

COMPARISON OF FOUR PARALLEL ALGORITHMS FOR DOMAIN DECOMPOSED IMPLICIT MONTE CARLO

Thomas A. Brunner*

Sandia National Laboratories
PO Box 5800, Albuquerque, NM 87185-1186

Todd J. Urbatsch, and Thomas M. Evans[†]

Los Alamos National Laboratory
PO Box 1663, Los Alamos, NM 87545

Nicholas A. Gentile[‡]

Lawrence Livermore National Laboratory
7000 East Ave., Livermore, CA 94550

ABSTRACT

We consider four asynchronous parallel algorithms for Implicit Monte Carlo (IMC) thermal radiation transport on spatially decomposed meshes. Two of the algorithms are from the production codes KULL from Lawrence Livermore National Laboratory and Milagro from Los Alamos National Laboratory. Improved versions of each of the existing algorithms are also presented. All algorithms were analyzed in an implementation of the KULL IMC package in ALEGRA, a Sandia National Laboratory high energy density physics code. The improved Milagro algorithm performed the best by scaling almost linearly out to 244 processors for well load balanced problems.

KEYWORDS: Monte Carlo methods, parallel computation, domain decomposition

1. INTRODUCTION

Monte Carlo simulations are embarrassingly parallel if you replicate the spatial domain on all processors of a distributed memory computer. However, this is not an option for many three-dimensional, coupled-physics problems because of computer memory constraints. In these cases, the spatial domain must be partitioned among the processors. As particles move through the system, they may hit a processor boundary and need to be moved to another processor.

Four different algorithms are outlined, and the performance of each is measured on two test

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

[†]Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

[‡]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

problems. These algorithms are implemented in the KULL IMC package[1] running inside of ALEGRA[2]. This package implements the Implicit Monte Carlo (IMC) scheme for thermal radiation transport of Fleck and Cummings[3].

The IMC algorithm solves the time-dependent transport equation for photons coupled with matter. The problem is meshed, with each zone in the mesh having an individual temperature. Particles representing photons are created by thermal emission in the zones or by other external photon sources and are tracked through the mesh. These particles deposit energy in each zone they pass through, in an amount calculated from the absorption opacity of the material in the zone. This deposition decreases the energy of the particle. Particles also execute scatters, if the material in the zones has a non-zero scattering opacity. The location of the created particles, and the scatters, are simulated using random numbers. Thus, the results of the algorithm have statistical noise. Particles are terminated when they reach problem boundaries or their energy reaches a small fraction, typically chosen to be 1%, of their initial energy. Particles which survive until the end of the time step are retained and continue in the next time step. At the end of the time step, the change in material energy is calculated by subtracting the energy radiated by thermal emission and adding energy from absorption, and the new material temperatures are calculated. This process is repeated on subsequent time steps until the end time of the simulation is reached. The photon population is held constant in each time step by varying the energy assigned to each photon. Energies for new photons in each time step are chosen so that the sum of all of the photon energies equals the source energy plus the energy in photons from previous time step. The only difference between serial and parallel implementations of the method is that particles must be transferred across domain boundaries as they traverse the mesh. How to do this efficiently is the subject of this paper.

ALEGRA supports fully unstructured two and three dimensional meshes with arbitrary spatial decomposition. Figure 1 shows a sample mesh used by ALEGRA. Figure 1(a) is a sample physical geometry with a cylinder of hot material radiating in the center of a rectangular box. The spatial decomposition of the mesh is shown in Figure 1(b), where the mesh has been broken into three regions, and each region is assigned to a different processor.

When a Monte Carlo simulation is run in a domain decomposed fashion, the result will not necessarily be the same as when it is run on a single domain. This occurs for two reasons: first, the random numbers used in the simulation will be different; second, the order in which events occur will change. For example, the deposition of energy in a zone by one particle can occur before the deposition by a second particle when the problem is run on one domain, but after when it is run on two domains. This makes the sum of energy deposited different, since finite precision floating-point addition is not commutative, for example $(x + y) + z = x + (y + z)$ may not be true. We eliminate the first problem by giving each particle a unique random number stream, so that it uses the same random number stream on every domain. To solve the second, we tally energy deposited into an integer data type, for which addition is commutative. This integer result is then scaled to a floating point number at the end of the time step. As a result of this procedure, we obtain identical results for our simulations on any number of domains. Details of this procedure can be found in [1, 4].

