Monte Carlo Simulation of Radiation Transport

for Benchmarking

Intel's iPSC/2, iPSC/860, and Touchstone Delta Machine

by

Thomas J. Klemas

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

at the Massachusetts Institute of Technology

December 1992

Author____
Department of Electrical Engineering and Computer Science
December 11, 1992

Certified by____
Gregory M. Papadopoulos
Thesis Supervisor

Accepted by____
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Monte Carlo Simulation of Radiation Transport
for Benchmarking the iPSC/2, iPSC/860, and Intel Touchstone Delta Machine

by

Thomas J. Klemas

Submitted to the
Department of Electrical Engineering and Computer Science

December 11, 1992

In Partial Fulfillment of the Requirements of the Degree
Bachelor of Science in Electrical Engineering

ABSTRACT

A substantial fraction of Los Alamos National Laboratory's computing power is spent in simulating particle transport. This thesis project involves writing a Monte Carlo simulation of radiation transport to be used for benchmarking the Intel iPSC/2, iPSC/860, and Touchstone Delta Machine. The results of such tests would assist scientists in evaluating the performance of the Intel parallel supercomputers. The Monte Carlo simulation of photon and neutron transport exhibits plenty of coarse-grained parallelism. The challenge involved in writing this code for the Intel parallel machines is that the size of the cross-section data set required by the tracking program is greater than the size of the memory at each node. Therefore, the simulation program divides the cross-section data set of large neutron problems into segments, by energy range, that will fit in the memory of a single node. Then, the simulation program sends each segment of cross-section data to the nodes successively and tracks each particle until its energy drops below the current cross-section data segment's lower energy bound. By using a double buffer technique, the delay associated with the communication is overlapped by computation involved with tracking particles; while one buffer of data is being used by the tracking program, the other buffer waits for the next segment of cross-section data to arrive. In this manner, the communication latency is hidden by overlapping it with computation at the nodes. Because each node tracks the same number of particles and the number of particles that the simulation program is tracking is usually very large, the time for one node to complete the tracking of its particles approaches some mean value. Therefore, the processor loads are balanced. Preliminary studies of this implementation strategy are very positive and indicate that is is a feasible approach for parallelizing the Monte Carlo radiation transport simulation. Results suggest that the simulation strategy scales well with problem size and the number of nodes being used.

Thesis Supervisor: Gregory Papadopoulos
Title: Assistant Professor of Electrical Engineering and Computer Science
Supervisor: Olaf Lubeck
Title: Research Staff at Los Alamos National Laboratory

Table of Contents

1. Introduction

A substantial fraction of Los Alamos National Laboratory's computing power is spent in simulating particle transport. This thesis project involves writing a Monte Carlo simulation of radiation transport to be used for benchmarking the Intel iPSC/2, iPSC/860, and Touchstone Delta Machine. The results of such tests would assist scientists in evaluating the performance of the Intel parallel supercomputers. The Monte Carlo simulation of photon and neutron transport exhibits plenty of coarse-grained parallelism. The challenge involved in writing this code for the Intel parallel machines is that the size of the cross-section data set required by the tracking program is greater than the size of the memory at each node. Therefore, the simulation program divides the cross-section data set of large neutron problems into segments, by energy range, that will fit in the memory of a single node. Then, the simulation program sends each segment of cross-section data to the nodes successively and tracks each particle until its energy drops below the current cross-section data segment's lower energy bound. By using a double buffer technique, the delay associated with the communication is overlapped by computation involved with tracking particles; while one buffer of data is being used by the tracking program, the other buffer waits for the next segment of cross-section data to arrive. In this manner, the communication latency is hidden by overlapping it with computation at the nodes. Because each node tracks the same number of particles and the number of particles that the simulation program is tracking is usually very large, the time for one node to complete the tracking of its particles approaches some mean value. Therefore, the processor loads are balanced. Preliminary studies of this implementation strategy are very positive and indicate that is is a feasible approach for parallelizing the Monte Carlo radiation transport simulation.

Results suggest that the simulation strategy scales well with problem size and the number of nodes being used.

The next few sections of this thesis will present benchmarking, radiation transport, simulation of this process using Monte Carlo techniques, specific features of the iPSC/2 parallel supercomputer, strategies for implementing the simulation on the iPSC/2, and initial test results from running the benchmark on the iPSC/2. The paper will conclude with a discussion of the future work to be done on this project.

1.1 Benchmarking

Benchmarking is a process used by researchers to evaluate the performance of a computer for certain types of computations. Because scientists need more computational power in order to solve some of the more complex problems such as climate modeling, fluid turbulence, and pollution dispersion, computer manufacturers are racing to deliver supercomputers which offer improved performance. Supercomputers, however, have no "tell all" statistics by which one can evaluate the performance of a given computer. As a matter of fact, supercomputer performance often varies widely depending on the problem being solved. Thus, benchmarking is a necessary step in the process of determining whether new machines will provide scientists with greater computational power.

At Los Alamos National Laboratories, C3, the High Performance Computing Group, is heavily involved in benchmarking new supercomputers. In order to benchmark new supercomputers successfully, C3 collects statistics regarding the usage patterns of the computers that are already present at the lab.

For example, many scientists at Los Alamos National Laboratories are interested in radiation transport problems. Apparently, 33% of the cycles of the Los Alamos Advanced Computing Facility are spent solving such problems. Because this problem is so important to the laboratory, the simulation of radiation transport is a good candidate for benchmarking new supercomputers.

Traditional high performance computers, such as the Cray supercomputers, have been able to supply the computational power required by the scientific community over the past several decades. These machines typically have a few powerful processors and use techniques such as pipelining to achieve high degrees of throughput. Such machines are called vector machines because they are optimized for performing computations on large vectors of operands that are streamed through the pipelines. However, many researchers are worried that the room for improvement in these vector machines is limited because the degree of parallelism in the pipelines; because it is important to keep the propagation delay of the logic between the registers much greater than that of the registers themselves, the number of stages in a pipeline is bounded (usually pipelines have about 10 stages).

Recently, computer designers have achieved significant improvements in the cost/performance ratio of supercomputers by adopting a new architectural model. In this new model, large numbers of less powerful processors (relative to the CRAY processors) are interconnected. The goal of this approach is to utilize all of the processors to perform operations in parallel. By performing separate computations simultaneously, these massively parallel supercomputers have potential to achieve extremely high throughput. The cost/performance ratio of these machines is superior compared with traditional supercomputers because they are composed of slightly less powerful processors which are often dramatically less expensive than those in the traditional supercomputers. An additional advantage of these machines is that cost and performance are often scalable, allowing users to purchase as much computational power as they can afford.

One disadvantage of these parallel systems is the difficulty associated with identifying and exploiting the parallelism in a user's problem. Some operations are dependent on the results of previous operations and certain data objects are shared throughout the computations of a program. The data dependencies constrain the order of execution of certain instructions within a program.

$$a = f(c)$$

$$b = f(a)$$

In the pseudo-code above, the second operation is dependent on the result of the first operation. Therefore the two operations must be executed sequentially. Traditionally, users write their code sequentially, and the order of execution is determined by the order in which the code is written. However, it is often the case that a significant portion of code in a program is independent of the rest of the program.

$$a = f(c)$$

$$b = f(d)$$

In the above example, the first and second operations are independent of each other and can be executed in parallel (assuming that there are no side-effects performed during execution of function f that constrain the order of the two operations). Detecting parallelism can be quite difficult, especially if the granularity of the parallelism is very large.

The Intel iPSC/2 is an example of a coarse-grained parallel supercomputer. The iPSC/2 uses message passing to explicitly coordinate its 128 numeric processing nodes. Each of the computational elements are powerful, but the time required for communication between the nodes is relatively high. Thus, the iPSC/2 is a coarse-grained parallel computer. This thesis project involved writing a Monte Carlo simulation of radiation transport to be used in benchmarking the iPSC/2 hypercube machine. We developed several strategies to exploit the coarse-grained parallelism inherent in the Monte Carlo photon-neutron tracking problem. The challenge of designing these strategies was to exploit coarse-grained parallelism within the constraints posed by the limited size of the memory at the numeric nodes. We implemented one of these strategies and examined the iPSC/2's performance while running this benchmark.

1.2 Simulation of Radiation Transport

Scientists spend a considerable amount of time and effort to research particle transport problems. One of the primary motivations for this research is to study shielding problems. For example, radiation levels and their effects are of considerable concern to anyone working with radiation. The importance of particle transport simulation is evident when one considers the extensive safety the processors is very high, and the cost of communication is very expensive relative to the cost of computation. Therefore, it is desirable to limit the communication between the processors. However, in order to reduce the communication between processors, the coarseness of parallelism must be increased, but, unlike fine grained parallelism, coarse grained parallelism is difficult to identify. Compilers are not very successful at extracting coarse grained parallelism from programs. As a result, the user is usually left the task of identifying and exploiting coarse grained parallelism for these types of parallel machines. Furthermore, since some problems do not exhibit a high degree of coarse grained parallelism, coarse grained parallel supercomputers are not able to achieve maximal performance on these types of problems. As a matter of fact, the performance of many coarse grained parallel supercomputers can be pretty dismal on problems lacking this type of parallelism. Thus, the performance of parallel machines (non-dataflow) appears to be problem dependent. Clearly benchmarking is extremely important on these machines since their performance varies so greatly.

The Intel iPSC/2 is an example of a coarse grained parallel supercomputer. Message passing is used to explicitly coordinate the computational elements of the iPSC/2, and its 128 numeric processing nodes are extremely powerful. Communication between the nodes is very expensive. Thus, the iPSC/2 is a coarse grained parallel computer. This thesis project involved writing a Monte Carlo simulation of radiation transport to be used in collision can cause the photon to lose energy and/or scatter in a new direction, or can cause the nuclide, with which the photon collided, to absorb the photon and possibly generate new particles termed daughter particles. These interactions include Compton scattering (incoherent or coherent), photo-electric interaction, and pair production.

Compton scattering can occur when a photon collides with an electron. Compton collisions are characterized by the preservation of the total energy and momentum of the particle. Photons can undergo two types of scattering during a Compton collision, incoherent or coherent scattering. The atomic structure of the nuclide partially determines the type of scattering which occurs.

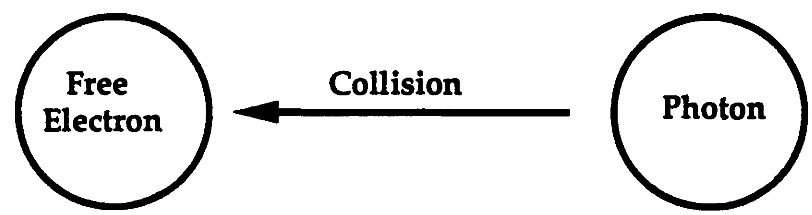


Figure1. During incoherent scattering, a photon collides with a free electron which is not connected with any atom.

Incoherent scattering occurs when a photon collides with a free electron. Since the electron is assumed to be independent of the atom, the simulation ignores interference effects, and the mass of the electron is assumed to be just that of the electron. Because the atomic electron can gain momentum from the collision and possibly become excited and depart from the atom, the photon can lose energy. Furthermore, the collision can redirect the photon.

