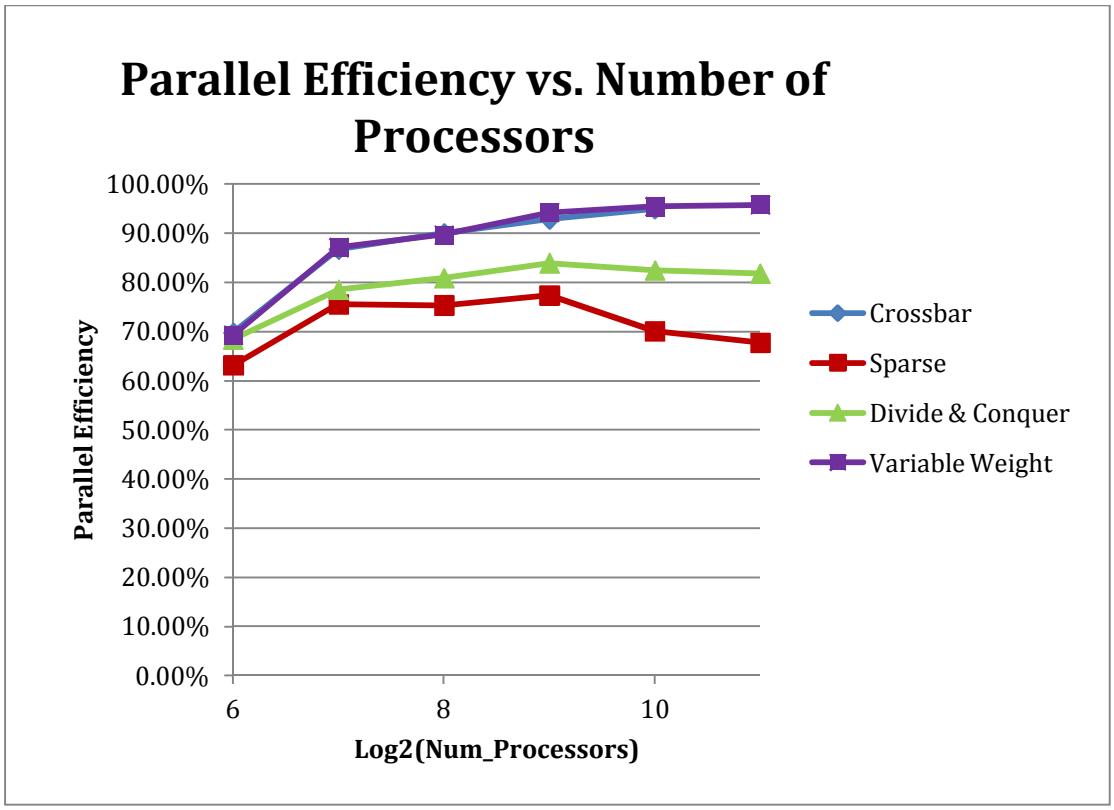


Figure 12: Parallel Efficiency for the four neighbor replication methods.

Figure 13 shows the Maximum Processor memory usage for each of the 4 neighbor replication methods. The Crossbar memory usage grows extremely quickly; the problem on 2^{12} processors could not run because it could not fit in memory. Even the variable weight method shows non-scaling behavior in the max processor memory usage. But this memory usage is not due to the variable weight algorithm, it is due to the num_foremen > 1 load balancing algorithm (we are currently working on improving the memory usage of that algorithm).

