A Dynamic Replication Approach for Monte Carlo Photon Transport on Heterogeneous Architectures

Ryan Bleile[1,2], Patrick Brantley[1], Matthew O'Brien[1], and Hank Childs[2]

[1] Lawrence Livermore National Laboratory
Livermore, California, USA 94550
[2] University of Oregon
Eugene, Oregon, USA 97403

Abstract. This paper considers Monte Carlo photon transport applications on heterogenous compute architectures with both CPUs and GPUs. Previous work on this problem has considered only meshes that can fully fit within the memory of a GPU, which is a significant limitation: many important problems require meshes that exceed memory size. We address this gap by introducing a new dynamic replication algorithm that adapts assignments based on the computational ability of a resource. We then demonstrate our algorithm's efficacy on a variety of workloads, and find that incorporating the CPUs provides speedups of up to 20% over the GPUs alone. Further, these speedups are well beyond the FLOPS contribution from the CPUs, which provide further justification for continuing to include CPUs even when powerful GPUs are available. In all, the contribution of this work is an algorithm that can be applied in real-world settings to make more efficient use of heterogeneous architectures.

Keywords: Monte Carlo · photon transport · load balance · GPU

1 Introduction

Monte Carlo transport is an important computational technique for nuclear science applications, including applications in physics, nuclear reactors, medical diagnostics, and more. The technique involves simulating phenomena by calculating how a sample of particles moves through and interacts with a background medium. A key consideration for the approach is how many particles to employ, as adding more particles provides more accuracy but comes at the expense of increased computational requirements. In many cases, achieving sufficient accuracy requires a large number of particles, which in turn creates significant computational requirements.

Supercomputers are often used to simulate computationally-expensive Monte Carlo photon transport problems. These machines can calculate many more operations per second than a normal computer, which in turn enables many workloads to be simulated on feasible time scales. That said, these machines

create significant challenges, as they require both parallel coordination within a compute node and across compute nodes. Further, the architectures of their compute nodes are increasingly heterogeneous, often containing both multi-core CPUs and one or more GPUs.

A typical strategy for a heterogenous supercomputer is to use the CPUs only for management and communication with other compute nodes and to use the GPUs to transport particles. This approach usually pairs each GPU with one CPU core to drive the application, and leaves the rest of the CPU cores idle. Based on the relative FLOPS, utilizing the CPU only for management tasks would appear to be an acceptable strategy. Using Lawrence Livermore's RZAnsel [1] supercomputer as an example, the GPUs make up 1,512 TFLOPS, while the CPUs make up 58 TFLOPS, for a total system GPU+CPU count of 1,570 TFLOPS. This means that GPUs and CPUs make up 96.3% and 3.7% of the total FLOPS, respectively — the programmer effort to engage the CPU may not be viewed as worthwhile. However, CPUs have other benefits, including increased memory size and reduced latency to access memory. Further, many operations for Monte Carlo photon transport are not FLOP-bound. In all, engaging CPUs to carry out computation has the potential to add benefits beyond their FLOPS contributions (e.g., beyond 3.7%).

O'Brien et al. [8] were the first to demonstrate benefits from incorporating CPUs alongside GPUs to carry out Monte Carlo photon transport. That said, their algorithm was limited in utility, because it could only be applied to meshes that could fit entirely within GPU memory. This limitation is crucial in the context of supercomputers, since typical simulations at large scale use computational meshes that exceed GPU memory. Such meshes are decomposed into domains (or blocks), with each block small enough to fit within memory and each compute node working on one (or more) blocks. This domain decomposition complicates execution, as each compute node can only transport particles where it has valid data. In this paper, we expand upon the work by O'Brien et al. to deal with domain-decomposed meshes. We accomplish this by introducing two new algorithms: one for load balancing and one for building communication graphs. We also analyze the effects of domain decomposition on the performance of hybrid heterogeneous approaches. In all, the contribution of this work is a practical algorithm that translates the potential demonstrated by O'Brien et al. into a real world setting.