The algorithms presented here only address the scalability for domain decomposed meshes that

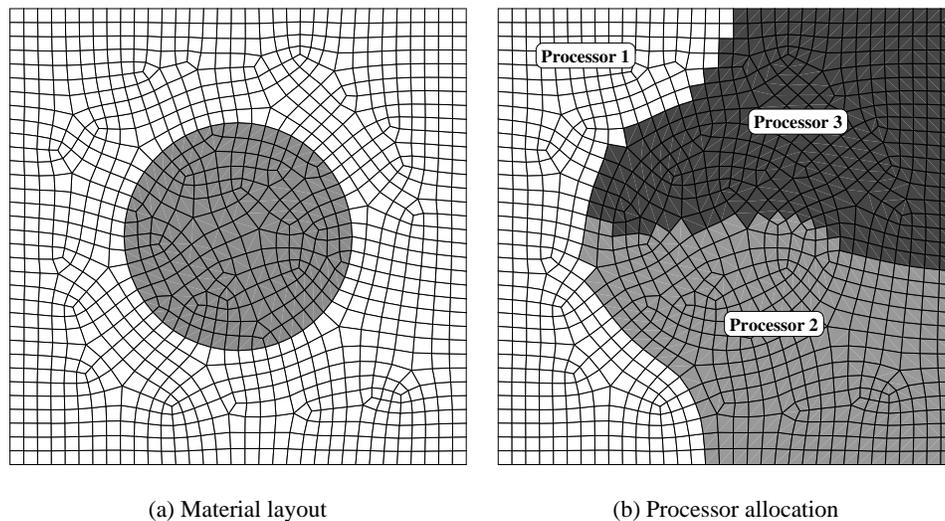


Figure 1. A sample two dimensional geometry specification used by ALEGRA. Here, a cylinder of hot radiating material is in the center of a rectangular box. The domain is divided into three regions, each assigned to one processor.

have reasonable particle load balancing. If one processor has significantly more particles than the others, all of the algorithms presented here will scale poorly. Two test problems will be used to illustrate the scaling of the algorithms in the perfectly load balanced and mildly unbalanced cases.

2. ALGORITHMS

In domain decomposed Monte Carlo, two sets of data must be communicated between the processors. The nearest neighbors must exchange particles that cross processor boundaries through shared faces. For example, Processor 7's nearest neighbors in Figure 2 are Processor 3, Processor 6, Processor 8, and Processor 11. A global communication operation must also be performed so that all the processors know when all the other processors are finished moving all the particles. The four algorithms for a time step in the IMC package vary in how they perform each of these two tasks. Specific MPI calls in the algorithms are shown.

2.1. Algorithm I: KULL

Algorithm I shows the original communication method used by the KULL IMC package[1]. The number of particles that need to be exchanged is not known, so this information must be sent before allocating memory for the receive buffer. As for all algorithms, the list of neighbor processors must only be gathered once per simulation; the mesh or its decomposition does not change between time steps. It is critical to have a sorted list of the neighbors' processor identification numbers, otherwise it is possible to get locked cycles of processors, each waiting on another to exchange particles. For the three processor decomposition shown in Figure 1(b), each

```

get sorted list of neighbor processor ID numbers
while any active particles on any processor (MPI_Allreduce)
    for each local particle
        move particle to a termination event
        if particle hit processor boundary
            buffer particle
    for each neighbor processor in list
        if my id is less than the other's id
            send buffer size to neighbor (MPI_Send)
            send particles to neighbor (MPI_Send)
            receive buffer size from neighbor (MPI_Recv)
            allocate memory for incoming message
            receive particles from neighbor (MPI_Recv)
        else
            receive buffer size from neighbor (MPI_Recv)
            allocate memory for incoming message
            receive particles from neighbor (MPI_Recv)
            send buffer size to neighbor (MPI_Send)
            send particles to neighbor (MPI_Send)