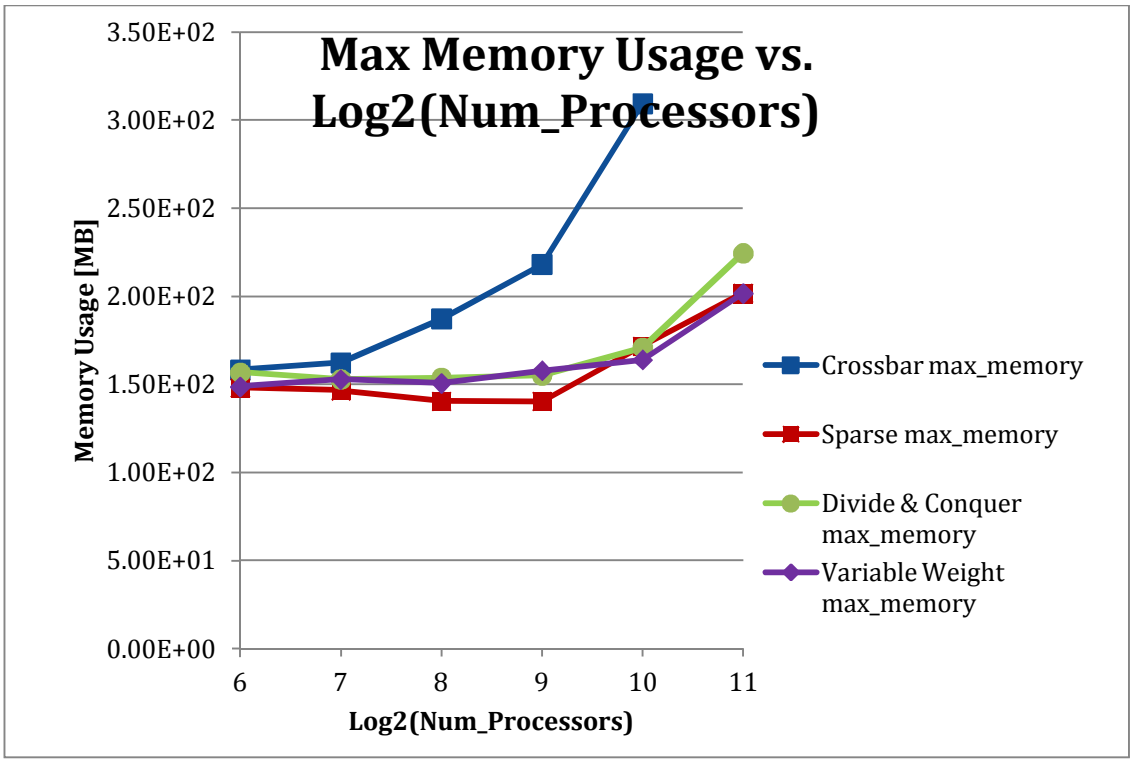


Figure 13: Max processor memory usage for the four neighbor replication methods.

Conclusions

We have rewritten our major parallel domain decomposed algorithms to be scalable. We discussed the following algorithms in this paper:

- Global Particle Find
- Test for Done
- Domain Neighbor Replication Coupling

A more complicated, multi-step "Global Particle Find" algorithm was discussed in previous papers, but in this paper we described a simple scalable "tree search" algorithm to search through a tree of every domain's bounding box. The construction of the tree is not scalable, since it requires assembling the bounding box of *every* domain in the problem, but the step is fast enough that we can tolerate it.

We have described a very simple MPI_Iallreduce() algorithm for deciding that particle streaming communication has finished. We have not yet carried out a large scaling study of this algorithm, but are hopeful that it will scale as well as the "hand coded" algorithm. We hope to replace the complicated "hand coded" algorithm with the MPI 3.0 call to MPI_Iallreduce().

The domain neighbor replication coupling turned out to be an important algorithm for scaling, and we were able to design an algorithm that was both 100% load balanced and minimized memory usage in terms of total number of connections to the adjacent domain. The algorithm uses a recursive Euclidean GCD type algorithm for constructing a bipartite graph between the adjacent domains, calculating variable edge weights to ensure load balancing for sending particles to the adjacent domain.

Some of our special purpose algorithms are still not completely scalable. Some aspects of these algorithms still need to be made scalable:

- Embedded Mesh/CG problems
- Criticality Probability and Extinction Probability
- Tracking on a mesh with gaps and overlaps

This will be the subject of future work as we attempt to make all aspects of the code scalable to large processor counts.

This paper has been released with document number LLNL-CONF-666780.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

1. P. Brantley and M. McKinley, "*Mercury Web Site*," (2011). [Online]. Available: https://wci.llnl.gov/codes/mercury/.
2. O'Brien, Matthew and Brantley, Patrick. *Scalable Algorithms for Domain Decomposed Monte Carlo Particle Transport.* Presented at Los Alamos National Laboratory, October 22, 2014. LLNL-PRES-784999.
3. O'Brien, Matthew. *Scalable Domain Decomposed Monte Carlo Particle Transport.* PhD Dissertation, UC Davis. LLNL-TH-647500. 2014.
4. Brunner, T. and Brantley, P. *An efficient, robust, domain-decomposed algorithm for particle Monte Carlo.* Journal of Computational Physics. *2009.*
5. G. Greenman, M. O'Brien, R. Procassini and K. Joy, "*Enhancements to the Combinatorial Geometry Particle Tracker in the Mercury Monte Carlo Transport Code: Embedded Meshes and Domain Decomposition*," in Proceeding from the ANS Mathematics and Computation 2009 Meeting, Saratoga Springs, (2009).
6. M. O'Brien, J. Taylor and R. Procassini, "*Dynamic Load Balancing of Parallel Monte Carlo Transport Calculations*," in ANS Monte Carlo 2005: The Monte Carlo Method: Versatility Unbounded In A Dynamic Computing World, Chattanooga, TN, (2005).
7. R. Procassini, M. O'Brien and J. Taylor, "*Load Balancing of Parallel Monte Carlo Transport Calculations*," in Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Application, Palais des Papes, Avignon, France, (2005).
8. M. J. O'Brien, P. S. Brantley and K. I. Joy, "*Scalable Load Balancing For Massively Parallel Distributed Monte Carlo Particle Transport*," in International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), Sun Valley, Idaho, 2013.
9. M. J. O'Brien, S. A. Dawson, P. S. Brantley and K. I. Joy, "*Scalable Algorithms for Monte Carlo Particle Transport*," in LLNL-CONF-643319, Livermore, 2012.