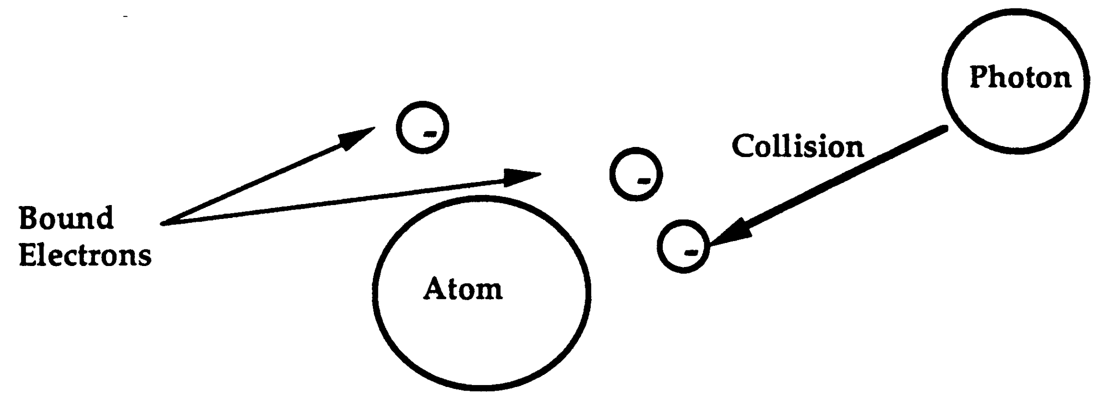


Figure2. During coherent scattering, a photon collides with an electron that is bound to an atom of the material through which the photon is travelling.

When a photon has low energies, coherent scattering will often occur as a result of collisions. The atomic electron involved in the collision remains bound to the atom, and the effective mass of this electron is essentially the mass of the atom. The photon loses no energy during the collision but may change direction.

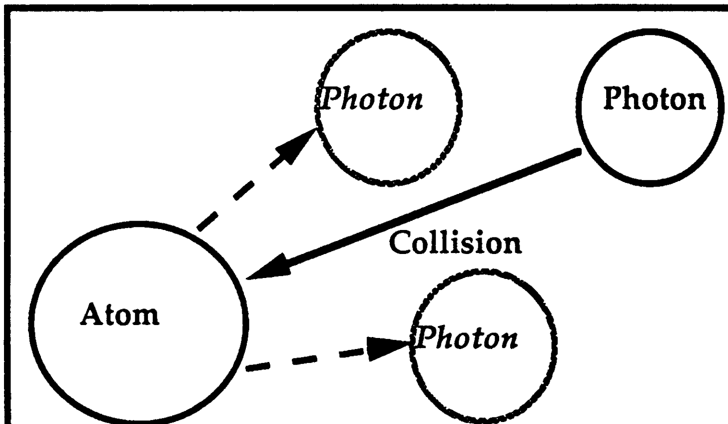


Figure 3. During photo-electric interactions, the atom absorbs the photon and may emit one or two photons. This diagram also describes pair production. During pair production, the photon is also absorbed, and the interaction may result in two new photons with half the energy of the original.

During photo-electric interactions, the atom consumes the photon. If the nuclide has an atomic number greater than twelve, the collision may cause fluorescence, resulting in the emission of one or two photons. The degree of fluorescence that occurs during a photo-electric interaction is also dependent on the photon's energy and the orbital of the atomic electron involved in the collision. These calculations involve discrete energy levels since the atomic electron may only possess certain energy values, depending on its orbital.

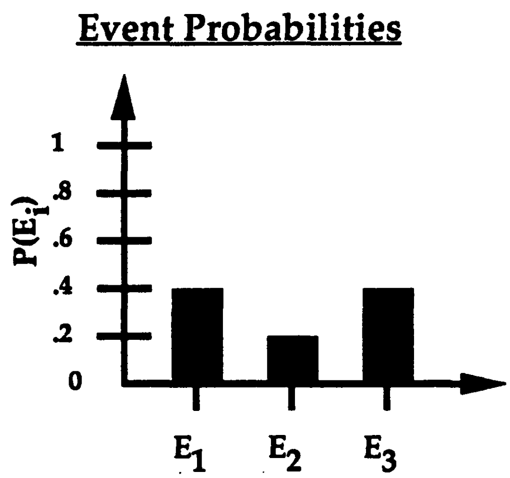
The last of the possible interactions that occur during a photonic collision is pair production. Pair production occurs frequently for particles with energy greater than 1.022 MeV. During pair production, the original photon is absorbed by the atom, and a positron and electron are generated, and the sum of the energy of these two particles is equal to twice the energy of the original photon. The positron is annihilated by combination with an electron, and the result is the creation of two photons with equal energies of .51108 MeV and opposite directions of travel.

1.3 Monte Carlo Techniques

Scientists are often unable to recreate every facet of a problem exactly. Sometimes the calculations involved in solving a problem deterministically are beyond the capacity of contemporary computing resources. Therefore researchers will often construct a simpler model for a system or process, based on experimental studies, and attempt to use the model to study that system or process.

There are several possible events that are associated with every system or process. Researchers are interested in constructing models that will allow them to best predict the possible sequences of events that might occur in processes, given certain initial conditions. Monte Carlo methods provide a technique by which one can simulate the behavior of a system or process using random sampling.

Monte Carlo methods [4] are based on the existence of probability distributions which describe the occurrence of events in a process or system. Often, these probability distributions will be determined experimentally, since they describe processes which are not easily modelled by equations.



Die Rolls for Sampling	Probability	Sampled Events
1,2,3,4	.4	1
5,6	.2	2
7,8,9,10	.4	3

Figure 4. Sampling with a 10 sided die (Discrete Case)

If one has a random number generator that generates uniformly distributed values, and one can determine an inverse of the observed probability distribution function, then one can sample the events which occur by inputting the random numbers into the inverted probability distribution function, as shown in figure 4 for the discrete case. For continuous distribution functions, one can use slightly more complex but analogous techniques to sample events.

To simulate photon transport using Monte Carlo techniques, one must have distribution functions to describe the distance a particle travels before collision and the type of interaction that occurs during a collision. It is possible to implement a simulator in which numbers of particles can be created and tracked through objects of varying geometries and material composition, based on these distribution functions. In the following equations **s** is the distance a photon travels before its next collision, **N** is the density of target particles, **A** is the cross-sectional area presented by the target particles, and **r** is a random number uniformly distributed between 0 and 1.

$$p(s) = 1 - e^{-sNA}$$

p(s) is the probability that a photon will travel at least distance s before its next collision.

$$p(r) = r$$

p(r) is the probability of generating a random number less than or equal to r.

In order to sample let

$$p(s) = p(r)$$

Since both functions are cumulative distribution functions they return a value between 0 and 1.

$$1 - e^{-sNA} = r$$

$$e^{-sNA} = 1 - r$$

After some algebra we obtain

$$s(r) = -\frac{1}{NA}\ln(1 - r)$$

Since (1 - r) and r are random numbers uniformly distributed between 0 and 1 we can substitute r for (1 - r) to obtain the following sampling function:

$$s(r) = -\frac{1}{NA}\ln(r)$$

With a random number generator that can supply uniformly distributed values for r, one can use this sampling function to sample the distance a photon travels before its next collision. Analogous sampling functions exist to sample the type of interaction that occurs during a collision. These functions are the basis for my implementation of a Monte Carlo simulator for radiation transport. The density information is obtained from the problem specifications created by the user of the simulator, and the cross-sectional area used in the functions is contained in a data set compiled by Howerton et al [2].

In order to improve the statistical accuracy of the simulation, variance reduction techniques are used to increase the sampling in regions of the geometry that are of interest. As particles enter areas of greater interest, their number is increased and their weight is decreased. Similarly in areas of less interest the simulator reduces the number of particles and increases the weight of the particles to improve performance. In the user's geometry specification for a problem, each region is labelled with an importance factor that is used with these techniques.

1.4 Machine Description

The Monte Carlo simulation of radiation transport was written for benchmarking several of Intel's parallel supercomputers, including the iPSC/2, the iPSC/860, and the Touchstone Delta machine. [1] [5] [3] Few changes would be required to test all three machines with the simulator because the source code is identical for each of these machines. However, we will only test the iPSC/2 for this thesis project because of time constraints and the lack of available machine time on the other machines. As the other machines are supposed to be successors of the iPSC/2, it is reasonable to expect that their performance would be better. Thus, testing the feasibility of performing neutron simulations on the iPSC/2 should also provide one with sufficient information with which to formulate expectations for the iPSC/860 and the Touchstone Delta Machine.

The iPSC/2 is a MIMD parallel supercomputer. Thus, each node of the 128 processor machine can be loaded with a different set of instructions and data. These nodes are interconnected with a hypercube network configuration, and each node consists of a 386/387 processor and a 4 megabyte memory. The processors run at 16 megahertz.

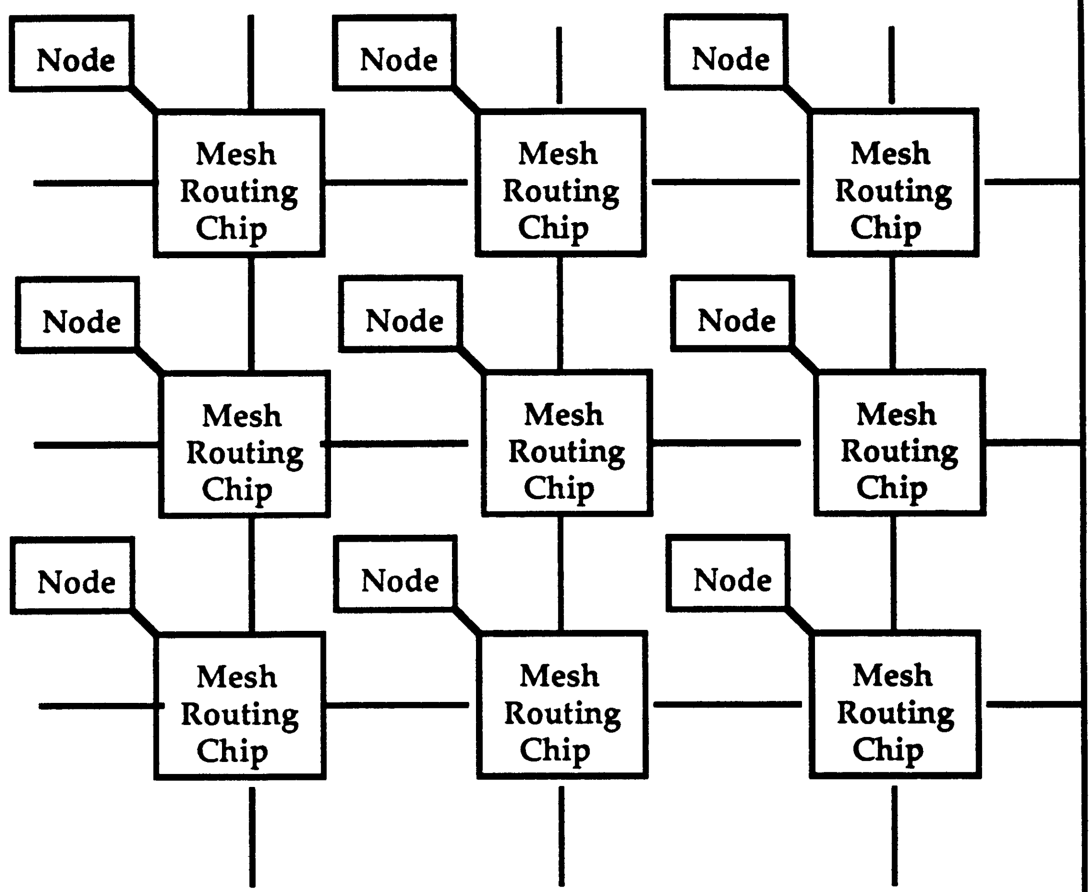


Figure 5. Network Architecture of Intel Touchstone Delta Machine. The iPSC/2 is very similar except that the network is in a hypercube arrangement of greater arity than this 2 dimensional mesh.