```

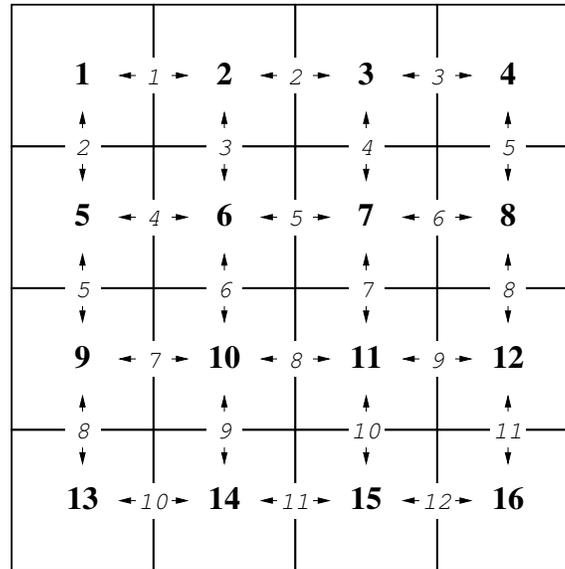
Algorithm I. The KULL Algorithm

processor must exchange particles with the other two. The following scenario could occur with unsorted lists of neighbors: Processor 1 expects to exchange particles with Processor 2 first, and Processor 2 expects to exchange particles with Processor 3 first, and finally Processor 3 expects to communicate with Processor 1 first. In this case, each of the processors will be waiting indefinitely for the others to begin communication. If each processor orders its communication by processor number, Processor 1 communicates with Processor 2 first, and processor 2 communicates with Processor 1 first, thus eliminating the lock.

Even though each processor needs to talk with only its neighbors, this algorithm can have a serial communication pattern in certain circumstances. For example, if you had a square problem domain cut into sixteen sub-domains, as shown in Figure 2, it takes twelve steps to communicate all the data. In general for square problems like this, it takes $4(\sqrt{p} - 1)$ steps, where p is the number of processors. It should take only four steps since each processor has at most four neighbors. This serialization is due to the fact that each processor talks with each of its neighbors one after another, in a specific order. Even if another neighbor is ready to communicate, a processor will wait for the next one in its list.

2.2. Algorithm II: Improved KULL

The blocking sends and receives in Algorithm I lead to non-scalable behavior, like the serialization in the example above. Algorithm II is an improved version of Algorithm I that uses nonblocking communication and combines the buffer size with each buffer, which eliminates a separate message. The nonblocking communication improves the algorithm in two respects. First, the serialization of Algorithm I is eliminated. Additionally, processors are free to communicate with the neighbors that are also ready to communicate, which helps performance



Key: Processor/Domain Number

← Communication Step →

Figure 2. Communication pattern for the KULL IMC algorithm for a square spatial domain distributed on sixteen processors. It takes twelve communication steps to fully exchange all particles when it should only take four steps.

```

get list of neighbor processors ID numbers
while any active particles on any processor (MPI_Allreduce)
  for each local particle
    move particle to a termination event
    if particle hit processor boundary
      buffer particle
  for each neighbor processor in list
    initiate nonblocking send of particles to neighbor (MPI_Isend)
  while any unreceived particle buffer messages from neighbors
    for each neighbor processor in list
      if incoming message from neighbor (MPI_Iprobe)
        get incoming message size (MPI_Get_count)
        allocate memory for incoming buffer
        receive particles from neighbor (MPI_Recv)
wait until all nonblocking sends of particles have completed (MPI_Waitall)

```

Algorithm II. The improved KULL algorithm.

when particles are not load balanced. If, for example, in Figure 2, Processor 2 had twice the number of particles of either Processor 1 or Processor 5, then Processor 1 and Processor 5 could exchange particles before Processor 2 finish, allowing for the overlap of work and computations. In Algorithm I, Processor 1 and Processor 5 must wait until Processor 2 has finished its computations as well as communicating with Processor 1.

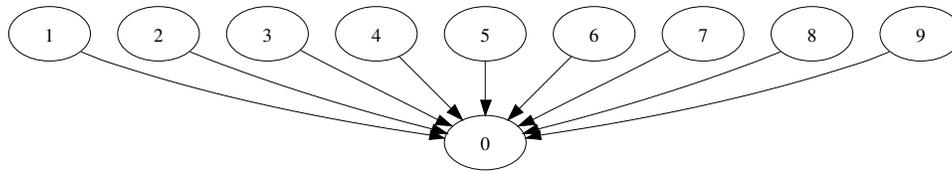


Figure 3. The fat tree communication pattern used in Algorithm III with ten processors. Processor 0 is the master processor, and all others are slaves. All processors perform particle calculations.

2.3. Algorithm III: Milagro

An algorithm based on the Milagro code[5–7] is outlined in Algorithm III. For particle transport on a spatially decomposed domain, each processor continuously loops over mutually exclusive options until every particle in the entire domain finishes.

