

# **PERFORMANCE MEASUREMENT OF MONTE CARLO PHOTON TRANSPORT ON PARALLEL MACHINES**

Amitava Majumdar  
San Diego Supercomputer Center  
University of California San Diego  
9500 Gilman Drive  
La Jolla CA 92093-0505  
majumdar@sdsc.edu

William R. Martin  
Department of Nuclear Engineering and Radiological Sciences  
University of Michigan  
Ann Arbor MI 48109-2104  
wrm@umich.edu

## **ABSTRACT**

We have parallelized a Monte Carlo algorithm that simulates photon transport in an inertially confined fusion (ICF) plasma. Three different parallel versions of the algorithm were developed. The first version is for the Tera Multi-Threaded Architecture (MTA) and uses Tera specific directives. The second version uses Message Passing Interface (MPI) library calls and is meant for parallel machines that utilize message passing paradigm. We have implemented the MPI version on both the CRAY T3E and the 8-way Symmetric Multi Processor (SMP) IBM SP with Power3 processors. On the SMP IBM SP each of the processors of a SMP IBM SP node was assigned a MPI task. The third version is a hybrid MPI-OpenMP implementation and is used on the SMP IBM SP. This version uses MPI to communicate between nodes and OpenMP to perform shared memory operations among processors within a node. We explain the three different parallelization approaches taken for the three different architectures. Finally, we present parallel performance results of these three parallel implementations on three different machines. We observe near perfect speedup for the three versions on the three architectures. The results on SMP IBM SP nodes suggest that hybrid MPI-OpenMP programming is suitable for SMP type machines.

## 1. INTRODUCTION

Monte Carlo particle transport is an inherently parallel (or embarrassingly parallel) computational method that has been studied on a number of alternative architecture [1]. Currently there are interest to simulate enormously large Monte Carlo particle transport problems for neutron and photon transport on teraflop scale machines under the Department of Energy's Accelerated Strategic Computational Initiative (ASCI) program. Present teraflop scale parallel architectures are switching from being either fully distributed memory or fully shared memory, and multiple node architectures where each node is a Symmetric Multi-processor (SMP) are emerging as the norm. These new architectures encourage a hybrid parallel programming paradigm. In this model computational work load is divided across distributed memory nodes using explicit message passing, and within a node work load is divided across multiple processors using shared memory programming. This does not, however, preclude explicit message passing among multiple processors within a node. Recently there is also interest to explore multithreaded architectures to improve parallel performance of scientific codes. Some future commodity processors may include multithreading capability.

This paper summarizes recent experiences with adapting a Monte Carlo photon transport code to several latest architecture machines, including programming styles, timing results and how they differ for the different architectures. The code has been adapted to different parallel programming styles, such as purely shared memory (using multithreading), purely message passing, and hybrid of message passing and shared memory. Parallel versions of the code were implemented using each of the three programming styles on three different parallel architectures. The different parallel architectures targeted are the shared memory TERA MTA, the distributed memory Cray T3E and the 8-way SMP IBM SP with Power3 processors. For the Tera MTA, directives specific to the Tera MTA was used to parallelize the code. For explicit message passing the Message Passing Library (MPI) was used and for shared memory programming among processors within a SMP node OpenMP was used. In the hybrid programming model, MPI was used across nodes and OpenMP was used across processors within a node.

## 2. DESCRIPTION OF MONTE CARLO PHOTON TRANSPORT CODE

TPHOT, the code used for this parallel performance study, is a time-dependent Monte Carlo photon transport code. TPHOT and other photon transport Monte Carlo codes have been used for several years to examine and benchmark parallel Monte Carlo particle transport on a variety of computers [2,3,4,5,6]. TPHOT simulates photon transport within high-density, high-temperature ICF plasma in two-dimensional r-z geometry, and includes realistic opacity data. The mesh structure is shown in figure 1. The plasma is divided into zones, each with its own composition, temperature, and density. Each zone is a simple volume of revolution, bounded by at most four surfaces. This geometry allows all particles to be treated simultaneously irrespective of zone.

Photons are sampled uniformly and isotropically within each zone from a Planckian energy spectrum. The energy range is discretized into several energy groups. The number of photons emitted within each zone is a function of material properties of the zone and volume of the zone. The photons that are emitted within each zone and energy group are then followed through the plasma until they are absorbed, escape, or reach census, i.e., the end of a time step. Besides absorption, the photons may undergo Thompson scattering.

The overall workload can be divided into four categories, (1) pre-processing work, (2) serial Monte Carlo work, (3) parallel Monte Carlo work, and (4) post-processing work. The pre-processing work includes reading the input data, the serial Monte Carlo work includes preparing geometry and material properties for all the zones, parallel Monte Carlo work includes the actual particle history tracking and accumulating tallies, and the post-processing work includes writing output results. Our measurement of timing does not include the pre-processing and post-processing work.