2 Background

Monte Carlo photon transport problems divide their spatial domains amongst its compute resources (i.e., MPI Ranks) in a non-traditional manner. In many physics simulations, there is a one-to-one mapping between compute resources and spatial domains — a physics simulation with N compute resources has N spatial domains, and each compute resource has its own unique spatial domain. With Monte Carlo photon transport problems, the full mesh is often too large to fit into one compute resource's memory, but not so large that it must be

fully partitioned across the total memory of all the compute nodes. Saying it another way, there are often fewer spatial domains than compute resources, and so multiple computational resources can operate on the same domain at the same time. Consider a simple example with two spatial domains (D_0 and D_1) and four compute resources (P_0, P_1, P_2, and P_3). One possible assignment is for D_0 to be on P_0, P_1, and P_2 and D_1 to be on P_3, another possible assignment is for D_0 to be on P_0 and P_1 and D_1 to be on P_2 and P_3, and so on. Overall, domain assignment is an additional component for optimizing performance.

In the Monte Carlo community, the mapping of spatial domains to compute resources is referred to as "replication," as the mapping will replicate some domains across the resources. There are two main strategies for replication: static and dynamic. Static replication makes assignments when the program first begins and uses those assignments throughout execution. Dynamic replication changes assignments as the algorithm executes, in order to maintain load balancing. Both replication strategies aim to improve efficiency — they operate by replicating the spatial domains that have more particles, in order to distribute the workload more evenly across compute resources.

Dynamic replication is part of an overall approach for Monte Carlo transport. Each cycle of a Monte Carlo approach consists of three phases: initialization, tracking, and finalization. When incorporating dynamic replication, the initialization phase executes the dynamic replication algorithm. The tracking phase does a combination of particle transport and communicating particles. Particle transport operates mostly in an embarrassingly parallel fashion, up until particles move from one spatial domain to another (and thus need to be re-assigned to a compute resource that has that spatial domain) and thus MPI communication is required. The finalization phase processes the distributed results of the tracking phase. Importantly, the initialization phase determines the performance of the tracking phase — if the domain assignments from the dynamic replication algorithm create balanced work for each compute resource, then all compute resources should complete the tracking phase at the same time, ensuring parallel efficiency.

Tracking is the computationally dominant portion of the algorithm. During tracking, each particle makes small advancements for short periods of time, and each advancement is referred to as a "segment." The type of activity within a segment can vary, which affects the computational cost and duration of the advancement for a segment. In this paper the three relevant activities are: (1) collisions with the background material, (2) moving between mesh elements, and (3) moving to the end of the time step. Tracking concludes when each particle has advanced for a period equal to the overall cycle duration — if the overall cycle takes ΔT seconds, if a given particle advances via N segments for that cycle, and if each segment i advances for some time t_i seconds, then $\sum_{i=0}^{N-1} t_i = \Delta T$.

3 Related Works

Many works have studied spatial domain decomposition methods for Monte Carlo particle transport. The method was introduced by Alme et al. [2], as they split a problem into a few parts, allowing for replications of spatial domains in order to parallelize the workloads while maintaining processor independence. Their proposed method was adopted by the Mercury simulation code and implemented in a production environment; this team then provided empirical evidence for its efficacy [11]. Spatial domain decomposition methods were further analyzed by Brunner et al. [4, 5], who also contributed improvements for increasing scalability and improving performance overall. One of their important improvements for scalability was to add point-to-point communication, allowing processors in different spatial domains to communicate directly with one another. This was a change from a model where each spatial domain had a single processor which was in charge of all communication for that group of processors.

Work by O'Brien et al. [9] introduced dynamic replication. Their scheme performed regular evaluation of parallel efficiency and then performed load balancing when efficiency dropped below a specified threshold. O'Brien et al. [10] extended this work by adding a communication graph, which defined which processors can perform point-to-point communication during a cycle. Using these new communication graph algorithms, O'Brien was able to successfully scale Mercury on LLNL's Sequoia supercomputer to over one million processors while maintaining good parallel performance. This work showed that keeping the load balance during particle communication within a cycle is important for scaling parallel performance. When particles were communicated to neighbors without considering load balance, a single processor could become bogged down with significantly more work — work which potentially could have been shared. Additional extensions to this work can be seen in other groups as well, such as with Ellis et al. who looked into additional mapping algorithms under specific conditions in the Monte Carlo transport code, Shift [6]. Their work extends the communication graph concept by combining it with Monte Carlo variance reduction techniques to improve the overall efficiency for their use-cases.