The network consists of router modules that connect each node to the network. The router modules allow messages to be routed from the source node to the destination node without interrupting any intermediate nodes. The header of a message creates a path between the source and destination nodes as it travels to the destination node. Once the path is created the rest of the message will follow, and the tail of the message releases the modules comprising the connection. This style of message routing is called circuit-switched routing. The alternative, called packet switched networks, restricts messages to units of packets which contain all routing or header information. In packet-switched networks, connections are only temporarily established between neighboring nodes rather than all of the nodes in the path between the source and destination nodes.

The results of a study [5] that was conducted by researchers at Auburn University provides some insight into the performance of the iPSC/2's communication network. First, the study claims that the number of intermediate nodes between the source and destination nodes does not have a significant impact on the latency of communication. For very small messages(50 to 100 byte messages), the researchers observed that message length had minimal effect on communication latency, but for large messages, the they demonstrated that communication time was directly related to the size of the message. The study concluded that the communication speed was slow relative to processor speed. Thus, based on the study, one reasonable assumption is that the iPSC/2 would perform better on problems with coarse-grained parallelism rather than fine-grained parallelism.

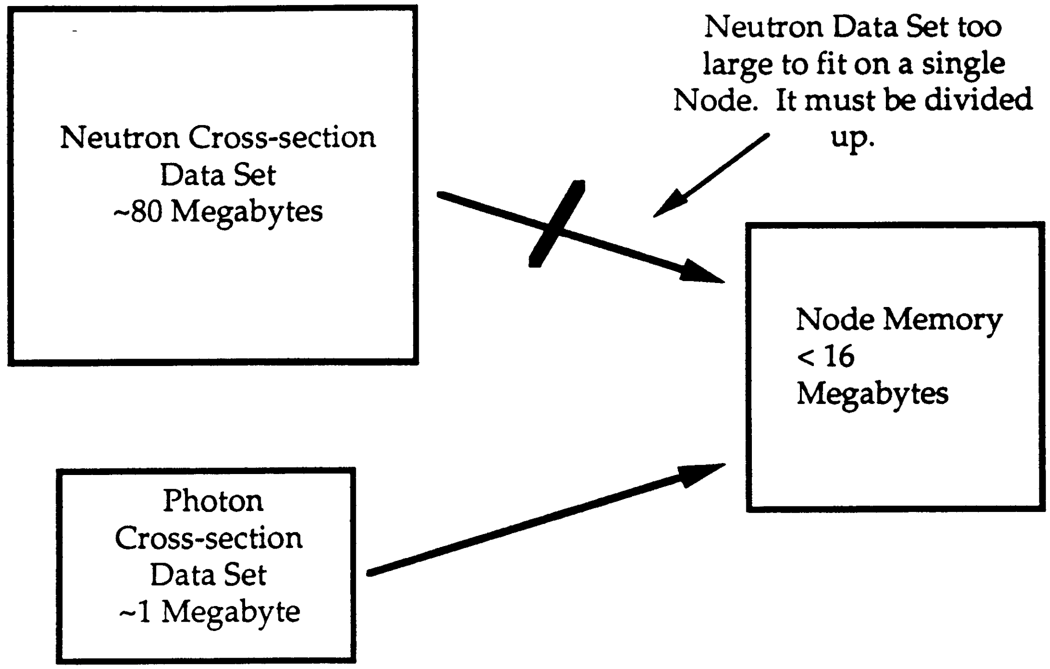
2. Implementation Issues

Until recently uni-processors have dominated the computer arena, and code written for these machines was sequential in nature. Now, parallel processors have been introduced with better performance than the traditional sequential machines. [7] [8] Some parallel computers allow the user to continue to write code in the same sequential style, and compilers for these machines attempt to identify and exploit any parallelism in a program. Most parallel computers, however, require instruction codes written in a different, more complex programming style. For example, Intel's parallel supercomputers boast great computational power, but the user is responsible for identifying parallelism and dividing a task among the processors to obtain optimal performance. Specifically, synchronization and communication must be explicitly inserted into programs to coordinate the activities of nodes, which further complicates the task of writing programs for these machines.

Communication is the overhead associated with dividing a task among the various processors of a parallel machine and coordinating them to complete it. If the amount of time required for communication is great relative to the time required to perform computations, the performance of the parallel machine might be worse than that of a sequential machine. Minimizing the total time that processors spend communicating with each other on the Intel parallel machines is challenging because message throughput is very low relative to computational speed. Furthermore, if the memory requirements of the code and data exceed the size of the memories at the nodes of the Intel machines, the nodes may have to perform additional communication in order to share data . (Node memories range in size between 4 and 16 Mbytes.)

The code for photon transport calculations is less than 1 Mbyte in size and the cross-section required for these computations is about 1 Mbyte in size. Thus, because the nodes can store all of the required data, the simulation of photon transport using Monte Carlo techniques is fairly simple on the Intel machines: one merely has to place a copy of the entire tracking code and the required cross-section data on each node, and start each node tracking its own particles. A node tracks each particle until termination, whereupon it creates a new particle and begins tracking it. Only the current particle's start is maintained, and therefore, the memory requirements of each node are relatively independent of the number of particles being tracked. Furthermore, the only coordination involved in this scheme occurs at the end when the host awaits the tally results from the nodes. The host can only compute the mean and variance of the weights of particles crossing the tally surface once it receives the results from each node. Because simulating photon transport problems requires minimal communication, writing the code is relatively easy and high performance is achieved on the Intel parallel computers.

The simulation of neutron transport is more difficult to implement successfully on the Intel parallel machines than the simulation of photon transport. Although the calculations performed during simulation of neutron transport are very similar to those performed for tracking photons, the cross-section data for neutron problems can reach sizes of approximately 80 Mbytes. Because the cross-section data in a neutron problem is too large to fit into the memory of Intel's machines, the nodes must communicate to share data, which increases the complexity of the simulation program considerably.



One Possible Sequence of Actions in Simulation of Radiation Transport

Photon Transport

1. Host sends cross-section data to Nodes
2. Create particle
3. Track and tally
4. Particle terminates - energy drops below cutoff etc.
5. Repeat step 1 until Node has tracked required number of particles
6. Send tally results to Host
7. Host computes necessary statistics using Node tally results

Neutron Transport

1. Host send first phase of cross-section data to nodes
2. Nodes create all of the required number of particles
3. Nodes track all particles until particle energies drop below lower bound of current phase of cross-section data
4. Nodes request more data
5. Host waits for requests to arrive from all Nodes
6. If all particles have terminated (energies below energy cutoff or absorbed etc.) Host sends next phase of cross-section data. Otherwise skip to step 8
7. Return to step 3
8. Nodes send tally results to host
9. Host computes necessary statistics using Node tally results

Figure 6. This figure demonstrates the difficulties associated with the large cross-section data set. Note that the sample sequence of actions for the neutron transport simulation is significantly more complex than the sequence of actions for the photon transport simulation.

To avoid additional complexity, we did not implement calculations for simulation of neutron transport. Instead, in order to test the feasibility of running neutron transport simulations on the Intel computers, we modified the photon tracking program to model the huge cross-section data set of a typical neutron problem (pretend the data will not fit in the memory of a single node by limiting the available memory for the cross-section data). The modified problem (called pseudo-neutron simulation hereafter) challenged me to identify parallelism and implement a strategy to exploit it with minimal communication. Even though we have not constructed an entirely accurate model of neutron transport simulations, we hope that running the photon and pseudo-neutron transport simulation codes on the Intel supercomputers will provide insight into the performance of these machines on a popular problem: the simulation of photon and neutron transport.

The primary statistics computed by the simulator are the sum of the weights and the sum of the squares of the weights of particles (used for computing the mean and variance of particle weights) that cross the tally surface, a designated surface of interest in the problem. (The weight of a particle that crosses the tally surface several times will be summed for each crossing.) These tallies are computed for all energy levels between 0 MeV and 100 MeV, in 2 MeV increments. Thus, a particle with energy of 15 MeV would be tallied in the 7th energy level(14MeV to 16 MeV) if it crossed a tally surface.

The calculation of the sum of squares of the weights of particles in tally routines can affect the strategy decisions involved in tracking particles and can add complexity because the weight of a source particle includes the weight of all of its daughter particles. Since the tally routines cannot square the tally weight for a source particle until the simulation has tracked all daughter particles, the simulator must maintain a weight-tally entry for each source particle that has been tracked until the tracking of all of its daughter particles has terminated.

If the tracking strategy involves tracking each source particle through to termination, one at a time, and the daughter particles are tracked immediately following the termination of their source particle, this tallying is not difficult. Photon simulation strategies would probably involve this tracking strategy because of its simplicity.

If the tracking strategy involves partial tracking of particles and multiple stages in which all the particles are eventually tracked to termination, a weight tally array must be maintained. The size of the weight-tally array is proportional to the number of particles being tracked in the simulation, and has up to 50 entries for each source particle, one for each corresponding energy level in the problem. Because of the size of the weight-tally array and because source particles can generate a variable amount of daughter particles, managing the node memory can be challenging. Although a tracking strategy in which particles are created initially and partially tracked in stages (until all particles are terminated) is more complex, it might be successful in hiding communication latency by taking advantage of temporal locality for neutron simulations, where a node can only store a portion of the cross-section data at one time.

3. Strategies for Implementation

Implementing a neutron transport simulation on the Intel parallel machines is a challenging problem for the programmer because of the combination of a limited memory size and huge data set. Obviously, in any successful strategy, the data must be segmented in some manner since it will not fit on a single node (for larger problems). Furthermore, either the data or the particles must be passed between nodes and shared. After careful analysis, we have devised several strategies for sharing the data between the nodes.

3.1 Global-Send Phases

The first strategy is the only one which has already been implemented. In this strategy, the cross-section data is split into segments according to energy ranges. Each line of the cross-section data set has a corresponding energy value. Since the natural logarithms of the energy values in the cross-section data are distributed fairly uniformly throughout the data set, it can be divided into approximately equal-sized segments by selecting equal ranges of the natural logarithms of the energy values. Once the data set is evenly divided into segments, the first segment of data is sent to each node. Thus, during phase one, each node receives all of the cross-section data corresponding to the range of particle energies with the highest values.

The first phase of cross-section data is stored in a buffer allocated from the node memory. A request for the second phase of data is sent to the host, although the nodes do not wait for its arrival. All of the particles to be tracked are created before any tracking starts. The particle bank is the array allocated for all of these particles and it contains space for any daughter particles which will be created during the simulation. All particles in the particle bank are tracked through the lower bound of the energy range of the current phase. Once all particles have energies below the lowest energy value of the current segment of cross-section data that is located at the node, the current cross-section data buffer is discarded and the node checks for the arrival of the next phase of data. If it has already arrived, the next phase of cross-section data is installed in a buffer. Otherwise, the node waits for the data to arrive from the host. Once it arrives and is installed as the current energy level, a request for the next phase of data is sent to the host and the node begins tracking all of its particles until their energies drop out of the current range. As before, the node requests the next energy level. This process continues until all particles are terminated.