In TPHOT, the geometry and material properties of zones are constant in time, and time stepping is used only to determine when results are output to a census file. In a more general case, one might model coupled photon transport and hydrodynamics, in which case the geometry and material properties could change from one time step to another. The present simulation includes only the within time step Monte Carlo calculations.

The main part of the particle transport algorithm, i.e. where the parallel Monte Carlo work can be done, has an outer loop over the zones to generate the photons. Inside this loop over zones is a loop over energy groups. In addition, inside the energy group loop is a loop over photons emitted in a particular zone and in a particular energy group. The simulation within a time step continues until the loop over zones has been completed and all of the photons have been emitted and followed.

The photon transport computation is a triply nested loop over zones, energy groups, and photons. Relevant pseudo-code for these loops follows. In the following pseudo code the variable `edep` is the energy deposited in each zone and energy group, and it is a function of the Planckian energy spectrum, the material properties, and the volume of each zone.

```
do i = 1, number_of_zones
  do j = 1, number_of_energy_groups
    .
    .
    (update of some tally variables)
    number_of_photons_in_i_and_j = function of (edep)
    do k = 1, number_of_photons_in_i_and_j
      call multiple_subroutines_to_track_the_photon_history
      (update of tally variables inside subroutines called
      from the innermost loop over
      number_of_photons_in_i_and_j)
    end do
  end do
end do
```

Since the photons do not interact with each other, their histories are independent and can be computed simultaneously. For distributed memory machines accumulations of various tallies at the end of the simulation is the only place where communication is necessary. For shared memory machines the tallies are shared variables, which are updated during each history simulation. This allows an embarrassingly parallel implementation, which should exhibit nearly perfect or linear speedup, provided that (1) the workload can be balanced by a suitable assignment of photons or zones to processors, and (2) each processor has ready access to the geometry and the material properties required for the tracking of particles assigned to that processor. The second condition implies that the geometry and material properties of the whole domain of interest must fit in the local memory of a processor. This is more of a concern for distributed memory machines than shared memory machines since shared memory machines usually have larger memory than distributed memory machines. In case of distributed memory machines, if the geometry and material properties of the whole domain do not fit in the local memory of a processor then the algorithm, although theoretically embarrassingly parallel, is not embarrassingly parallel in practice. If a particle moves to a zone whose properties are not available in the local memory, then communication would be required among processors while tracking a particle history and the algorithm is no longer embarrassingly parallel.

To obtain reproducible results, each processor should also generate an independent reproducible sequence of random numbers, which are not correlated to another processor's random number sequence. We have used the parallel Linear Congruential random number generator from the NAS 2 Parallel Benchmark [7] suite, which satisfies this condition.

### **3. DESCRIPTION OF PARALLEL ARCHITECTURES**

In this section, we provide characteristics of each of the parallel architectures used in this performance evaluation study. The well known architectures of the Cray T3E and the 8-way SMP IBM SP with Power3 processors have been explained briefly, whereas the less well known architecture of the Tera MTA has been explained in more detail.

#### **3.1 THE TERA MTA**

The Tera MTA [8, 9, 10, 11] represents a radical departure from traditional vector- or cache-based computers. MTA processors have no data cache or local memory. Instead, they are connected via a network to commodity memory, configured in a shared memory fashion. Randomized memory mapping and high interconnectivity network provide near-uniform access time from any processor to any memory location. Hardware multithreading is used to tolerate high latencies to memory. This latency is typically on the order of 150 clock cycles. Expected benefits of the MTA include high processor utilization, near linear scalability, and reduced programming effort specially compared to distributed memory machines using explicit message passing.

Each Tera processor has up to 128 hardware streams. Each stream holds the context for one thread in a program counter and 32 registers. The processor switches from one stream to another every clock period, executing instructions from non-blocked streams in a fair fashion approximating round robin. A stream can execute an instruction only once every 21 clocks which is the length of the instruction pipeline. So a minimum of 21 streams are required to keep a processor fully utilized, even if no instructions reference memory. This does imply that single threaded performance on the MTA can be poor. If every instruction required data from the previous instruction, then at most 128 clocks of memory latency could be tolerated. However, each stream can issue 8 memory references without waiting for any to return, so instruction lookahead can often sidestep this problem.

The compiler annotates each instruction with a lookahead number: the number of subsequent instructions from the same stream that can be executed before the memory reference of the current stream must be completed. A lookahead of 0 means that the next instruction needs the current memory reference and cannot be executed until the memory reference has been completed. If lookahead were always 5 then about 30 streams would suffice to saturate the processor. The compiler effectively uses software pipelining to schedule memory references ahead of their uses and often achieves the maximum lookahead of 7.

The processor can issue an instruction containing a memory reference and two other operations per clock period. The other operations can be floating point add and a floating point fused multiply-add. Thus, the theoretical peak speed of a processor is three floating point operations per clock. In practice, no more than two floating point operations per clock have been sustained on realistic computations. Typically, performance is further limited by the network bandwidth, which can return at most one 8 byte (64 bit) word to each processor every clock.