While many works have focused on algorithmic improvements, many others have focused on evaluating load imbalance effects. In his PhD thesis, Paul Romano expanded upon the concept of domain decomposition algorithms by providing new analytical understanding [12]. In particular, Romano provided a basis for understanding the importance of load imbalance and being able to determine analytically the benefit of this method. Wagner et al. [13] took a more empirical approach when studying load imbalance of reactor physics problems. They considered the problem of load imbalance stemming from spatial decomposition, and proposed new decomposition methods for handling this issue. Similarly, Horlik et al. [7] explored several spatial domain decomposition methods and analyzed their effect on load imbalance. In summary, each of these groups identified load imbalance as a problem and proposed analysis and solutions that fit their specific needs.

As noted in the introduction, our closest comparator is a separate work from O'Brien et al. [8]. This work considered the problem of balancing particles in a given spatial domain among processors of varying speeds, but it did not consider domain decomposed meshes. As domain replication strategy is an important aspect to achieve performant algorithms, developing an algorithm that supports both heterogenous computing and domain decomposition is non-trivial and requires fresh investigation. This gap is the focus of our work.

4 Our Method

This section describes our novel dynamic replication algorithm for Monte Carlo transport. Our algorithm is optimized for heterogenous architectures — it assumes that individual computational resources will have different levels of compute power, and makes assignments based on that knowledge. Our algorithm consists of three steps:

1. **Assignment** (Section 4.1): identify how many times to replicate each spatial domain, and then assign those domains to compute resources.
2. **Distribution** (Section 4.2): partition the particles across compute resources.
3. **Mapping** (Section 4.3): build a communication graph between compute resources in order to communicate particles that have exited their current spatial domain during tracking.

4.1 Step 1: Assignment

This step produces an assignment of compute resources to spatial domains, with the goal of making an assignment that minimizes execution time. In particular, the number of particles per spatial domain varies, and so the goal is to replicate the domains with the most particles in order to assign a commensurate level of compute to each domain. The algorithm works by considering work and compute as proportions — if a domain has 10% of the particles, then that domain should be replicated so that it gets 10% of the compute resources. Further, if the assignments are effective, then all compute resources should complete at the same time during the tracking phase.

To make assignments, our algorithm needs to understand (1) how much work needs to be performed and (2) how capable the compute resources are. In both cases, we use results from the previous cycle, which we find to be a good representation for what work to expect in the next cycle. Explicitly, the total work for each domain is the number of segments to execute. We consider the per-domain work from the previous cycle as our estimate for the upcoming cycle. For compute rate, we consider how many segments per second each type of resource achieved. That is, we measure the average number of segments per second over all of the CPUs and the same for GPUs. Using past performance automatically accounts for variation in translating FLOPS to segments across hardware; where the FLOP ratio between a GPU and CPU may be 100:1, the ratio in average number of segments per second may be much lower, like 20:1.

Our algorithm depends on considering both work and compute in proportion to the whole, and we define three terms for ease of reference. Let PW_i be the proportion of work within spatial domain i. For example, if domain i has 10% of the total estimated work, then $PW_i = 0.1$. Further, let $PC\text{-}GPU$ and $PC\text{-}CPU$ be the proportion of total compute for a GPU and a CPU, respectively. For example, if a GPU can do 100 million segments per second, if a CPU can do 5 million segments per second, and if there are 4 GPUs and 20 CPUs, then the total capability is 500 million segments per second, and $PC\text{-}GPU = 0.2$ and $PC\text{-}CPU = 0.01$.

At the beginning of program execution, we assign each domain one GPU and one CPU. This ensures that every domain has "surge" capability in case the work assignment estimates are incorrect (which can happen when particles migrate from one domain to another at a high rate). Such surge capability prevents the worst case scenario — one compute resource takes a long time to complete its work, and the others sit idle. Further, one of these compute resources (either the CPU or GPU) can act as a "foreman" for its spatial domain. These foremen are bound to a spatial domain throughout program execution. When a compute resource is assigned a new spatial domain, it can get that domain from the appropriate foreman. The remaining compute resources can then be assigned to work on spatial domains dynamically.

Our assignment algorithm works in two phases. The first phase decides how many compute resources should be assigned to each spatial domain, and what type they should be. The second phase uses this information to make actual assignments to specific compute resources, being careful to minimize communication by keeping the same spatial domains on the same compute resources when possible.