Although this strategy requires a significant level of communication for the nodes to request data from the host and for the host to send data to the nodes, much of the latency of this communication is hidden because the nodes track particles in the current energy range as the node requests the next energy level. Waiting only occurs if all of the particles have been tracked out of the current energy range and the next phase of data has not yet arrived. However, the phases of cross-section data are sent by global send operations. In other words, the host waits until it receives requests from all nodes before it sends the next phase of data. Thus, if one node takes longer to track all of its particles, it will be delayed in sending a request for the next phase of data. Therefore, since the host waits until it has received requests from all nodes before sending data, all of the rest of the nodes may have to wait idly for the next phase of data if one node takes more time to track its particles.

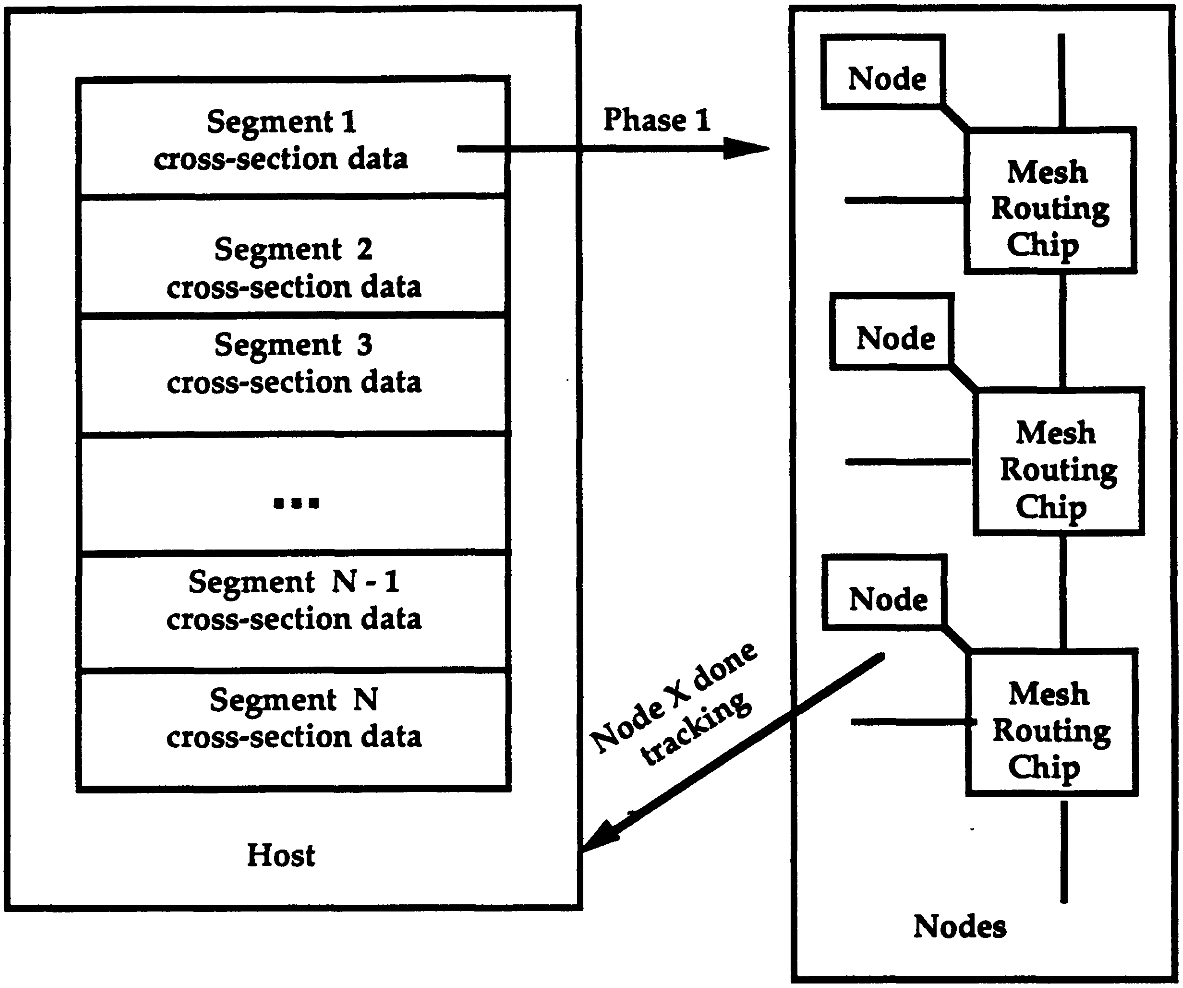


Figure 7. The Global-Send Phases strategy for neutron transport simulation. The large neutron cross-section data set is divided into segments. Each segment is sent to all nodes in a single phase. The nodes track all of their particles during a phase until the energy of each particle drops below the cutoff for that segment of cross-section data. The next segment corresponding to a lower energy range is sent in the next phase. When all of the nodes inform the host that they have completed tracking particles with the current segment of data, the host sends the next segment of the data.

Idle-waiting in nodes corresponds to reduced performance. Thus, it is essential that the nodes are load balanced or possess tasks of roughly equal size in terms of time spent in computation. Because of the randomness associated with the Monte Carlo simulation, it is impossible to guarantee that the particles on each node will require equal time for computation. By the Laws of Probability, however, as the number of particles to be tracked on each node is increased towards infinity, the average amount of time required to track a particle will converge to some value that is the same for all of the nodes. Thus, the randomness of the process will assist in load balancing the nodes, and idle-waiting will be minimized. Furthermore, since the numeric nodes are tracking particles while the cross-section data is being sent(except for the first phase), the latency of communication will be hidden.

An issue that complicated the implementation of this strategy was the tallying of statistics performed by the simulator. As mentioned in the previous sections of this paper, the weights of daughter particles must be added together before squaring for calculating the sum of squares of the weights of source particles. Since the particles are all created initially at each node, a two-dimensional weight-tally array must be created and maintained for each source particle at each energy level (up to 50 is possible). Since this array is proportional to the size of the number of particles, it can be immense, possibly larger than the size of the node memory. This problem was solved by storing the tallies in an array of 50 tally arrays. Each of the tally-arrays contains a weight-tally entry corresponding to each source particle. In order to reduce the memory requirements of these tally-arrays, the array for each energy level is created only when the first particle in that energy level crosses a tally-surface and the array is destroyed when all of the particle energies have dropped below that energy level. The result is that only a few energy level arrays will exist at a time, rather than all 50. This strategy reduces the memory occupied by one energy level weight-tally array.

Another disadvantage of this strategy is that there are situations in which the nodes perform unnecessary work. For example, during the tracking of the particles there are certain energy levels which particles rarely enter. In fact, it is possible that there are entire phases of data that could go unused by the nodes. However, in this strategy, all energy levels of cross-section data are sent to all of the nodes anyway. Obviously, this communication is unneeded and wasteful. Several approaches could be used eliminate the sending of this unneeded data. One approach would be to compute the largest energy value of any particle, so that at the end of the phase the nodes would be able to determine which phase of data they would need next. Phases which most of the nodes required could be sent with global send operations, those required by only a few nodes could be sent individually, and unneeded phases would not be sent at all. Thus, unneeded communication could be eliminated. In current versions of the simulator, all phases are sent to all nodes.

3.2 Pipelined Particle Tracking

In this approach, equal-sized segments of the cross-section data set are each sent to a different node and the data set is divided among the minimum number of nodes that can store all of it. In this manner, nodes are grouped together so that the entire cross-section data set is contained in each group and each node in a group contains a different segment of the data set. Thus, each group is a duplicate of the rest and serves as a pipeline for tracking particles.

Particles are created at the node with the cross-section data corresponding to the highest energy values in the group. They are tracked on the current node until their energy values drop below the lower bound of the energy values corresponding to the cross-section data contained at that node. At this point, the particles are sent to the node which possesses the cross-section data for the next lower energy level. This continues until the particles terminate at the last node in the group. Since photons always lose energy during interactions, they will never have to return to a node or go back up the pipeline.

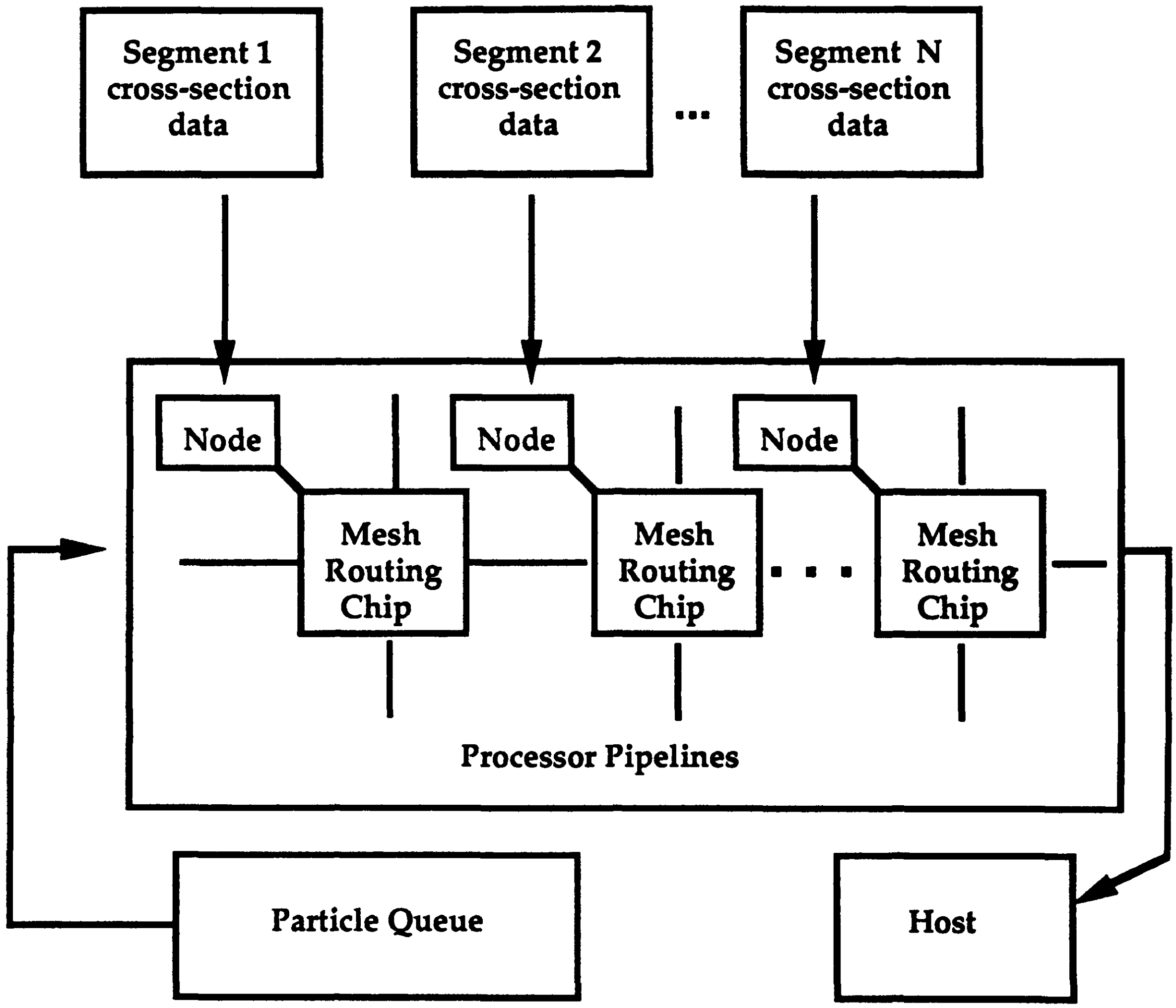


Figure 8. This figure illustrates the Pipelined Particle Tracking strategy. Processor pipelines are composed of groups of N neighboring processors, where N is the number of cross-section data segments. Particles are created and streamed through the pipeline. At each stage the particles are tracked until their energies drop below that stage's lower energy cutoff for the cross-section data. Following tracking, each node sends the host its tally results.