The overhead to acquire a hardware stream is of the order of hundreds of clocks. In general, spawning a thread to do less than 500 instructions worth of work is counter productive. On the MTA it is preferred to perform outer loop parallelization.

Currently the largest MTA and the only one other than Tera company's own machine, is located at the San Diego Supercomputer Center (SDSC). It has 8 processors running at 260 MHz and has 8 gigabyte of shared memory.

### 3.2 THE CRAY T3E

SDSC's Cray T3E has 272 distributed memory processors of which 260 are available for running dedicated parallel applications. Each processor is a DEC Alpha 21164 chip running at 300 MHz clock speed and has 128 megabytes of memory. The DEC Alpha chips are superscalar pipelined chips and each processor can perform two floating point operation per clock period and are capable of a theoretical peak speed of 600 MFLOPS. MPI, SHMEM, and other message passing library calls are used to develop parallel programs on this architecture.

### 3.3 THE IBM SP

SDSC recently received the latest Power3 processor based SMP IBM SP nodes. Currently there are 144 SMP nodes with 8 processors per node. Each SMP node has 4 gigabyte of memory shared among its eight Power3 processors running at 222 MHz each. The Power3 processors, like DEC Alpha chips, are also superscalar pipelined chip and are capable of executing four floating point operation per cycle. The theoretical peak performance of the Power3 chip is 888 MFLOPS [12].

The SP nodes allow symmetric multi-processing among the processors within a node and message passing across nodes. MPI library calls can be used to do message passing across nodes. Other message passing libraries such as MPL, LAPI etc. can also be used to communicate between nodes. Within nodes either OpenMP library calls or pthreads can be used to perform shared memory programming among processors. As mentioned in section 1, MPI can also be used to communicate between the processors within a node i.e. the shared memory processors can be utilized in a message passing paradigm. In this case each processor has access to  $1/8^{\text{th}}$  of the total memory and each processor is assigned one MPI task. One restriction of using MPI within a node is that if one uses only one processor as the only MPI task per node, then only 2 gigabyte of memory is available to that processor. This is due to the 32 bit MPI currently available on the SP. A second restriction of the current machine is that up to 4 MPI tasks on 4 processors may run on a single node when using the fast interconnect switch. To run 8 MPI tasks per node a slower interconnect switch must be used. An environment variable can be set that allows the MPI tasks residing on processors, within a node, to communicate with each other without getting out of the node. In this case MPI communication is done among processors, within a node, by modifying data in the shared memory of the node. Not setting this environment variable causes MPI tasks residing on processors, within a node, to communicate with each other by getting out of the node and then return to the processors in the node.

## 4. PARALLELIZATION ON THE TERA MTA

Two approaches were taken to parallelize TPHOT on the TERA. Below we explain these two parallelization efforts.

### 4.1 PARALLELIZATION BY ZONES ONLY

Since the computations across zones are independent and the Tera MTA prefers outer-loop parallelization, in our first implementation on the MTA we parallelized the outermost zone loop. The inner loops include many subroutine calls and shared accumulators, so the Tera compiler was not able to parallelize the outermost loop, over zones, automatically. Thus an `ASSERT PARALLEL` directive and several `ASSERT LOCAL` directives were inserted to force the compiler to parallelize the loop and to identify the local variables respectively. Moreover, each global tally (such as the number of photons escaped, number of photons absorbed, etc.) had to be preceded with an `UPDATE` directive to insure determinacy. The `UPDATE` directives perform

atomic update on shared variables. The parallel pseudo-code, including the aforementioned directives in bold, is as follows.

```

C$TERA ASSERT PARALLEL
  do i = 1, number_of_zones
C$TERA ASSERT LOCAL (various local variables like
particle's position, velocity, etc. and local variables required
for the random# generator)
    do j = 1, number_of_energy_groups
      .
    .
C$TERA UPDATE directives before each tally statement
    number_of_photons_in_i_and_j = function of (edep)
    do k = 1, number_of_photons_in_i_and_j
      call multiple subroutines to track the photon history
(C$TERA UPDATE directives before each tally statement inside
subroutines called from the innermost loop over
number_of_photons_in_i_and_j)
    end do
  end do
end do

```

As will be seen shortly from Results and Discussion section, parallelization done over zones only does not scale well on the MTA. There was insufficient parallelism to get good load balance among MTA processors. MTA provides a GUI based tools, called TRACEVIEW that allows user to capture and view dynamic execution from files. Using TRACEVIEW it is relatively easy to identify load imbalance among MTA processors.