After simulating each particle, the communicator is checked to see if any particles have arrived from neighboring domains on other processors. This frequent checking was necessary in the original implementation in Milagro on the SGI Octane “Bluemountain” supercomputer (now decommissioned) at Los Alamos National Laboratory. Skipping even a small number of checks would occasionally lock the processors on that machine.

If incoming particles have arrived, they are put into the active particle list in a last-in, first-out manner. During transport, particles that leave the processor’s domain are buffered and eventually sent to the appropriate processors.

When a processor has no more local particles or incoming particles, it deliberately flushes its buffers, sending only the number of bytes needed to transfer the partially full buffer. The number of particles that are being sent is encoded into the beginning of the message buffer and extracted when the buffer is received.

At the beginning of a time step, a master processor, typically Processor 0, gets a total number of global particles that must be simulated over the course of the time step. When there appears to be no more incoming particles, each slave processor sends a message to the master processor indicating how many slave-local particles have been completed since the last such message. Figure 3 shows the communication pattern between the master and slave processors. The master processor occasionally checks for these incoming messages, and adds the incoming slave particle completed counts to its own. Once the master determines that all global particles have been completed, it broadcasts the finished status message to all slave processors. The master processor and all the slave processors perform particle calculations; the master processor has the additional work load of tallying the global number of particles completed. This extra work does impact the scalability, which will be discussed in Section 3.

```

get list of neighbor processor ID numbers
for each neighbor
    post nonblocking receive for maximum buffer size (MPI_Irecv)
if master processor
    post nonblocking receives for particles completed from all slaves (MPI_Irecv)
else slave processor
    post nonblocking receive for finished message from master (MPI_Irecv)
sum to master total global number of particles (MPI_Reduce)
while global finished flag is not set
    if any local particles
        move the last particle in the list to a termination event
        if particle hit processor boundary
            buffer particle
            if buffer full
                send particle buffer to neighbor (MPI_Send)
        else
            increment local particles completed counter
    for each incoming particle buffer (MPI_Test)
        unpack number of incoming particles from buffer
        process particles, adding to end of list
        repost nonblocking receive (MPI_Irecv)
    if no active local particles
        send any partially full particle buffers (MPI_Send)
    if master processor
        for each completed particle count message from slaves (MPI_Test)
            add to local number of particles completed
            repost nonblocking receive for particles completed (MPI_Irecv)
        if all global particles are completed
            set global finished flag
        for each slave
            send global finished message to slave (MPI_Send)
    else slave processor
        send number of local particles completed to master (MPI_Send)
        reset local particles completed to zero
        if global finished message from master (MPI_Test)
            set global finished flag
cancel all outstanding nonblocking receives

```

Algorithm III. The Milagro algorithm

2.4. Algorithm IV: Improved Milagro

Identifying the deficiencies in the Milagro algorithm, we may propose an improved algorithm. The improved version of the Milagro algorithm is shown in Algorithm IV.

While the Milagro algorithm avoids any global synchronizations during the time step, its scalability is limited in three key areas. The Milagro algorithm uses a fat communication tree for the “particles completed” messages, where Processor 0 is the master of all other nodes. The master processor checks for many “particles completed” messages, which causes a load imbalance and poor scaling. The improved Algorithm III uses a binary tree communication

```

get list of neighbor processors
for each neighbor
    post nonblocking receive for maximum buffer size (MPI_Irecv)
calculate parent and children processor ID numbers
for each child
    post nonblocking receive for particles completed (MPI_Irecv)
post nonblocking receive for global finished message from parent (MPI_Irecv)
sum to root processor total number of particles (MPI_Reduce)
while global finished flag is not set
    if any local particles
        move the last particle in the list to a termination event
        if particle hit processor boundary
            buffer particle
            if buffer is full
                send particle buffer to neighbor (MPI_Send)
        else
            increment local particles completed counter
    for every  $N$  particles or if no active local particles
        for each incoming particle buffer (MPI_Testsome)
            unpack number of incoming particles from buffer
            process particles, adding to end of list
            repost nonblocking receive (MPI_Irecv)
        for each completed particle message from children (MPI_Testsome)
            add to local number of particles completed
            repost nonblocking receive for particles completed (MPI_Irecv)
    if no active local particles
        send any partially full particle buffers (MPI_Send)
        send number of particles completed to parent (MPI_Send)
    if root processor
        if all particles completed
            set global finished flag
        else
            reset local particles completed to zero
            if global finished message from parent (MPI_Test)
                set global finished flag
for each child
    send global finished message to child (MPI_Send)
cancel all outstanding nonblocking receives