The first phase employs a greedy algorithm, and is described in pseudocode below labeled "MakeGreedyAssignments." It begins by setting up an array variable that tracks how much work is remaining for each spatial domain ("RemainingWork") using the predicted work (PW_i) and taking into account the pre-allocated resources (one CPU and one GPU for each of the M spatial domains). The final step is to assign the remaining compute resources to spatial domains. $NGPU$ is the number of GPUs, it begins by assigning the $NGPU - M$ available GPUs to spatial domains, one at a time. Each time, the algorithm first finds the spatial domain d with most remaining work, i.e., its evaluations takes into account that resources have been assigned previously. After the GPUs, it then makes assignments for each of the $NCPU - M$ available CPUs in a similar manner.

```
function MakeGreedyAssignments(M, NGPU, NCPU, PW, PCGPU, PC-
CPU)
    for i in range(M) do
        WorkRemaining[i] = PW[i]
    end for
    for i in range(M) do                    ▷ Account for preallocated resources
        WorkRemaining[i] -= (PCGPU+PCCPU)
```

```
    end for
    NGPU -= M
    NCPU -= M
    for i in range(NGPU) do          ▷ Replicate remaining resources greedily
        d = FindDomainWithMostWork(WorkRemaining)
        WorkRemaining[d] -= PCGPU
        AssignGPUToSpatialDomain(d)
    end for
    for i in range(NCPU) do
        d = FindDomainWithMostWork(WorkRemaining)
        WorkRemaining[d] -= PCCPU
        AssignCPUToSpatialDomain(d)
    end for
end function
```

All replication schemes nearly always have some load imbalance. Consider a problem with two spatial domains with equal amounts of particles ($PW_0 = PW_1 = 0.5$) and three GPU compute resources, C_0, C_1, C_2 where $PC\text{-}GPU = 0.333$. Then C_0 and C_1 will be foremen, and the only question is whether to replicate domain 0 or 1 on C_2. Whatever the outcome, one domain will have a $WorkRemaining$ value of 0.167. In this example, it would be up to the foreman to carry out this extra work and it would be likely that the extra compute resources would be idle as it does so. Fortunately, these effects get smaller as concurrencies get larger. Also, the heterogeneous nature of compute helps on this front, as there are more resources (the CPUs) that are smaller (i.e., smaller values of $PC\text{-}CPU$) leading $WorkRemaining$ values being closer to 0 on the whole.

The second phase assigns specific compute resources. Every time a compute resource is assigned a new domain, it must retrieve this domain from its corresponding foreman, incurring a communication cost. So the goal of this phase is to repeat assignments between compute resources and domains. For example, if the output of the first phase indicates that domain d should have 3 GPUs, then the second phase checks to see if there are 3 GPUs that had d in the previous cycle. If so, then those GPUs should be assigned to d again for the current cycle, as this prevents unnecessary communication. Of course, as the number of compute resources applied to a domain increases, new compute resources must be located and communication costs are inevitable.

4.2 Step 2: Distribution

This step partitions the particles across compute resources. This partitioning must honor the spatial domain assignments, i.e., if particle P lies within spatial domain D, then the particle can only be to assigned to compute resources that were assigned D. In our approach, we perform this partitioning relative to performance — GPU compute resources get more particles and CPU compute resources get less, and the proportion between them corresponds to $\frac{PC\text{-}GPU}{PC\text{-}CPU}$. The remainder of implementation details follow trivially from previous work [8].

4.3 Step 3: Mapping

Mapping refers to establishing a communication graph between compute resources. This mapping is needed when particles exit their spatial domain. When this happens, they need to be sent from their current compute resource to another compute resource that is operating on their new spatial domain.

In a domain replication environment, a poor communication graph can affect overall performance. For example, assume that domain d is replicated by K compute resources — C_0, C_1, ... $C_{(K-1)}$. One possible communication graph could instruct all other compute resources to send their particle entering d to C_0. This is bad: C_0 would spend more time doing communication than the other C_i resources and it also will end up with more particles to transport. Instead, a better mapping would lead to an even spread of particles between the C_i's.