#### 4.2 PARALLELIZATION BY FUSED ZONES AND ENERGIES

In the second approach, to parallelize for the MTA, parallelism was increased by collapsing the two outer loops over zones and energies into a single loop. This increased the total number of iterations for the loop. In addition, the order of processing the zones was reversed, since the number of photons in a zone is proportional to its size, and the size of a zone grows with a zone's index value. The resulting pseudo-code is as follows.

```

C$TERA ASSERT PARALLEL
  do ij = number_of_zones*number_of_energy_groups-1,0,-1
C$TERA ASSERT LOCAL (various local variables like particle's
position, velocity, etc. and local variables required for the
random# generator)
    .
  .
C$TERA UPDATE directives before each tally statement
  i = (ij/ number_of_energy_groups) + 1
  j = ij - number_of_energy_groups*(i - 1) + 1

```

```

      .
      number_of_photons_in_ij = function of (edep)
      do k = 1, number_of_photons_in_ij
        call multiple subroutines to track the photon history
(C$TERA UPDATE directives before each tally statement inside
subroutines called from the innermost loop over
number_of_photons_in_ij)
      end do
    end do

```

This approach showed near perfect speedup as explained in Results and Discussion section.

## 5. PARALLELIZATION ON THE CRAY T3E

The parallel version of TPHOT developed for the Cray T3E uses the MPI library. MPI is a standard library for message passing and is appropriate for distributed-memory machines such as the T3E. Moreover, the parallelization strategy for the T3E is different from that used on the MTA.

If NP is the number of T3E processors available, then the parallel pseudo-code for the main computation-intensive part of the code is as follows, with the modified part of the pseudo-code in bold letters.

```

do i = 1, number_of_zones
  do j = 1, number_of_energy_groups
    .
    .
    (update of some tally variables)
    number_of_photons_in_i_and_j =function of((edep)/NP)
    do k = 1, number_of_photons_in_i_and_j
      call multiple subroutines to track the photon history
      (update of tally variables inside subroutines called
      from the innermost loop over
      number_of_photons_in_i_and_j)
    end do
  end do
end do
.
.
Call to multiple MPI_REDUCE(..) to add up all tally
variables.

```

This code is executed by each of the NP processors. This makes the energy deposited in each zone and each energy group a fraction  $1/NP$  of the total energy for each processor. The net effect

is that each of the NP processors simulates  $1/NP$  of the total photons over all the zones. This parallelization strategy effectively parallelizes by distributing  $1/NP$  of the total number of photons to each processor. Since each photon history is independent of any other photon history, it is possible to parallelize this way.

At the end of the simulation one of the NP processors needs to perform multiple MPI reduction operations to add up various global tallies (such as the number of photons escaped, number of photons absorbed, etc.) accumulated on each of the NP T3E processors. Besides these reduction operations at the end, no other MPI library calls are required other than the MPI initialization calls at the beginning to identify the total number of processors, each processor's MPI rank, etc.

All the processors execute the initial part of the code where they read input parameters, generate cross sections based on material properties, generate mesh structure for the domain etc. All these cross sections and mesh structure properties are in COMMON block storage and reside on each processor's local memory. As noted previously one requirement for this parallelization strategy, specifically of concern for distributed-memory machines, is that the cross section and mesh structure data for the whole domain must fit in the memory of each processor. As attempts are made to solve larger problems this condition may become a limiting factor on distributed memory machines.

## **6. PARALLELIZATION ON THE IBM SP**

Parallelization on the 8-way SMP Power3 SP was done in two different ways. In the first approach the purely MPI version of TPHOT, developed for the Cray T3E, was used on the SP. In the second approach a hybrid MPI-OpenMP parallel version was developed. In this second version photons were equally distributed among nodes, like the T3E MPI version, and MPI was used to communicate between nodes. On the other hand within each node parallelization was done over zones, similar to the MTA version, using OpenMP among the processors of a SMP node. OpenMP is a standard parallel library for shared memory parallel programming. In next two sections we describe this effort.

### **6.1 PARALLELIZATION USING MPI ON THE IBM SP**

In this approach we used the same parallel version of TPHOT that was used on the T3E i.e. a purely MPI version of the code. To use this version the SMP based SP was treated, from users point of view, as a purely distributed memory machine. The eight processors within a node have access to  $1/8^{\text{th}}$  fraction of the total memory as long as more than one MPI task is used per node. As mentioned in section 3.3 if one processor was used as the only MPI task per node then it had access to only 2 gigabyte of the memory (this is due to the 32-bit MPI currently available on the machine). Message passing was done among the eight processors within a node as well as among processors across nodes using MPI. As mentioned in section 3.3, to use 8 MPI task per node a slower interconnect switch had to be used. The User Space (US) protocol of fast interconnect was used for up to 4 MPI task per node and the slower interconnect of Internet Protocol (IP) was used for 8 MPI task per node. The US and IP are environmental variables that can be set in the

script used for batch submission of jobs on the SP. The environment variable that allows processors within a node to do MPI communication without getting out of the node was set for all of the timing tests.