Since studies have shown that an optimum size exists for messages at which maximum bandwidth is achieved, experimentation can be used to determine how many particles should be sent between nodes. This number is also the number of particles that will be created at the first node at the start of each phase and sent to successive stages after being tracked out of each energy level. The total number of phases can be computed by dividing the total number of particles to be tracked by the number of particles that are tracked at each stage and sent between them.

Although this seems to be a rather elegant approach, there are several flaws in this strategy. First of all, all of the nodes in a group except for the node with the highest energy level will be idle at the first phase in which particles are created and tracked at the first node. The rest of the nodes will become busy, one by one, at each successive phase until all are busy. At completion, the nodes become idle, one by one, at each successive phase until the entire pipeline is idle and tracking is completed.

n is the number of nodes in a group

i is the sum of the idle phases spent by the nodes in a group

$$i = n(n - 1)$$

From the above equation above, a total of i node-phases are idle during the start and finish of the simulation. However, if enough particles are tracked, the number of intermediate phases between the filling and emptying of the pipeline will be large enough so that the idle time will be negligible compared to the total tracking time. Thus, the performance of the pipeline approach improves for large numbers of particles.

Another flaw in this strategy is that some stages will take longer than others to track the particles. Some nodes will have to waste time waiting because each stage must wait until the slowest stage has completed the last phase before accepting new particles for tracking and sending the old particles to the next node in the pipeline. This is very similar to the load balancing issue that arose in the global-send approach. In the previous strategy, however, each node had the same data. For a large number of particles, therefore, the average time to track a particle would converge to one value for all of the nodes and the load would be relatively balanced. In this approach, each node in the pipeline contains different cross-section data corresponding to a different range of energy values. Therefore, nodes with certain segments of cross-section data will spend more time tracking particles than others because particle energies are not uniformly distributed.

One way to amortize the cost of sending messages containing particle state between the nodes would be to use the same double buffer technique used in the previous strategy. Thus, while one set of particles is being tracked on a node, the next phase would be arriving into another buffer. Waiting would only occur if the particles in the new phase had not arrived. Once the latest phase of particles had arrived, the node would send a request for the next phase to the node which provides its particles and tracking would begin on the current phase. This technique overlaps some of the communication with computation. However, if there is a difference in the average time for each stage of the pipeline to track a phase of particles, this time differential will show up as idle-waiting and will not be hidden by computation.

In this approach, calculation of the tallies requires the allocation of a tally array for each of the energy-tally-levels in the node's energy range. Each tally array contains separate tally entries corresponding to each source particle so that daughter particle weights can be added before computing the sum of squares of the weights of particles that cross tally surfaces. If the number of particles in each phase is very large, this approach could have memory shortages due to the size of this array. However, the array can be squared and added at the completion of each phase. Since the number of particles in each phase is a constant that is relatively independent of the total number of particles, the size of this array will also be a constant, which makes memory management much easier than in the previous strategy.

The pipeline approach incorporates many of the programming features required in the global-send approach. As a result, implementation of this strategy will be easier since the previous approach has already been implemented. Preliminary analysis reveals that the pipeline would achieve its best performance for large numbers of particles. However, since the stages appear to be mismatched in terms of the time required for computation, the performance under this strategy will never approach the maximal performance possible from the Intel parallel machines.

3.3 Caching Cross-section Data

With this strategy, each node must create and maintain a cross-section data cache. The cache would be used to store recently requested fragments of data that the node was not able to store in its own memory. The cache would take advantage of temporal locality in the cross-sections data requests caused by tracking of particles. Each node would also possess a permanent portion of the data, which it would use for replying to cross-section data requests from other nodes. This strategy could be successful if there was sufficient locality in the cross-section data requests arising from tracking particles.

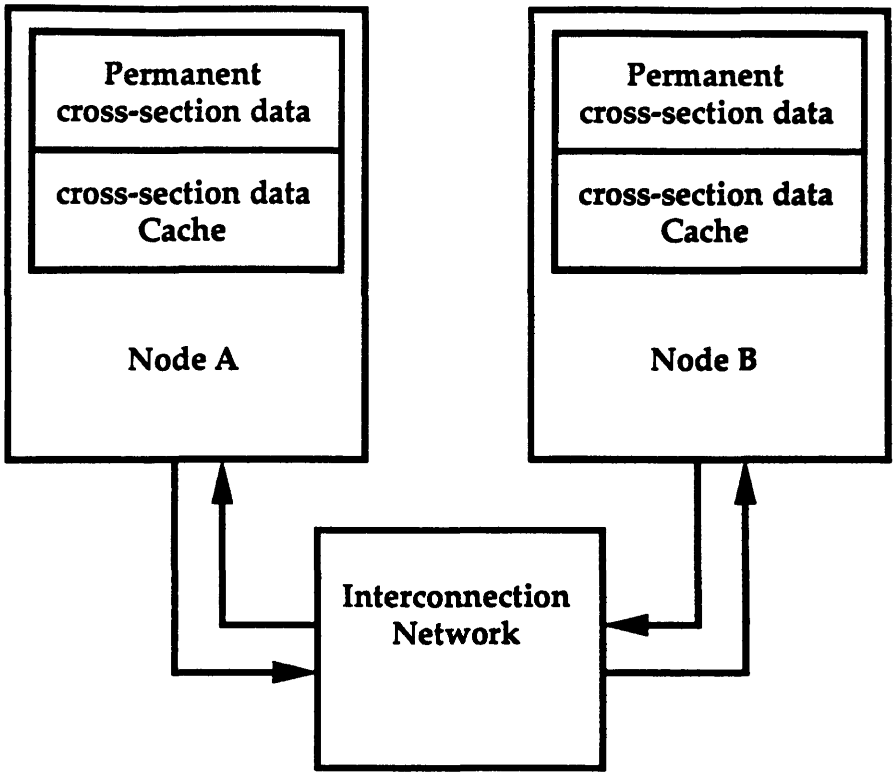


Figure 9. In this figure, Node A has a cache miss and requests the cross-section for a required energy range. Node A can potentially track other particles as it waits for the data from Node B. Node B, which has the requested data in its permanent segment of cross-section data, sends the data to Node A. Node A places the received data in its cache and uses it to track particles.

One advantage of this strategy is that unneeded cross-section data would not be sent to nodes, and since the nodes operate fairly independently, there would not be any global send operations, which require synchronization and can cause idle waiting. One disadvantage of this strategy is that cross-section data can be requested more than once at each node if the cache replaces it with new data. There are two alternative methods that can be used to implement this strategy. Each have potential advantages and disadvantages, but experimental testing would probably be needed to determine which is the best method.

The first of these methods is to create a particle and track it and any of its daughter particles until termination. Similarly, the next particle would be tracked, and so on, until all particles have been tracked. This would eliminate the large, troublesome tally arrays required by the other strategies. However, the success of this approach depends on whether there is sufficient locality in the cross-section data requests generated during the tracking of particles. Specifically, if tracking a particle fills part of the cache with cross-section data, it is desirable that the requests generated from tracking other particles be entirely satisfied by the cache. One possible setback of this method of implementing caches for cross-section data is that thrashing is a likely result if the cache size is too small. In other words, it is likely that the cache will thrash if the cache is small enough that frequently requested data has to be replaced by other frequently requested data.

The second method for implementing the cross-section data caches is to initially create all of the particles, fill the cache by tracking the first few particles, and track all of the rest of them as long as the data that they request is in the cache. Once all of the particles are tracked out of the energy values represented in the cache, the particles are tracked again, adding cache entries until the cache is refilled. Once the cache is again full, the particles are only tracked as long as the cross-section data that they require is in the cache. This process continues until all particles are terminated. This approach sounds similar to the global send approach in many ways, but the primary differences are that communication occurs in smaller messages which contain one energy level (maybe a few with prefetching) and that data will only be sent if it is required by some other node. (In the unmodified global send approach, all data is sent to all nodes regardless of whether it is requested.) In this approach, thrashing is less likely because less data is less likely to be needed after all of the particles are tracked out of a cache than in the previous approach where particles were successively created and tracked until terminated.

Cache size is very important to the success of this general strategy. Although some of the code in the global-send strategy can be reused for this approach, the design and implementation of the caches would still be fairly complicated. Furthermore, a great deal of experimentation and fine tuning would be required to obtain a successul implementation of this strategy. Caching is a general technique that has been implemented successfully for many different problems. Since there is plenty of locality in the cross-section data references of the simulator, this strategy should be capable of achieving respectable performance.

4. Testing and Results

It is important to understand that the goal of this project was to implement the global-send phases strategy for the pseudo-neutron simulator (photon simulation which mimics a neutron simulation problem by modelling the larger neutron cross-sections data set). As part of this goal we will examine the feasibility of the strategy but will not thoroughly test the simulator program. The reason for making this distinction is so that the reader will take the performance results with a "grain of salt" and understand that this program is a model of a neutron simulator rather than the program itself.

4.1 Testing Correctness

To test the feasiblity of the global-send phases strategy, we chose a problem geometry (see problem illustration), ran a range of different problem-sizes (varying number of particles) on the iPSC/2 supercomputer, and compared the tallied results of the pseudo-neutron simulator to those of the photon simulator. Since all of the tallied statistics for both simulators matched the fortran simulator's results very closely for every test, it seems reasonable to trust the correctness of the global-send phases strategy. Some of the statistics that we compared were the total number of collisions, the total number of tally surface crossings, and the number of entries for each cell. Although these tests lack thoroughness, they do demonstrate that both of the simulators produce very similar results.

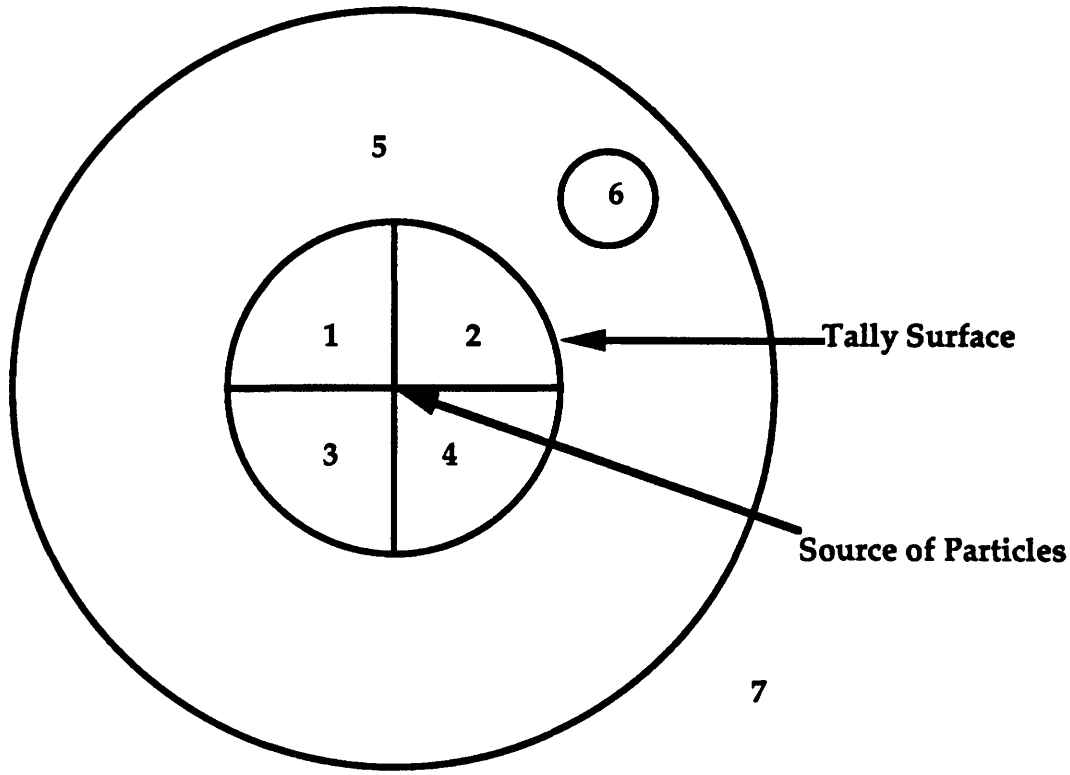


Figure 10. This illustration shows the problem geometry that was chosen for testing the Monte Carlo Radiation Transport Simulator. Each number represents a different cell which can contain different materials. In this case, all cells have the same composition of nuclides except for 6 and 7. Cell 7 is free space, in which particles can have no interactions. When particles cross the indicated tally surface, the tally arrays are modified appropriately to record the event. Particles emanate from the source at the center of the diagram.