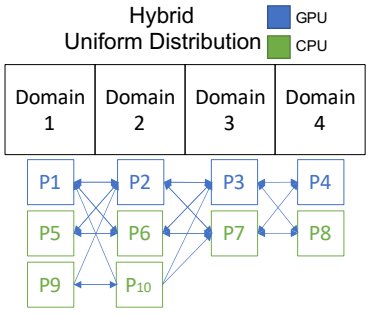


Fig. 1. Result of our Map step with 4 spatial domains, 4 GPU compute resources, and 6 CPU compute resources. The square boxes show which domains neighbor (1-2, 2-3, 3-4).

For a given compute resource, our Map algorithm makes connections to all neighboring domains. It uses a round robin algorithm to prevent load imbalance, specifically:

$$index_A \ mod \ size_B = index_B \ mod \ size_A$$

where A is a list of resources from one domain and B is a list of resources from a second domain (see Figure 1). Our Map algorithm makes two connections for each neighboring domain d — one to a CPU compute resource that contains d and one to a GPU compute resource that contains d. Each connection also has a weighting which dictates the proportion of particles communicated. For our experiments, we set the weights to be proportional to their compute abilities ($PC\text{-}GPU$ and $PC\text{-}CPU$), i.e., a GPU resource would be sent many more particles than a CPU resource. That said, exploring different weights would be interesting future work, in particular weights where CPUs get more particles.

5 Experiment Overview

This section provides an overview of our experiments, and is organized into three subsections. Subsection 5.1 describes the hardware and software used for our experiments. Subsection 5.2 describes the factors we vary to form our set of experiments. Finally, Subsection 5.3 describes the measurements we use to evaluate our results.

5.1 Hardware and Software

Our experiments were run on LLNL's RZAnsel supercomputer. This platform has two Power 9 CPUs (22 cores per CPU, of which 20 are usable), 4 Nvidia Volta GPUs (84 SMs per GPU), and NVLink-2 Connections between the sockets on each node. In addition, there are a total of 256 GB of CPU memory and 64 GB of GPU memory per node [1]. For software, we used Imp [3], a Monte Carlo code that solves time-dependent thermal x-ray photon transport problems.

5.2 Experimental Factors

Our experiments vary two factors: workload (11 options) and Hardware configuration (3 options). We ran the cross product of experiments, meaning 33 experiments overall.
Workloads: our 11 unique workloads consisted of three distinct problems ("Crooked Pipe," "Holhraum," and "Gold Block"), with one of those problems ("Gold Block") having nine different variations. Details for each of the three distinct problems are as follows:

- **Crooked Pipe:** a problem that simulates transport through an optically thin pipe with a U-shaped kink surrounded by an optically thick material. The Crooked Pipe problem is load imbalanced since particles are sourced into the leading edge of the pipe, causing spatial domains that contain this region to have a much higher amount of work per cycle than the others. This is a common test problem in the Monte Carlo photon transport community as well as an excellent driver for testing load balancing methods.
- **Hohlraum:** a problem that simulates the effects of Lawrence Livermore's NIF laser on a gold hohlraum. Particles in this problem start in an incredibly hot gold wall and then propagate throughout the mostly hollow interior, colliding with a central obstruction as well as the surrounding gasses. This problem starts out very load imbalanced with most work in the hot region.
- **Gold Block:** a homogenous test problem that simulates a heated chunk of gold. This problem is a solid cylinder of gold with reflecting boundary conditions. Since this problem is a homogeneous material with reflecting boundary conditions, we can modify the length scale of the problem in order to change the ratio of the number of collision segments with the number of total segments by changing the number of mesh element crossing segments and leaving all else fixed. We use this length scaling to create a total of 9 configurations, with an unscaled version at the center we refer to as the Base Gold Block. Specifically, we took our Base Gold Block problem and halved the length scale 4 times consecutively, and similarly doubled the length scale 4 times consecutively to create these configurations. The goal with this scaling is to understand performance with respect to the percent of time performing collisions segments versus other segment types.

Hardware configurations: we ran each of the workloads with:

– **Hybrid:** our algorithm, scheduled with both GPUs and CPUs.
– **CPU-Only:** scheduled using only CPU resources
– **GPU-Only:** scheduled using only GPU resources

For the **CPU-Only** and **GPU-Only** tests, we were able to perform experiments using our algorithm, since our algorithm simplifies to be the same as predecessor work when the resources are homogeneous. Further, all experiments were run on 4 nodes, meaning we used: for CPU-Only 160 CPU resources, for GPU-Only 16 GPU resources, and for Hybrid 144 CPU resources + 16 GPU resources. Each of these experiments were run with a total of 80 million particles.