## 6.2 PARALLELIZATION USING HYBRID MPI-OpenMP ON THE IBM SP

In the hybrid mode of MPI-OpenMP parallelization, TPHOT was first parallelized across the nodes using MPI as described for the T3E parallelization in section 5 i.e. by dividing edep by the total number of MPI tasks. One MPI task was assigned to each SMP node. Then within a node OpenMP was used to parallelize the outer most loop over zones similar to the Tera parallelization described in section 4.1. The parallel pseudo code for the hybrid MPI-OpenMP version of TPHOT is as follows, with the modified part of the pseudocode in bold letters.

```

!$OMP PARALLEL DO
!$OMP& PRIVATE(*newly created temporary tally variables
that are tallied in subroutines called within the
inner most loop over number_of_photons_in_i_and_j)
!$OMP& PRIVATE(various private variables like particle's
position, velocity, etc. and private variables required
for the random# generator)
!$OMP& reduction(**various tally variables)

do i = 1, number_of_zones
  do j = 1, number_of_energy_groups
    .
    .
    (update some **tally variables)
    number_of_photons_in_i_and_j =function of((edep)/NP)
    do k = 1, number_of_photons_in_i_and_j
      call multiple subroutines to track the photon history
      (Pass in *newly create temporary tally variables to
above subroutines and update tallies in them)
      (Do OpenMP reduction operation on these newly created
temporary tally variables returned back from above
subroutines)
    end do
  end do
end do
.
.
Call to various MPI_REDUCE(..) to add up tally variables.

```

The OpenMP version of the code required additional modifications than simply replacing Tera directives with equivalent OpenMP directives. The tally variables are shared variables, located in

COMMON blocks, and to insure proper updates of these tally variables we declared them as OpenMP reduction variables. The OpenMP standard specifies that a private copy of each reduction variable is created for each thread and at the end of the parallel do loop these private copies are reduced to a single variable. When a subroutine (such as the subroutines called within the inner most loops over `number_of_photons_in_i_and_j`) manipulates a variable that exists in a COMMON block, it does not affect the private copy. Hence it was necessary to create additional temporary tally variables (for each of the tally variables that were updated in the subroutines called within the inner most loop over `number_of_photons_in_i_and_j`) and pass them into these subroutines. OpenMP reduction operations done on these new temporary tally variables insured determinacy.

Like the fully MPI implementation, described in section 6.1, MPI\_REDUCE operations were done among MPI tasks across nodes to complete the tally of all the tally variables created in each node. In summary the hybrid version distributes total number of photons equally among nodes, and within a node distributes the zones among processors.

## 7. RESULTS AND DISCUSSIONS

The physical problem simulated is of photon transport through an inertial confinement fusion plasma consisting of a 50%-50% mixture of deuterium and tritium (D-T) at elevated temperature and density. This mixture is surrounded by a SiO<sub>2</sub> region, also at elevated temperature and density. Figure 2 shows the configuration for the test problem. A single time step is modeled, during which approximately 24,000,000 photons are emitted in both regions. The regions are divided up into 1,960 zones, arising from 49 axial mesh intervals and 40 radial mesh intervals. Twelve energy groups are used. For output the code keeps track of various tallies such as the number of photons absorbed and the number of photons escaping the plasma, as well as the number of photons escaping in each energy group, etc.

Tables I, II, III, and IV contain TPHOT timing results on various parallel machines. We provide the wall clock execution time, the speedup, and the efficiency. Speedup is the ratio of the code execution time on one processor to that on multiple processors. Efficiency is defined as the speedup divided by the number of processors.

Table I gives performance results for solving the test problem with TPHOT on the MTA using the two different strategies as explained in section 4. Tables II and III give performance results of TPHOT, parallelized using MPI, on the T3E and the SP respectively. On a single-processor, the MTA is about four times faster than the T3E and about 2.5 times faster than the SP. On the MTA even for one processor parallel execution takes place due to multithreading. Each MTA processor typically used 60 threads. On the MTA the second strategy of parallelizing across fused loops of zones and energies shows better performance even on single processor. This is due to two reasons. The first reason is that switching loop indices from higher to lower allowed better load balance among 60 threads since the volumes of the zones became larger at the beginning and smaller at the end. Number of photons emitted in a zone is proportional to the size of the zone. The second reason is that there are lot more iterations in the loop due to collapsing of zones

and energy groups. This allowed a MTA processor to higher utilization so that memory latency could be tolerated more effectively. Scalability on the T3E is nearly linear to 64 processors. Scalability on the SP is also almost linear to 64 processors. We notice minimal effect of the slow IP switch, which is required for the cases of 8 MPI task per node in current state of the SP machine. Since this MPI implementation of TPHOT has very little communication, except at the end, the effect of the slow switch is minimal even for 8 nodes each using 8 MPI tasks per node..