```

Algorithm IV. The improved Milagro Algorithm

pattern[8], which is optimal for short messages[9], for the asynchronous “particles completed” communications and the finished message broadcast. Each processor has exactly one parent, except for Processor 0, which is the root of the tree. Each processor has up to two children as well. For example, in Figure 4, Processor 4’s parent is Processor 1, and Processor 4’s children are Processors 9 and 10. This communications pattern ensures that each processor has an even and minimal workload for incoming message tests. Each processor checks for the number of particles completed messages from its children, and then forward that count to its parent. The root processor, Processor 0, plays the same roll as the master processor in Algorithm III, namely it knows the global number of particles that need to be processed and keeps the tally of the global

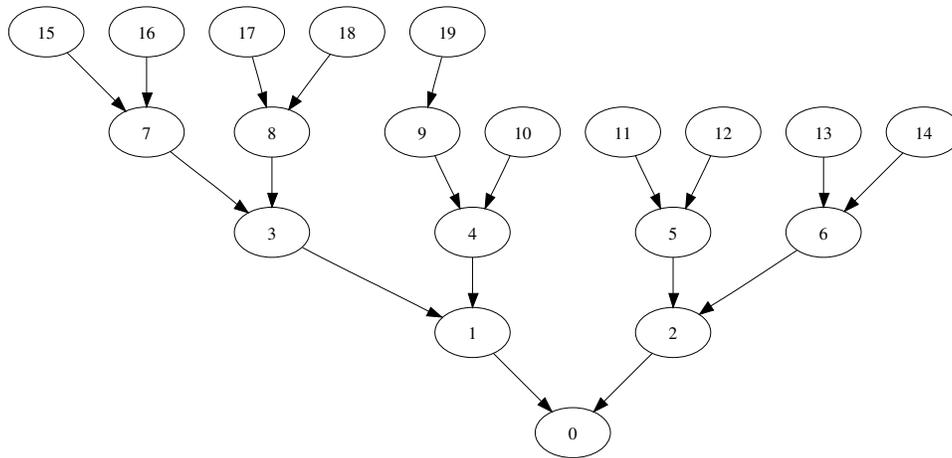


Figure 4. Binary tree communication pattern used in Algorithm IV with twenty processors. Each processor has one parent, except for Processor 0, which is the root of the tree. Each processor also has at most two children. All processors perform particle calculations.

number of particles that have been completed so far. Once the root has determined that all particles on the entire mesh have been finished, the finished flag is broadcast up the tree in the reverse manner. Each processor waits for the finished message from its parent, exits the particle processing loop, and then forwards the finished message onto its children.

The improved algorithm also eliminates the frequent checking for incoming messages. The message queue is only checked after N particles have been simulated, allowing for greater scalability than the (machine) limited $N = 1$ case of the Milagro algorithm.

We also found a performance increase in the checks for incoming messages by making one call to `(MPI_Testsome)` instead of looping over MPI requests and making multiple calls to `(MPI_Test)`.

3. PERFORMANCE RESULTS

The four algorithms have been tested on two problems in a constant work scaling study. The first is a perfectly load balanced problem and the second is a mildly load-imbalanced problem.

All timings include only the particle transport section of the code and do not include things such as input, output, or startup costs. The simulations were run on a Linux cluster with dual 3.05 GHz Pentium Xeon nodes and Myrinet interconnects between the nodes. For all except the one-processor runs, both processors on a compute node were used. In all the results, there is a drop in efficiency from one to two processors mainly due to the fact that the memory bandwidth is shared between the processors.

Only the parallel communication section of the code was modified for each of the algorithms; the physics routines remained the same. The code has been designed to get identical results on

varying numbers of processors[4, 7], which has the consequence that the particles can be simulated in order. Because of this feature of the original KULL IMC package, all results from all algorithms on any number of processors are identical.