5.3 Measurements

To analyze our results, we considered three types of measurements:

– **Throughput** defines the number of segments, on average, that a processor will be able to process in one second. This metric is used to compare application performance in a consistent manner, regardless of hardware or software configuration.
– **Segment Counters** divide the segments into the three different activity types considered in this paper (see Section 2). Specifically, these counters count the total number of times each type of segment has occurred across all segments in the simulation. Segment counters are useful for understanding how performance varies with respect to different segment types.
– **Efficiency** determines the success of a load balance algorithm. For each domain i, we calculate the ratio of the compute resource applied to that domain (sums of $PC\text{-}GPU$ and $PC\text{-}CPU$ for the assigned C_i's) and work for that domain (PW_i). For example, a given domain may have 8% of the compute resources and 10% of the total work, for a ratio of 0.8 or an efficiency of 80%. Our efficiency metric is the minimum of these ratios over all domains, meaning 1.0, 100%, is a perfect score (compute resources applied perfectly in proportion to work for all domains) and less than 1.0 indicates the inefficiency — a score of 0.5 indicates that one domain has been given half the resources it needs, i.e. it has an efficiency of 50%.

6 Results

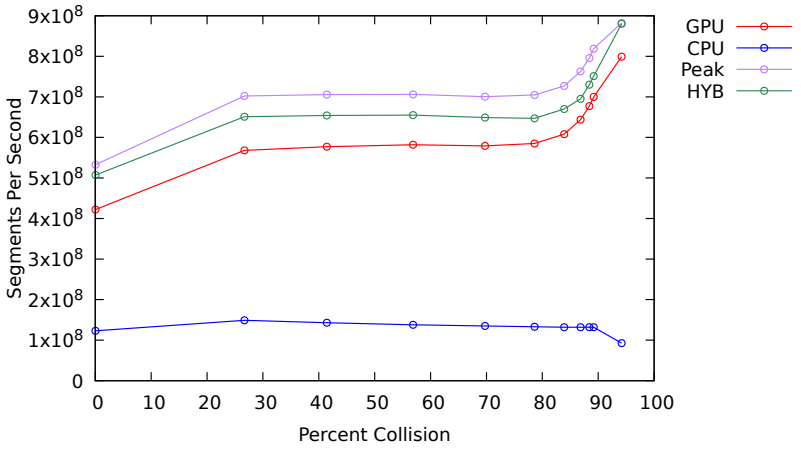
Our results are organized into three parts:

– Section 6.1 evaluates the performance of our overall heterogenous algorithm.
– Section 6.2 evaluates the efficacy of our load balancing algorithm.
– Section 6.3 evaluates the importance of our algorithm's surge capability.

6.1 Algorithm Performance

Figure 2 shows the performance results for our 33 experiments. This figure contains a line for "peak" performance. This does not represent actual experiments,

Fig. 2. Plot showing throughput (segments per second) as a function of what proportion of the segments were of the type "collision," as given by the segment counters. It plots four lines, one for each of our three hardware configurations, and one for a theoretical "peak" configuration (described in Section 6.1). Each of the dots come from our workloads — the left-most (∼0% collisions) come from the Crooked Pipe problem, the right-most (∼100% collisions) come from the Hohlraum problem, and the remainder come from variations of the Gold Block problem.



but rather a theoretical analysis of the potential peak speedup from using both CPUs and GPUs. This line was calculated by taking the GPU performance and adding 90% of the CPU performance (since some CPUs are needed to manage GPUs in a heterogeneous environment, specifically 36 of the 40 CPU cores were used for computation, while the remaining 4 managed GPUs). This peak line should be viewed as a "guaranteed-not-to-exceed" comparator.

One important finding is on the potential of heterogenous computing for this problem. While the CPUs have only 3% of the FLOPs of the GPUs, their performance (i.e., throughput) is much better than 3%. CPUs have 26.2% of the throughput for the Crooked Pipe problem, 20.4% for the Base Gold Block problem, and 10.4% for the Hohlraum problem. In all, this provides important evidence that including CPUs can be much more beneficial than a basic FLOP analysis. Of course, this potential can only be leveraged with an effective algorithm.