For the MPI implementation of TPHOT, used on the T3E and the SP, parallelization is done across the 24,000,000 photons. Even on 64 processors, the work per processor ( $=24,000,000/64=375,000$  photon simulations) is substantial. Since the computation is embarrassingly parallel and there is little communication between processors, scalability is linear. Scalability on the MTA is poor when parallelization is only by zones, but nearly linear to 8 processors when parallelization is by both zones and energies. Poor scalability on the MTA when parallelized across just the 1,960 zones is due to insufficient parallelism and poor load balance. For the 8-processor case, each processor covers 245 ( $=1,960/8$ ) zones and simulates all 24,000,000 photons in these 245 zones. On the MTA typically 60 threads are used per processor. This allows each thread to cover on the average about 4 ( $=\sim 245/60$ ) iterations. This does not provide enough work for all the threads in a processor to hide latency efficiently. Since the volumes of the zones differ, being smaller at the beginning and larger at the end of the do loop, this strategy also creates a load imbalance. Parallelizing over fused zones and energy groups resulted in total iteration of 23520 ( $=1960*12$ ) for the outer loop. This allowed each processor to cover 2940 ( $=23520/8$ ) iterations. Switching the direction of the do loops, so that zones with larger volumes are simulated at the beginning of the do loop and zones with smaller volumes are simulated at the end of the do loop, contributed to better load balance. Clearly these modifications provided enough iterations for all the threads, even for 8 MTA processors, to hide latency, amortize overheads, and load balance the work and hence resulted in near perfect scalability.

For this embarrassingly parallel problem, the distributed memory of the T3E and the distributed memory implementation on the SP provide advantage. Characteristic information of each particle, such as a particular particle's position, energy, velocity etc., is naturally kept separate on different processors. The only need for communication is for tallying values (using MPI\_REDUCE) at the end of the time step. By contrast, on the MTA with its shared memory, it is necessary to avoid having different threads use different particle's characteristic information incorrectly. Thus variables that could incorrectly be shared, such as the position, energy, velocity etc. of each particle, need to be identified and declared as local. In addition, the atomicity of tally operations needs to be insured by inserting pragmas before each operation.

The good scalability on the T3E and the SP is made possible because the data for all the zones fit in the local memory of each processor. For much larger problems this would not be possible. If this happens then as soon a particle moves to a zone whose material properties are not available in the local memory, the particle's location, velocity, energy etc. information has to be communicated to another processor. Many such communications, while particle tracking, would negatively affect the scaling of the MPI algorithm. On the MTA, on the other hand, the only issue is parallelism. As long as, the number of zones times the number of energy groups is much larger than the number of streams, performance is expected to scale linearly. And it is expected

that to solve enormously large problems this condition will hold true for the MTA implementation of TPHOT.

Next we discuss the parallel performance of the hybrid MPI-OpenMP implementation of TPHOT on the SMP IBM SP. These results are given in table IV. We notice that the OpenMP parallel performance across processors within a node is near perfect for TPHOT. This is evident from the timing and speedup data shown in the first four rows, with numerical data, of table IV where number of threads increase from 1 to 8. We implemented both of the shared memory parallelization approaches, as explained in section 4.1 and 4.2, using OpenMP. Unlike the MTA there was no difference in performance between these two approaches on the shared memory processors, within a node, of the SP. This is expected on the SP since even for the parallelization over zones there is enough iteration (1960) for 8 OpenMP threads of the SP processors within a node. The scaling of the MPI-OpenMP implementation across nodes is almost linear as evident from the timing results shown in the last five rows of table IV. This is expected since this involves parallelization by distributing photons across nodes. We have observed similar performance for the pure MPI version of TPHOT.

The hybrid MPI-OpenMP version of TPHOT is the most complex, in terms of programming, among the three parallel versions. OpenMP is also prone to false sharing when an array of data is accessed and updated by all the processors of a node and may become the bottleneck due to poor cache utilization. Currently OpenMP compilers are also less robust than MPI compilers and hence require careful programming and testing.

## CONCLUSIONS

In conclusion the Tera MTA, a shared memory machine, has provided encouraging result for TPHOT for up to 8 MTA processors. MTA allowed relatively easy parallelization across fused energy and group loops and showed perfect speedup. The MPI implementation on the T3E and the SP parallelizes across particles, and as expected showed perfect speedup. As mentioned before, to solve enormously large problems the limiting factor would be the local memory size of distributed memory machines such as the T3E. The SP when used as a purely distributed memory machine, as was done for the MPI implementation of TPHOT on the SP, would also encounter this memory size limitation. The hybrid MPI-OpenMP implementation, although difficult to program, debug, and optimize, also provides good performance. In addition, this hybrid programming model allows to use the whole shared memory of a SMP node. It is necessary to use the hybrid MPI-OpenMP mode of TPHOT on the SP since the large share memory available within a node will be required to simulate very large problems. Comparing the scaling results between table III and IV it appears that for large simulations that use very large number of nodes, the hybrid model might outperform the straight MPI model on SMP IBM SP.