One might mistakenly expect that the results of the neutron and photon simulators should be identical because the seeds for the random number generators on the nodes and the tracking routines are identical in each program. (The main difference between the programs is that the photon simulator places the entire photon cross-section data set on each node and the pseudo-neutron simulator models the larger cross-section data set of a neutron problem, which may be too large to place in the memory of a single node.) The reason the results are not identical is that the photon simulator tracks each particle through to completion without interruption, whereas the pseudo-neutron simulator tracks particles only until they drop out of the energy range for the current phase of cross-section data. Thus, although the sequences of random numbers which serve as the basis for the Monte Carlo simulations are the same for each simulator, the photon and pseudo-neutron simulators utilize their random number sequence differently.

4.2 Testing Performance

As mentioned earlier, testing performance was not one of the goals of this project. However, in order to check the feasibility of the chosen strategy and my implementation of it, it is necessary to examine its performance. Therefore, we performed tests to help determine if the global-send phases strategy for the pseudo-neutron simulation was able to achieve reasonable performance. In order to accomplish this, we ran the same problem geometry (see problem illustration) several times with differing numbers of particles and timed each of the simulations. we also ran the same tests on the photon transport simulation in order to gather information about the difference in performance between the pseudo-neutron simulation, which partitions the cross-section data set into segments and only stores two of the segments on the node at any time, and the photon simulation, which places the entire cross-section data set on the node at once. The results of running both simulations on the iPSC/2 are shown in Figures 11 and 12 (based on tables 1 and 2 in Appendix A). Analysis of these results revealed several important observations.

First of all, small problems took very little time (about 10 seconds for tracking 50 particles) to complete relative to large problems(about 20 minutes for tracking 20,000 particles), even though they require the same amount of communication to send the cross-section data set (see Figure 11). Since communication takes very little time and is independent of problem-size(numbers of particles), both the photon simulation and the pseudo-neutron simulation scale very nicely as the problem size is increased. The results of the tests demonstrate that the particle-throughput increased as the problem size was increased: for 50 particles, the simulation tracked about 5 particles per second, and for 1,280,000 particles the simulation tracked over 1000 particles per second.

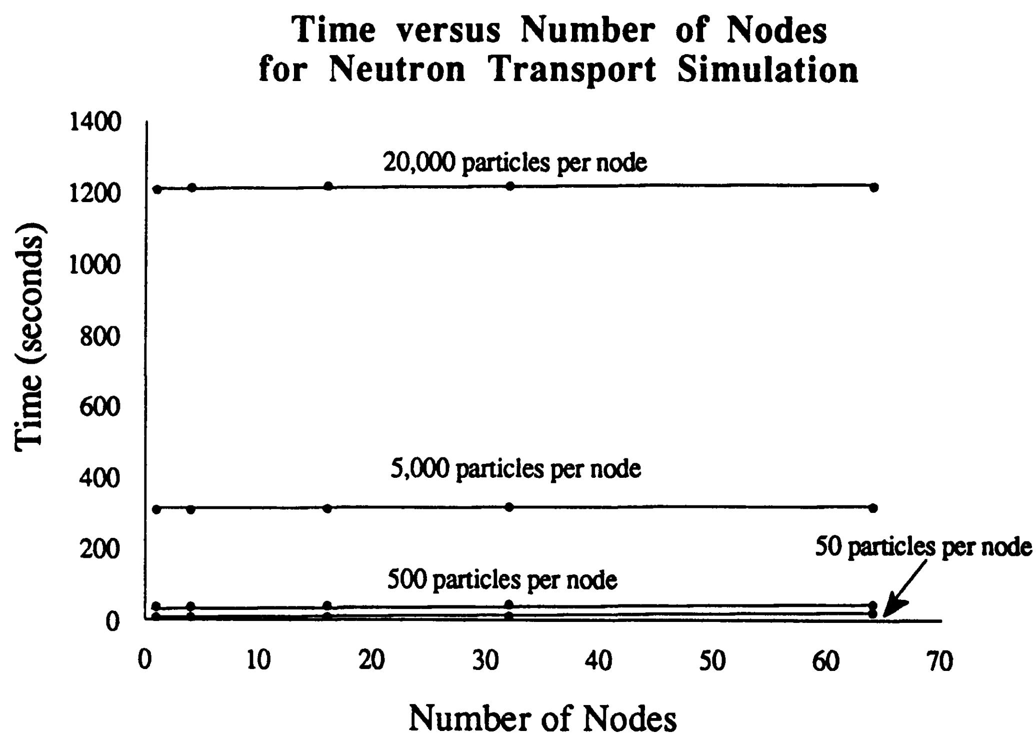


Figure 11. This graph demonstrates that the neutron simulation time is dependent on the number of particles but independent of the number of nodes used in a simulation. (see table 1 Appendix A).

The increase in particle-throughput results from the increase in the ratio of time spent in computation(increases with problem-size) to the time spent in communication(independent of problem-size).

One significant point to note for pseudo-neutron simulations is that the increase in particle-througput is limited by the size of node memory because the size of the tally arrays and the particle bank are dependent on the number of particles and must fit in the node memory. The maximum particle-throughput corresponds to the largest problem (number of particles) in which the node memory is full with the particle bank, tally arrays, and the rest of the data, which has a size that is problem independent. Larger problems can be simulated by repeating simulations which track the largest number of particles that fit on a node, but performance will be limited to the maximum particle-throughput attained when all of the additional node memory is utilized to hold particles and tally-results.

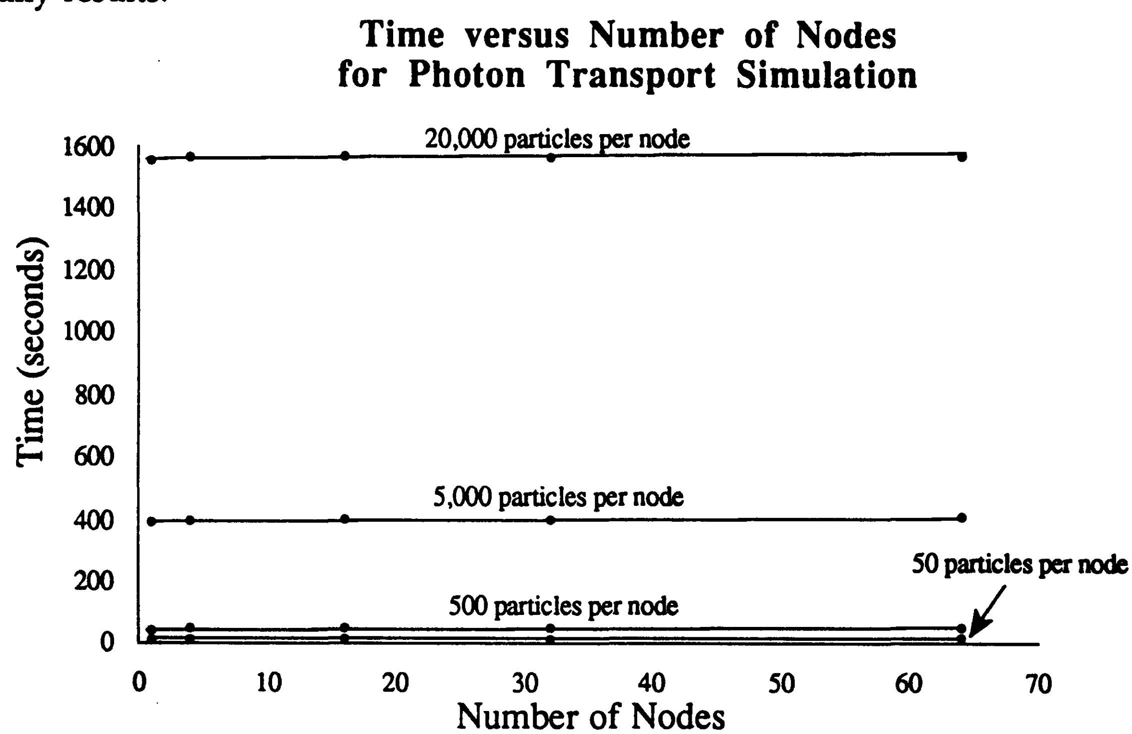


Figure 12. As in the case of the neutron transport simulation, this graph demonstrates that the photon simulation time is dependent on the number of particles but independent of the number of nodes used in a simulation. (see table 2 Appendix A)

Another observation, based on the results in Figures 11 and 12, is that the photon simulation times are larger than the neutron simulation times. This contradicts our expectations, because we assumed that that extra communication, caused by segmenting the data set and sending it in phases, in the pseudo-neutron simulation would have a severe impact on performance despite the double buffer technique which attempts to mask communication latency. The results indicate, however, that the double buffer technique is effective. Since the communication in the photon simulation occurs in one synchronous message-send, for which the nodes wait, the iPSC/2 nodes in the pseudo-neutron simulator should have a smaller idle-waiting time. Because the idle-waiting time is relatively independent of problem size the pseudo-neutron simulation should have a time advantage, which should be relatively constant for all sizes of a given problem, over the photon simulation.

Further examination of the results in Figures 11 and 12 reveals that the time differential between the pseudo-neutron and photon simulations increases as the problem size increases. These unexpected results are explained by the following differences in the tracking routines of the pseudo-neutron and photon simulators: the photon simulator creates a particle and tracks it, thus avoiding the necessity of maintaining state and tally information for all source particles at once, and the pseudo-neutron simulator creates all particles immediately and tracks them all partially, until new cross-section data is needed. Because the pseudo-neutron simulator tracks all particles until they fall out of the current phase's energy range and maintains tally entries for each source particle for each energy level, it only squares and adds weights in the affected energy level tallies for each source particle before continuing to track. The photon simulator, on the other hand, must square all 50 tally entries, that are added as the current source and its daughters are tracked for each particle before tracking the next particle.