With respect to actual achieved performance, our heterogenous algorithm ("Hybrid") performed quite well. It was 20.4% faster than GPU-only for the Crooked Pipe problem, and approximately 10% faster for the other problems. Relative to the peak line, our algorithm achieved 95.1% for Crooked Pipe, 91.8% for Base Gold Block, and 99.6% for Hohlraum. The performance is greatest where the amount of collision segments is dominant, which is also where there is a larger amount of compute used by the resources. In this region, the CPUs are less valuable, but still more valuable than the hardware specification predicts. On the other end of the spectrum, where there are less collisions and more mesh element crossing segments, the compute is lower, and the GPUs are less performant. This enables the CPUs to provide an even greater benefit overall.

6.2 Load Balance Efficiency

Table 1. This table shows the efficiency for our three workloads over a full program execution. Minimum efficiency represents the worst assignment over all compute resources and cycles: for one cycle of the Crooked Pipe problem, there was a compute resources which had about 20% too much work to finish on time with the other compute resources. Maximum efficiency speaks to the best cycle: for one cycle of the Crooked Pipe problem, the most underpowered compute resource had only 0.1% too much work. Average efficiency speaks to the behavior across cycles: (1) for each cycle, identify the most underpowered compute resource and calculate how much extra work it has, and (2) take the average over all cycles of the extra work amounts. For the Crooked Pipe problem, the average efficiency is 99.55%, meaning that the average slowdown for completing a cycle due to load balancing was <0.5%.

Problem	Minimum Efficiency	Maximum Efficiency	Average Efficiency
Crooked Pipe	81.76%	99.92%	99.55%
Base Gold Block	81.76%	99.9%	99.90%
Hohlraum	81.76%	86.16%	85.80%

Table 1 plots efficiency results for our three workloads. On the whole, the minimum efficiency values for these workloads are low. That said, these conditions occur in the first few cycles, as these cycles do not have a history of performance to base their load balancing decisions on. For Crooked Pipe and Base Gold Block, the average efficiencies indicate that good load balance is achieved quickly and maintained throughout the run. The Hohlraum problem had worse efficiency. This was because one domain was a "hot spot" — it had much more work than the other domains. This topic is explored further in the following subsection.

6.3 Surge Capability

The Hohlraum workload demonstrates the value in our "surge capability" (ensuring that one GPU and CPU are assigned to each domain). This workload had 4 domains, and domain 1 had the majority of particles, to the point meriting assignment of every non-foreman compute resource. That said, during a compute cycle, domain 1's particles stream out rapidly into neighboring domains. Our surge capability ensured extra compute resources were allocated, and this made a 3X performance improvement for this case. Figure 3 has more details on this comparison, with Gantt charts that show behavior within a cycle. Finally, the "surge" allocation had no impact on the other two problems since their work was more balanced, and they would have received those resources anyway.

7 Conclusion

In this paper, we introduce a novel load balancing algorithm which can efficiently partition heterogeneous compute resources across domains. We demonstrate re-

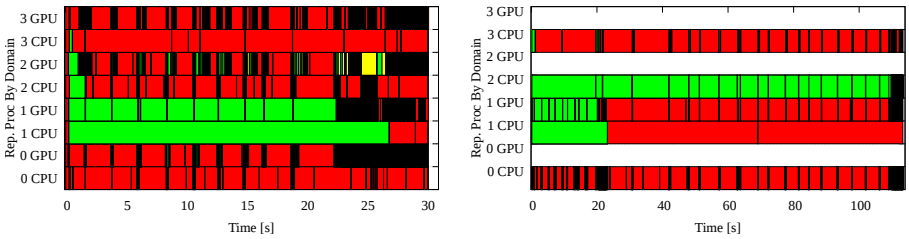


Fig. 3. Gantt charts for a single cycle of the Hohlraum workload. The left Gantt chart corresponds to our algorithm, and completes in 30s. The right Gantt chart is a variant of our algorithm where there is no minimum compute allocation (i.e., the "surge capability" is disabled). This variant took 113s, 3.8X slower. The Gantt plots show an evolution over time per compute resource, with red representing "idle" time, yellow representing communication time ("MPI Send/Recv"), and green representing time spent tracking particles. The blank spaces in the right chart occur because there is no compute resource assigned to that domain, for example no GPU resource for domain 3. Finally, these Gantt charts show only the first CPU and first GPU for a domain, and the other compute resources are not plotted. In particular, the remaining compute resources in the left Gantt chart (our algorithm) are doing tracking (green) at a high rate, consistent with the overall efficiency of 85% — some of the worst performers for this workload (domains 0, 2, and 3) are being plotted.