## ACKNOWLEDGEMENTS

We would like to thank John Feo of Tera Computer Company for providing valuable suggestions in developing parallel version of the code for the Tera MTA. We also acknowledge the computational resources at the San Diego Supercomputer Center, a research unit of University of California San Diego and funded by National Science Foundation. First author was partially supported by the DARPA contract DABT63-97-C-0028, the NSF grant ASC-9613855, and the NSF NPACI Cooperative Agreement number ACI-9619020.

## REFERENCES

1. W. R. Martin, A. Majumdar, J. A. Rathkopf, and M. Litvin, "Experiences with Different Parallel Programming Paradigms for Monte Carlo Particle Transport Leads to a Portable Toolkit for Parallel Monte Carlo," Proceedings of International Joint Conference on Mathematical Methods and Supercomputing in Nuclear Applications, Karlsruhe, Germany, Vol. II, pp. 418 (April 1993).
2. F. W. Bobrowicz, J. E. Lynch, K. J. Fisher, and J. E. Tabor, "Vectorized Monte Carlo Photon Transport," *Parallel Computing*, **1**, pp. 295-305 (1984).
3. W. R. Martin, P. F. Nowak, and J. A. Rathkopf, "Monte Carlo Photon Transport on a Vector Supercomputer," *IBM Journal of Research and Development*, **30**, pp. 193 (1986).
4. W. R. Martin, T. C. Wan, T. S. Abdel-Rahman, and T. N. Mudge, "Monte Carlo Photon Transport on Shared Memory and Distributed Memory Parallel Processors," *International Journal of Supercomputer Applications*, **1** (3), pp. 57 (1987).
5. P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, D. V. Pryor, "Vectorization of Monte Carlo Particle Transport: An Architectural Study Using the LANL Benchmark "GAMETAB"," Proceedings Supercomputing89, Reno, Nevada (Nov. 13-7, 1989)
6. W. R. Martin and F. B. Brown, "Status of Vectorized, Monte Carlo for Particle Transport Analysis," *International Journal of Supercomputer Applications*, **1** (2), pp. 11 (1987).
7. See <http://science.nas.nasa.gov/Software/NPB>
8. <http://www.Tera.com/www/library/index.html>
9. L. Carter, J. Feo, and A. Snavely, "Performance and Programming Experience on the Tera MTA", Proceeding SIAM Conference on Parallel Processing, San Antonio, Texas (March 1999).

10. A. Snively, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo, and B. Koblenz, "Multi-processor Performance on the Tera MTA," Proceedings Supercomputing98, Orlando, Florida (Nov. 7-13, 1998)
11. J. Boisseau, L. Carter, K. S. Gatlin, A. Majumdar, and A. Snively, "NAS Benchamarks on the TERA MTA", Proceedings Workshop on Multi-Threaded Execution, Architecture, and Compilers (M-TEAC), Las Vegas, Nevada (January 1998).
12. "RS/6000 Scientific and Technical Computing: Power3 Introduction and Tuning Guide", SG24-5155-00.

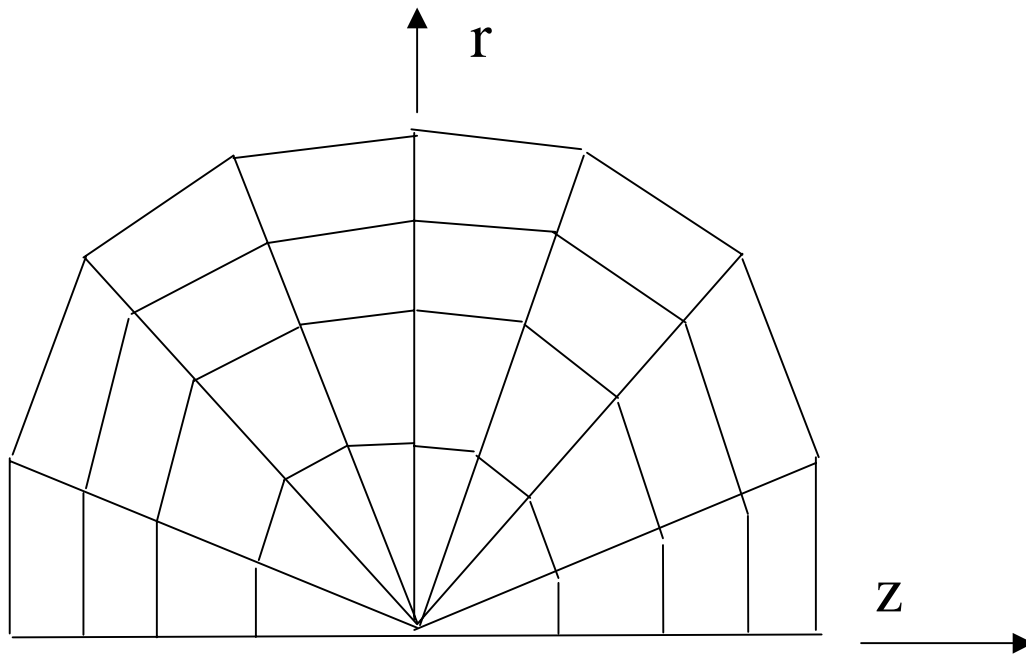


Figure 1. Typical 2D (r-z) Axisymmetric Mesh for a Spherical Configuration.  
(r=radial coordinate and z = axial coordinate)