Each simulation was run a number of times, and the average run time computed for each case using

$$\bar{t} = \frac{1}{N} \sum_{i=1}^N t_i, \quad (1)$$

where t_i is the time from a given run, and N is the total number of runs. The estimated error in the mean[10] is computed using

$$\sigma_{\bar{t}} = \sqrt{\frac{1}{N(N-1)} \sum_{i=1}^N (t_i - \bar{t})^2} \quad (2)$$

The parallel efficiency for an algorithm, for a given number of processors, is computed using

$$\epsilon_p = \frac{\bar{t}_{\text{best serial}}}{p\bar{t}_p}, \quad (3)$$

where p is the number of processors, \bar{t}_p is the average time, and $\bar{t}_{\text{best serial}}$ is the average time from the best serial algorithm. In practice, all four algorithms had very similar serial run times for both problems. Because the errors in the run times are all independent, we can estimate the error in the efficiency by adding the errors of the run times from Equation 3 in quadrature[10], namely

$$\sigma_{\epsilon_p} = \epsilon_p \sqrt{\left(\frac{\sigma_{\bar{t}_{\text{best serial}}}}{\bar{t}_{\text{best serial}}}\right)^2 + \left(\frac{\sigma_{\bar{t}_p}}{\bar{t}_p}\right)^2} \quad (4)$$

The error bars on the figures are computed using this prescription.

3.1. A Hot Box

This problem is a cube with one centimeter long sides and is discretized with sixty zones per side for a total of 216,000 zones in the mesh. All boundaries are reflecting. The box is filled with a uniform, hot material at $T = 1.1604505 \times 10^7$ K (1 keV), with an absorption cross section of $\sigma_a = 5000 \text{ m}^{-1}$, with a scattering cross section of $\sigma_s = 1000 \text{ m}^{-1}$, with a density of $\rho = 1000 \text{ kg/m}^3$, and a heat capacity of $C_v = 5 \times 10^9 \text{ J/K kg}$. Twenty time steps were computed, each with a constant size of $\Delta t = 3 \times 10^{-9}$ sec. With these parameters, the effective scattering cross section of the Fleck and Cummings method is approximately $\sigma_{\text{eff}} = 6000 \text{ m}^{-1}$. This is designed to be perfectly load balanced during the entire simulation, and each of zones in the mesh is one mean free path thick.

3.1.1. Buffer Size and Message Check Frequency

The message check period, N in Algorithm IV, and maximum buffer size were varied to find the best values for the remainder of the tests. Sixty four processors were used to simulate 69,120,000 particles. Table V shows the run time as a function of buffer size and the message check period

Buffer size (particles)	Message check period (particles)				
	1	2	10	100	1000
10	536.7	495.2	463.0		
100	498.7	450.2	424.4	409.1	
1000	488.1	437.3	405.0	400.6	440.1
5000	484.6	432.7	405.4	395.4	399.1

Algorithm V. Run time in seconds as a function of buffer size and message check period with the Algorithm IV on a sixty four processor hot box problem.

N. Generally the bigger the buffer and the longer time between checking for incoming messages the higher the parallel efficiency and the lower the run time. With bigger buffers, fewer messages need to be sent. Longer check periods also means less work done to support the parallel algorithm. However, if the buffer is too big, the processors can run out of memory, and the problem will fail to run. Additionally, if messages are not checked frequently enough, the run times can increase by orders of magnitudes since processors will be waiting on each other to receive messages. In certain circumstances, typically where the message check period was equal to or greater than the buffer size, we've noticed a locking of the processors. Buffer size and message check period are likely to depend on both machine and problem. We chose a buffer size of 5000 particles and a message check period of 100 particles for the remainder of the tests in the Algorithm IV.

3.1.2. Constant Work Scaling

Figure 5 shows the constant work efficiency of the four algorithms with four million particles per time step. The same physical mesh was decomposed into roughly cubic chunks, one for each processor; the same problem was run on an increasing number of processors. Each case was run ten times. Algorithm I and Algorithm III do not scale well, each for a different reason. Algorithm I has a serialized communication pattern, as discussed in Section 2.1. In Algorithm III the master processor spends a lot of time checking for messages from all other processors. This leads to a significant load imbalance as the number of processors is increased. Algorithm II scales very well, but suffers slightly, when compared to Algorithm IV, from the multiple global communications within each time step. Algorithm IV scales very well to 244 processors. In fact, the biggest performance decrease happened between one and two processors and is more a result of machine architecture than of the algorithm behavior.

3.2. A Vacuum Box

This is the same mesh as the hot box problem, but there is no material in the mesh. It is initially cold, with a uniform isotropic source of $T = 3.5 \times 10^5$ K on one side. Initially the load balance is not good, but by the end of the time step, the box is uniformly filled with particles. Only one time step of $\Delta t = 3 \times 10^{-9}$ sec was run.