sults using this algorithm, in a production Monte Carlo photon transport code, running a variety of workloads. This work was motivated by the performance difference seen in practice between Monte Carlo transport codes running on CPUs and GPUs when compared with the ratio of the available FLOPs. Our algorithm demonstrated up to a 20% performance benefit, which is much greater than the 3.7% predicted by solely looking at the ratio of FLOPs. Additionally, our algorithm achieves 85% to 99% load balancing efficiency on the problems demonstrated.

In terms of future work, we wish to study more workloads, such as neutron transport, and to run larger problems using more compute resources. We also plan to look more at the surge capability, and whether better predictions can be made about when it is needed. In particular, the assignment of at least one GPU to every domain can be wasteful in some configurations, possibly limiting performance by as much as 15% in extreme cases. Further, while domain replication algorithms assume that each MPI rank has a single domain for a given cycle, more adaptive approaches, including switching a domain mid-cycle and splitting a domain into pieces could have performance benefits. Another improvement would be adapting assignments based on current loads, in case the target resource already is overloaded. Finally, we plan to extend our algorithm to work with even more heterogeneous architectures in the future, including multiple types of accelerators on a node and even nodes with different types of compute power. This work will happen as such machines come online, as most supercomputers are using strictly CPUs and GPUs at this time.

References

1. Livermore computing center high performance computing: Rzansel, https://hpc.llnl.gov/hardware/platforms/rzansel, accessed: 2020-12-09
2. Alme, H.J., Rodrigue, G.H., Zimmerman, G.B.: Domain decomposition models for parallel monte carlo transport. The Journal of Supercomputing **18**(1), 5–23 (2001)
3. Brantley, P., et al.: A new implicit monte carlo thermal photon transport capability developed using shared monte carlo infrastructure. In: The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2019). pp. 25–29. Portland, Oregon (August, 2019)
4. Brunner, T.A., Brantley, P.S.: An efficient, robust, domain-decomposition algorithm for particle monte carlo. Journal of Computational Physics **228**(10), 3882–3890 (2009)
5. Brunner, T.A., et al.: Comparison of four parallel algorithms for domain decomposed implicit monte carlo. Journal of Computational Physics **212**(2), 527–539 (2006)
6. Ellis, J.A., et al.: Optimization of processor allocation for domain decomposed monte carlo calculations. Parallel Computing **87**, 77–86 (2019)
7. Horelik, N., Siegel, A., Forget, B., Smith, K.: Monte carlo domain decomposition for robust nuclear reactor analysis. Parallel Computing **40**(10), 646–660 (2014)
8. O'Brien, M., et al.: Hybrid cpu-gpu load balancing for monte carlo particle transport. In: Proceedings of the 26th International Conference on Transport Theory (ICTT-26), Sorbonne Univeristy, Paris, France (2019)
9. O'Brien, M.J., Brantley, P.S., Joy, K.I.: Scalable load balancing for massively parallel distributed monte carlo particle transport. In: Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), Sun Valley, Idaho. vol. 45, pp. 647–658 (2013)
10. O'Brien, M.: Dynamic load balancing of parallel monte carlo transport calculations via spatial redecomposition. In: Proceedings of the Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications. pp. 16–19 (2007)
11. Procassini, R., O'Brien, M., Taylor, J.: Load balancing of parallel monte carlo transport calculations. In: Proceedings of the 2005 ANS Topical Meeting in Mathematics and Computation. Avignon, France (September 12-15, 2005)
12. Romano, P.K.: Parallel algorithms for Monte Carlo particle transport simulation on exascale computing architectures. Ph.D. thesis, Massachusetts Institute of Technology (2013)
13. Wagner, J.C., et al.: Hybrid and parallel domain-decomposition methods development to enable monte carlo for reactor analyses. Progress in nuclear science and technology **2**(1), 815–820 (2011)