One important shortcoming of this pseudo-neutron simulation model is that neutrons, unlike photons, can gain energy during interactions. As a result, particles can reenter the energy range of previous cross-section data phases. This behavior is undesirable, especially if a particle's energy leaves the current phase's energy range because previous phases of the cross-section data will have to be sent to the node in order to complete tracking of such particles. One simple approach to this problem is to repeat sending all phases of data, in the same order as before, until all particles are tracked to termination. It is difficult to estimate the impact of requiring several cycles of global-send phases, but, since the common case is that particles lose energy as a result of collisions, it is likely that this general strategy would still have reasonable performance.

The second set of measurements was taken to determine whether or not the global-send phases strategy achieves reasonable performance. These tests were comprised by measuring the time from the start to the finish of a simulation for a range of different simulation sizes. Just by looking at the largest problem size, it is easy to demonstrate the feasibility of this strategy. For 64 nodes, each tracking 20,000 particles, the pseudo-neutron simulation took about 20 minutes. Thus, the iPSC/2 can track over 64,000 particles in one minute. This stand-alone statistic, 64,000 particles per minute, is pretty impressive. Although these numbers do give us significant information for determining feasibility, preliminary results were also obtained from running the same range of simulations on the Cray 2 for purposes of comparison. The results of running the same problem-geometry on the Cray 2 suggests that the iPSC/2 performance is inferior to that of the Cray. However, these results are not included because the radiation transport simulation implementation on the iPSC/2 has not been sufficiently tested for correctness to allow comparison to other machines. The results in tables 1 and 2 are only really useful for looking at scaling issues. The significance of the numbers is in relation to each other rather than as a performance measurement.

When one compares the times for the pseudo-neutron simulations to the times for the photon simulations one discovers that the neutron simulation took less time to track particles even though it assumes a larger cross-section data set. Although, this observation appears to make no sense, it can be explained by remembering that the global-send phases strategy for the pseudo-neutron simulation employs the double-buffer technique which overlaps communication with tracking of particles. In contrast, the photon simulation sends cross-section data synchronously in one big send while the nodes sit waiting. Thus, the more frequent communication of phases of cross-section data in the pseudo-neutron simulation had less impact on the total time required to track particles than the few initial messages in the photon simulation, because the nodes in the pseudo-neutron simulation track particles as the messages containing the cross-section are sent rather than waiting idly for required data to arrive.

5. Conclusion

The testing performed on the pseudo-neutron simulator suggests that the global-send phases strategy can achieve respectable performance. The results compared favourably to those of the photon simulation and of the fortran MCNP code on the Cray 2. The double buffer technique for masking message latency with computation seems to be effective. Although we did not thoroughly test correctness and performance, the results of testing this implementation suggest that it is a feasible approach for neutron tracking problems. All goals for this stage of the project, namely implementation and feasibility testing of one strategy for handling complexities of neutron tracking simulations, were met by this thesis. Future work will include complete testing of correctness and performance with a wide range of simulation problems which differ in geometry, number of particles, and number of nodes. The results will be compared to identical simulations on the Cray 2 and possibly to dataflow machines running an Id version of the program.

In this study as well as others, the iPSC/2, one of the earlier Intel parallel supercomputers, has demonstrated that it achieves good performance on problems with sufficient coarse-grained parallelism. Because the iPSC/860 and Touchstone Delta Machine are successors to the iPSC/2 with similar architectures, they will probably demonstrate similar performance levels that reflect their improvements.

Appendix A.

# Nodes	# Particles	Simulation Time (seconds)
1	50	9.7
4	50	9.7
16	50	11.2
32	50	13.2
64	50	21.5
1	500	36.2
4	500	37.7
16	500	40.5
32	500	43.5
64	500	44.7
1	5000	307.9
4	5000	308.1
16	5000	312.2
32	5000	316.6
64	5000	317.7
1	20000	1208.1
4	20000	1214.6
16	20000	1219.6
32	20000	1219.5
64	20000	1221.0

Table 1. This table contains the results of a neutron transport simulation. Notice the minimal change in time as the number of processors is increased. This fact that increasing the number of processors does not significantly affect the simulation time indicates that communication, in particular global communication, is minimal. Thus, the results in this table are one indication that this strategy is feasible.

# Nodes	# Particles	Simulation Time (seconds)
1	50	11.4
4	50	13.2
16	50	15.6
32	50	14.3
64	50	18.8
1	500	41.1
4	500	50.0
16	500	50.6
32	500	51.4
64	500	51.6
1	5000	391.4
4	5000	395.8
16	5000	399.5
32	5000	399.0
64	5000	406.4
1	20000	1554.9
4	20000	1566.3
16	20000	1568.3
32	20000	1564.4
64	20000	1569.1

Table 2. This table contains the results of a photon transport simulation. Again, note the minimal change in time as the number of processors is increased. The results in this table are surprising because the simulation times are not less than those in the neutron simulation. This observation, however, is explained by the fact that the neutron simulation strategy overlaps communication of the data set with computation, but the photon simulation just sends the cross-section data set in one big chunk at the start of the simulation.