Figure 6 shows the efficiency results from a constant work study using one million particles. Three runs for each case were used to compute the average run times and efficiencies. The

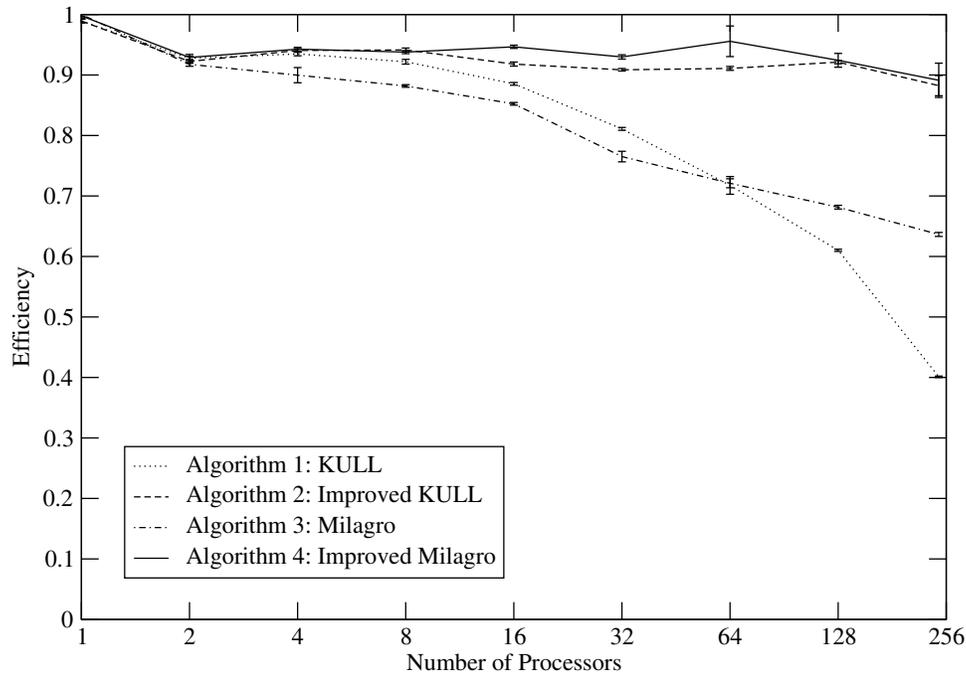


Figure 5. The average parallel efficiency ε , defined in Equation 3, for the constant work hot box problem. Ten runs were computed with each case. The error bars are computed using Equation 4.

efficiency actually improves for this problem as the number of processors increases. As the number of processors is increased, particles traverse the mesh on each processor more quickly, and must be transferred to other processors more often. Particles get transferred to more processors sooner, so the work can be more evenly shared. In the extreme case of two processors for Algorithm I and Algorithm II, Processor 1 must move all the particles from the source boundary to the sub domain boundary while Processor 2 waits to receive some particles. Once Processor 1 is finished, it sends the particles to Processor 2, which then moves the particles to completion while Processor 1 sits idle, resulting in a 50% parallel efficiency. The other two algorithms, Algorithm III and Algorithm IV, do not suffer from this problem as severely because they exchange particle buffers more frequently. Similar things have been noticed before in discrete ordinates simulations, where decomposing a three dimensional mesh into two dimensional columns can dramatically improve performance[11] because information can be exchanged more often allowing otherwise idle processors to do work.

4. CONCLUSIONS

Two production algorithms for asynchronous parallel Implicit Monte Carlo radiation transport were analyzed and then improved. The improved version of the Milagro algorithm, Algorithm IV, performed the best by scaling almost linearly out to 244 processors on a Linux cluster for load balanced problems. The improvements were to check for messages less frequently and to use a scalable, nonblocking version of the standard reduce and broadcast functions. It is critical not to