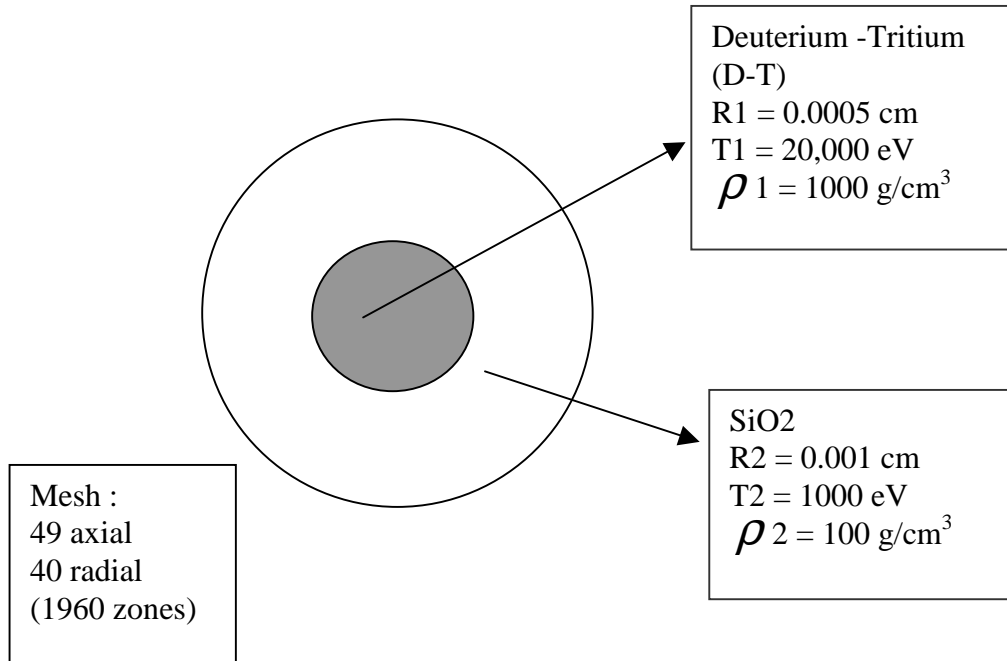


Figure 2. Configuration of ICF Test Problem (two concentric spheres).

Table I. Parallel Performance on the MTA Using Multithreading

Processors	Time(sec)	Speedup	Efficiency
Parallelization by zones only			
1	764	1.00	1.00
2	400	1.91	0.95
4	227	3.37	0.84
8	167	4.58	0.57
Parallelization by fused zones and energies			
1	745	1.00	1.00
2	370	2.01	1.01
4	187	3.98	0.99
8	94	7.92	0.99

Table II. Parallel Performance on the T3E Using MPI

Processors	Time (sec)	Speedup	Efficiency
1	2,997	1.00	1.00
2	1,507	1.99	0.99
4	746	4.01	1.02
8	377	7.95	0.99
16	189	15.86	0.99
32	95	31.55	0.98
64	47	63.76	0.99

Table III. Parallel Performance on the SP Using MPI (\* used the slow IP switch)

# of nodes	# MPI task per node	Total # of MPI tasks	Time(sec)	Speedup	Efficiency
1	1	1	1924	1.00	1.00
1	2	2	963	1.99	0.99
1	4	4	482	3.99	0.99
1	8*	8	243	7.92	0.99
2	2	4	482	3.99	0.99
2	4	8	241	7.98	0.99
2	8*	16	123	15.64	0.97
4	2	8	242	7.95	0.99
4	4	16	121	15.90	0.99
4	8*	32	61	31.54	0.98
8	2	16	121	15.90	0.99
8	4	32	62	31.03	0.96
8	8*	64	32	60.12	0.94
16	8*	128	17	113.17	0.88

Table IV. Parallel Performance on the SP Using MPI-OpenMP

#of nodes	# of MPI task per node	# of OpenMP threads per node	(# of MPI task)*(# of OpenMP threads.)	Time (sec)	Speedup	Efficiency
1	1	1	1	1943	1.00	1.00
1	1	2	2	960	2.02	1.01
1	1	4	4	485	4.00	1.00
1	1	8	8	244	7.96	0.99
2	1	8	16	122	15.92	0.99
4	1	8	32	61	31.85	0.99
8	1	8	64	31	62.67	0.98
16	1	8	128	16	121.43	0.95