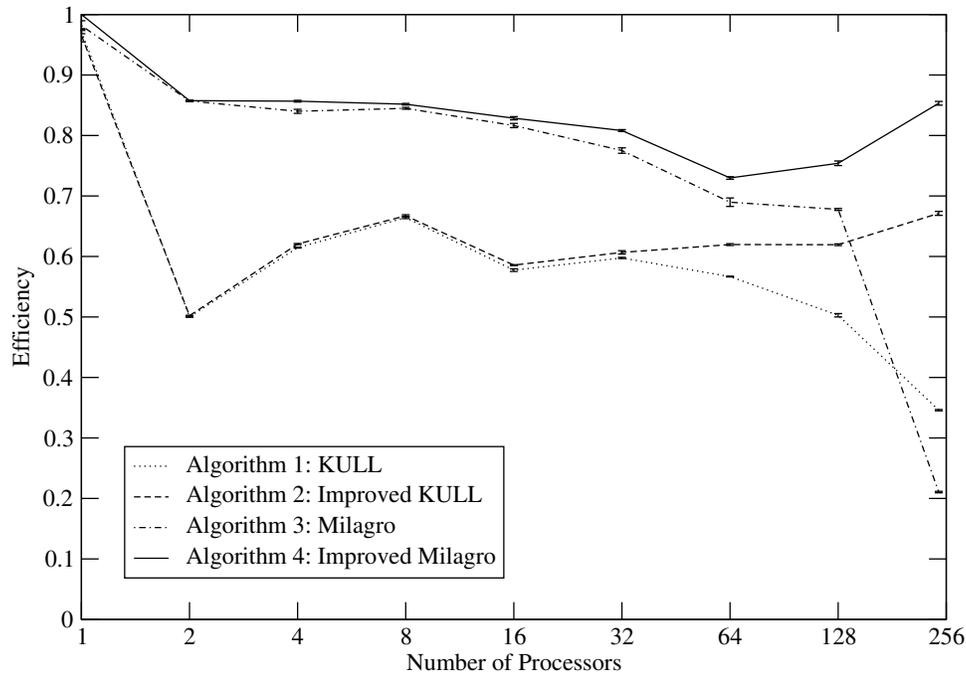


Figure 6. The average parallel efficiency ϵ , defined in Equation 3, for the constant work vacuum box problem. Three runs were computed with each case. The error bars are computed using Equation 4.

have one processor do more work than the others, even if it seems like it is a trivial amount of work, such as checking for incoming messages. The algorithms that used blocking communication do not perform well due to unnecessary contention for processor time.

All of these algorithms begin to suffer when the computational work is not balanced well between processors, as in the vacuum box problem. Load imbalance generally presents the largest obstacle to achieving good scaling in real applications. While this obstacle has not been addressed by these algorithms, achieving good parallel scaling characteristics for load balanced problems is critical for any algorithm that attempts to address the load imbalanced problem.

REFERENCES

1. N. A. Gentile, N. Keen, J. Rathkopf, “The KULL IMC Package,” Tech. Rep. UCRL-JC-132743, Lawrence Livermore National Laboratory, Livermore, CA (1998).
2. T. A. Brunner, T. A. Mehlhorn, “A User’s Guide to Radiation Transport in ALEGRA-HEDP, Version 4.6,” Tech. Rep. SAND2004-5799, Sandia National Laboratories, Albuquerque, NM (2004).
3. J. A. Fleck, Jr., J. D. Cummings, “An Implicit Monte Carlo Scheme for Calculating Time and Frequency Dependent Nonlinear Radiation Transport,” *Journal of Computational Physics*, **8**, pp. 313–342 (1971).
4. N. A. Gentile, M. H. Kalos, T. A. Brunner, “Obtaining Identical Results on Varying Numbers

- of Processors in Domain Decomposed Particle Monte Carlo Simulations,” Tech. Rep. UCRL-PROC-210823, Lawrence Livermore National Laboratory (2005).
5. T. J. Urbatsch, T. M. Evans, “Milagro Version 2, An Implicit Monte Carlo Code for Thermal Radiative Transfer: Capabilities, Development, and Usage,” Tech. Rep. LA-14195-MS, Los Alamos National Laboratory (2005).
 6. T. J. Urbatsch, T. M. Evans, “MILAGRO Implicit Monte Carlo: New Capabilities and Results,” Tech. Rep. LA-UR-00-6118, Los Alamos National Laboratory (2000).
 7. T. J. Urbatsch, T. M. Evans, “Reproducibility in Parallel Monte Carlo Codes,” Los Alamos National Laboratory Memorandum (1999).
 8. Derrick Coetzee, “Binary tree,” Wikipedia article, URL http://en.wikipedia.org/wiki/Binary_tree.
 9. Rolf Rabenseifner, “Optimization of Collective Reduction Operations,” M. Bubak, G. D. v. Albada, P. M. A. Sloot, J. J. Dongarra, editors, *International Conference on Computational Science, Krakow, Poland*, Vol. 3036 of *Lecture Notes in Computer Science*, pp. 1–9, Springer-Verlag (2004).
 10. J. R. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*, University Science Books (1982).
 11. R. Baker, K. Koch, “An S_n algorithm for the massively parallel CM-200 computer,” *Nuclear Science and Engineering*, **128**, pp. 312–320 (1998).