Appendix B.

```
/* track() tracks a particle through the specified problem
** geometry and simulates its interactions with the materials in
** the object.      In the neutron simulation, the particle can only be tracked
** until its energy has dropped below the lower energy bound of the
** corresponding energy range of the current cross-section phase.  Once
** this has happened, the tracking of a particle can only continue once
** the next phase of data is available from the host (according to the
** global-send phases strategy)      Eventually a particles will terminate and
** tracking will cease.
*/
        void
track(part_p)
        particle        *part_p;        /* pointer to current particle              */
{
        double          distance_c;     /* distance to next collision               */
        double          sigma;          /* cross-sectional area                     */
        cross_sections  cs;             /* cross-section data at energy level       */
        nuclide         t_nucl;         /* nuclide atomic # and % of material       */
        nuclide         *t_nucl_arr;    /* array of structures as above             */
        cell_specs      *temp_c_sp;     /* cell material,importance,& density       */
        char            change;         /* if TRUE then calc. new sigma             */
        int             old_cell_ind;   /* old cell index                           */
        int             nuclide_index;  /* array index for nuclide                  */
        int             num_mat_nuclides;       /* # nuclides in a material         */
        double          rnd;            /* random number between 0 and 1            */
        double          old_erg;        /* used to see if erg changes               */
        int             ind;
        int             i;
        double          k;
        int             curr_mat;


        /* Set up some pointers to arrays */

        temp_c_sp = stats_on_cells + part_p->cell_index;
        curr_mat = temp_c_sp->material;
        t_nucl_arr = material_nucl_arr[curr_mat];
        change = TRUE;

        /* This while loop tracks particles until they are no longer in the
        ** energy range of the current phase of cross-section data.
        */

        while(part_p->ergs > erg_phase_lbnd) {

                /* The cross-section area only needs to be recomputed when
                ** a particle enters a new cell (different material) or has
                ** a collision and changes energy (change == TRUE).
                */

                if(change == TRUE) {
                        num_mat_nuclides = material_num_nuclides[curr_mat];

                        /* Test if the particle energy is less than the upper
                        ** bound of the energy range of the current phase of
                        ** cross-section data.      (part_p->ergs > erg_phase_unbd)
                        ** If this is true, the particle must have gained
                        ** energy, since the next phase is never made available
                        ** until all particle energies have dropped below the
```

```
		** lower bound of the energy range of the current
		** phase.  Thus, there is an error.
		*/

		if(part_p->ergs > erg_phase_ubnd) {
			printf("\n***********************************\n");
			printf("particle erg %f, ergubnd %f ",
			part_p->ergs, erg_phase_lbnd
					+ erg_rng);
			printf("particle_bank %d, curr_particle %d",
					particle_bank, curr_bank_part);
			printf("\nERROR. Particle erg > ergubnd\n");
			printf("\n***********************************\n");
		}
		cs.inc = cs.coh = cs.photo = cs.pp = 0.0;

		/* Calculate the cross-section area for the material
		** in the current cell.
		*/

		for(i = 0; i < num_mat_nuclides; i++) {
			t_nucl = t_nucl_arr[i];
			interpolate(t_nucl.atomic_num,part_p->ergs);
			k = t_nucl.percent;
			/* inc coh photo pp */
			cs.inc += return_cs.inc * k;
			cs.coh += return_cs.coh * k;
			cs.photo += return_cs.photo * k;
			cs.pp += return_cs.pp * k;
			/* totals is used for sampling the nuclide*/
			totals[i] = cs.inc + cs.coh + cs.photo + cs.pp;
		}
		if(part_p->ergs < ERG_THRESHHOLD)
			sigma = cs.inc + cs.coh + cs.photo + cs.pp;
		else
			sigma = cs.inc + cs.photo + cs.pp;
		rnd = RND;

		/* sample to determine the nuclide with which the
		** particle collides
		*/

		for(i = 0; i < num_mat_nuclides; i++)
			if (rnd < (totals[i] / sigma)) {
				nuclide_index = i;
				break;
			}

		/* Just recalculated cross-section area after losing
		** energy or entering new cell. Reset change to
		** false
		*/

	change = FALSE; /* energy or material did not change */
}
/* Sample the distance until the next collision */

distance_c = dist_collision(sigma, temp_c_sp->density);
```

```
/* Calculate the distance until the nearest surface in the
** path of the particle
*/

dist_to_near_surf(part_p);

/* If the distance until the next collision and nearest surface
** are infinite then the particle has escaped the object and
** the particle travels forever in free space.  Therefore
** stop tracking it
*/

if(distance_c >= HUGE_NUM && intrsctn.d >= HUGE_NUM) {
        part_p->ergs = 0.0;
        return;
}

/* If the particle enters a new cell without any collisions
** then determine the new cell and perform any necessary
** tallies
*/

if (distance_c >= intrsctn.d) {
    translate(part_p, intrsctn.d + SMALL_NUM);
    old_cell_ind = part_p->cell_index;
    find_new_cell(part_p, intrsctn.surf_num, old_cell_ind);
    (cell_entries[part_p->cell_index]++);
    if(intrsctn.surf_num == tally_surface) {

                /* Tally mean and variance of energies of
                ** particles that cross the tally surface
                */

            ind = (int)floor((double)(part_p->ergs / 2.0));
            if(tally_v_sums[ind] == NULL) {
                    tally_v_sums[ind] = (double *)
                                calloc(num_particles, sizeof(double));
                    if(tally_v_sums[ind] == NULL) {
                                printf("Tally_v_sums not ");
                                printf("allocated in track!\n");
                                fflush(stdout);
                                exit(0);
                    }
                }
                tally_m[ind] += part_p->weight;
                /* Daughter particle energy values
                ** must be summed for each source
                ** particle separately and then these
                ** sums will be squared and summed
                ** to calculate the variance.
                ** Tally_v_sums is the array
                ** used to sum the daughter erg values
                ** for each source.
                */
                tally_v_sums[ind][part_p->parent_id] +=
                                        part_p->weight;
    }
    variance_reduction(part_p, old_cell_ind);
    temp_c_sp = stats_on_cells + part_p->cell_index;
```

```
                        curr_mat = temp_c_sp->material;
                        t_nucl_arr = material_nucl_arr[curr_mat];
                        change = TRUE;

                /* Otherwise a collision will occur before the particle reaches
                ** a new cell. Sample the type of collision which has occurred
                ** and modify the particle state to reflect the effects of the
                ** collision
                */

            } else {
                    (cell_collisions[part_p->cell_index])++;
                    translate(part_p, distance_c);
                    old_erg = part_p->ergs;
                    if(old_erg < ERG_THRESHHOLD)
                            detailed_photonics(part_p, cs, sigma,
                                    nuclide_index);
                    else
                            photonics(part_p, cs, sigma);
                    if(part_p->weight == 0.0)
                            part_p->ergs = 0.0;
                    if(part_p->ergs != old_erg)
                            change = TRUE;
            }
        }
        /* When the particle's energy value has dropped out of the energy range
        ** of the current phase or has dropped below the energy cutoff for
        ** tracking, it is no longer necessary to track this particle (at least
        ** until the next phase of cross-section data has arrived).  The
        ** particle can also terminate by absorption as a result of collision
        ** or by variance reduction techniques
        */

    return;
}


/* replace_reg_arr_pntrs() called by init_data_arrays(). A cell is a region
** composed of a combination of primitive surfaces (planes, spheres,
** cylinders)  and other regions. Thus a region is a structure which points to
** both an array of pointers to regions and an array of primitive surfaces.
** In order to achieve high efficiency, the host creates one single array of
** regions and another array of primitive surfaces to send to each node.
** The node must recalculate all of the pointers in the structures.
** separate arrays of the same data type into one large array.
** replace_reg_arr_pntrs() recalculates and resets the pointers in cells[]
*/
    void
replace_reg_arr_pntrs(t_reg_arrp, t_surf_arrp)
    region **t_reg_arrp;
    int    **t_surf_arrp;
{
    int    i;
    region *regp;

    regp = *t_reg_arrp;
    regp->surf_number = *t_surf_arrp;
    *t_surf_arrp += regp->num_surfs;
    for(i = 0; i < regp->num_regs; i++) {
```

```
                    (*t_reg_arrp)++;
                    regp->regs[i] = *t_reg_arrp;
                    replace_reg_arr_pntrs(t_reg_arrp, t_surf_arrp);
    }
}


/* The host created single arrays of the one data types out of separate arrays
** of the same data type. Sending data to nodes in one message is more
** efficient because it avoids the overhead of sending multiple messages.
** The host creates other arrays of pointers to the portions of these single
** arrays (subarrays) corresponding to each originally separate array in the new
** larger arrays. init_data_arrays() recalculates the pointers in the arrays
** of pointers used to address the subarrays in the larger arrays.
** init_data_arrays creates arrays of pointers to offsets in arrays for various
** subarrays. It also replaces old array pointers in the region constructs
** with addresses into the arrays of regions and surface numbers;
*/
    void
init_data_arrays()
{
    int    i, j;
    int    tnum_xsects;
    int    temp = 0;
    region *t_reg_arr;
    int    *t_surf_arr;

    /* Allocate arrays */
    incff_arr = (double **) calloc(num_nuclides, sizeof(double *));
    cohff_arr = (double **) calloc(num_nuclides, sizeof(double *));
    fl_arr = (double **) calloc(num_nuclides, sizeof(double *));
    xsect_arr = (xsect **) calloc(num_nuclides, sizeof(xsect *));
    /* Check arrays to make sure they were allocated */
    if(incff_arr == NULL || cohff_arr == NULL || fl_arr == NULL
                    || xsect_arr == NULL) {
            printf("\n**********************************\n");
            printf("array in init_data_arrays() not malloced");
            printf("\n**********************************\n");
            exit(0);
    }
    tnum_xsects = 0;
    incff_tot = 0;
    cohff_tot = 0;
    /* Create arrays of arrays from single arrays */
    for(i = 0; i < num_nuclides; i++) {
            incff_arr[i] = incff + incff_tot;
            cohff_arr[i] = cohff + cohff_tot;
            fl_arr[i] = fl + fl_tot;
            xsect_arr[i] = phase_xsect + tnum_xsects;
            incff_tot += NINCFF;
            cohff_tot += NCOHFF;
            fl_tot += nfls[i];
            tnum_xsects += phase_nxsects[curr_phase][i];
        }
        material_nucl_arr = (nuclide **) calloc(num_materials, sizeof(nuclide *));
    if(material_nucl_arr == NULL) {
                printf("\n**********************************\n");
                printf("material_nucl_arr in init_data_arrays() not malloced");
```

```
		printf("\n**********************************\n");
		exit(0);
	}
	for(i = 0; i < num_materials; i++) {
		material_nucl_arr[i] = material_nuclides + temp;
		temp += material_num_nuclides[i];
	}
	x_ref_array = (int **) calloc(num_surfaces, sizeof(int *));
	if(x_ref_array == NULL) {
		printf("\n**********************************\n");
		printf("x_ref_array in init_data_arrays() not malloced");
		printf("\n**********************************\n");
		exit(0);
	}
	temp = 0;
	for(i = 0; i < num_surfaces; i++) {
		x_ref_array[i] = cross_refs + temp;
		temp += cross_ref_nums[i];
	}
	t_reg_arr = reg_arr;
	t_surf_arr = surf_arr;
	cells = (region **) malloc(num_cells * sizeof(region *));
	if(cells == NULL) {
		printf("\n**********************************\n");
		printf("cells in init_data_arrays() not malloced");
		printf("\n**********************************\n");
		exit(0);
	}

	/* The region structure contains pointers to arrays and structures.
	** Reset the pointers for the arrays allocated at the nodes
	*/
	for(i = 0; i < num_cells; i++) {
		cells[i] = t_reg_arr;
		replace_reg_arr_pntrs(&t_reg_arr, &t_surf_arr);
		t_reg_arr++;
	}
}

/* mcp() starts the particle transport simulation running.  It initializes
** an inputted number of particles to source coordinates and
** samples for direction cosines and energy etc. by calling sample
** source.
*/
	void
mcp()
{
	int		i, j;
	int		tnum_xsects;
	double		t_tally_v;
	int		hi_erg_ind;
	particle	*curr_partp;
	double		curr_erg, max_part_erg;

	/* the bank should have sufficient room for any daughter particles */
	total_bank_size = num_particles + 5000;
	totals = (double *) calloc(num_nuclides, sizeof(double));
	tally_v_sums = (double **) calloc(50, sizeof(double *));
```

```
hi_erg_ind = (int) floor((double) (exp(max_ergval) / 2.0));
for(i = 0; i < hi_erg_ind; i++)
        tally_v_sums[i] = NULL;

/* allocate an array for all of the particles to be tracked */
particle_bank = (particle *) malloc(total_bank_size * sizeof(particle));

/* Did the particle_bank array get allocated? If not, halt! */
if(particle_bank == NULL) {
        printf("\n**********************************\n");
        printf("Particle Bank not malloced");
        printf("\n**********************************\n");
        fflush(stdout);
        exit(0);
}

/* create all source particles initially */
num_bank_parts = num_particles;
for(i = 0; i < num_particles; i++) {
        create_particle(particle_bank + i, i);
}
curr_bank_part = 0;

/* while loops through each phase until all particles terminate */
while(curr_phase > -1) {
        max_part_erg = 0.0;

        /* Track all of the particles that are created until their
        ** energies drop below the lower bound of the energy range
        ** for the current phase
        */
        while(curr_bank_part < num_bank_parts) {
                curr_partp = particle_bank + curr_bank_part;
                if(curr_partp->ergs > erg_phase_lbnd)
                        track(curr_partp);
                curr_erg = curr_partp->ergs;
                if(max_part_erg < curr_erg)
                        max_part_erg = curr_erg;
                curr_bank_part++;
        }

        /* Tally any energy levels if all particle energies
        ** have dropped below that energy level's range. Thus
        ** all daughter particles tallies will be associated with their
        ** parent's tally as required for statistics because no
        ** particles can still cross a tally surface at that energy
        ** level. Once the tally is computed, the tally_v_sums array
        ** with entries for each source particle can be freed for that
        ** energy level.
        */
        while(hi_erg_ind > (int) floor((double)(max_part_erg / 2.0))
                || (curr_phase == 0 && hi_erg_ind == 0)) {
                t_tally_v = 0.0;
                tally_v[hi_erg_ind] = 0.0;
                if(tally_v_sums[hi_erg_ind] != NULL) {
                        for(i = 0; i < num_particles; i++)
                                t_tally_v += pow(tally_v_sums[hi_erg_ind][i],
                                                                 2.0);
```

```
                /* Here we can free the tally_v_sums array
                ** since it is no longer necessary to
                ** distinguish between source particles for
                ** tally purposes because no new daughter
                ** particles can be created with energies
                ** at that energy level.
                */
                free(tally_v_sums[hi_erg_ind]);
                tally_v[hi_erg_ind] = t_tally_v;
        }
        hi_erg_ind--;
}
/* Wait for the cross section data to be put
** in the extra buffer in case it hasn't arrived yet
** If this strategy is any good this waiting will be
** minimal
*/
curr_phase--;
if(curr_phase > - 1) {
        printf("\n********WAITING phase %d**** %d\n", curr_phase,
                                                Node_num);
        /* Wait for the new erg level of cross section data.
        ** Hopefully, in most cases, it has already arrived
        ** because mcp requested it in advance while it was
        ** still tracking particles with the previous phase of
        ** data.
        */
        msgwait(rqstd_xsect_msg_id);

        printf("\n********RECEIVEDphase %d**** %d\n", curr_phase,
                                                Node_num);
        /* Deallocate memory holding the last phase of
        ** cross-section data
        */
        free(phase_xsect);
        phase_xsect = next_phase_xsect;
        num_xsects = num_xsects_next;
        tnum_xsects = 0;
        for(i = 0; i < num_nuclides; i++) {
                xsect_arr[i] = phase_xsect + tnum_xsects;
                tnum_xsects += phase_nxsects[curr_phase][i];
        }

        /* Set upper and lower bound for energy range of
        ** cross-section data
        */
        erg_phase_ubnd = erg_phase_lbnd;
        erg_phase_lbnd = exp(log(erg_phase_lbnd) - erg_rng);
        if(curr_phase == 0)
                erg_phase_lbnd = ERG_CUTOFF;
        curr_bank_part = 0;
}

/* Allocate array for next phase of cross-section data and
** send request for next phase of data
*/
if(curr_phase > 0) { /* curr_phase decrements above*/
        num_xsects_next = phase_num_xsects[curr_phase - 1];
```

```
                        csend(18, &num_xsects_next, sizeof(int), Host_num, 17);
                        next_phase_xsect = (xsect *)
                                calloc(num_xsects_next, sizeof(xsect));
                        if(next_phase_xsect == NULL) {
                        printf("\n**********************************\n");
                        printf("nex_phase_xsect in mcp() not malloced");
                        printf("\n**********************************\n");
                        exit(0);
                        }

                        /* Asynchronous receive for next erg level of cross-
                        ** section data. In the meantime, track all
                        ** particles in the current phase's energy range.
                        ** Double buffer technique masks cost of
                        ** communication and sending data.
                        */
                        rqstd_xsect_msg_id = irecv(17, next_phase_xsect,
                                        sizeof(xsect) * num_xsects_next);
                }
        }

        /* Tally variance of erg levels. these sums include the weight
        ** of all daughter particles. They can only be squared after
        ** all daughter particles for a source have been tracked.
        */
        j = -1;
        csend(18, &j, sizeof(int), Host_num, 17); /* Done */
}


/* main() allocates arrays, receives data from the host, and sets program
** variables. It then calls mcp() to begin tracking particles.          When
** the node terminates particle tracking, tally statistics are sent to the
** host at the end of main(). crecv() is used to receive data from the host
** at the node and csend() is used to send data from the node to the host
*/

main()
{
        int     *int_arr;
        double  *double_arr;
        int     i, j;
        int     int_arr_size = 15;
        int     double_arr_size = 7;
        int     total_offsets;
        int     *offsets;

        /* Allocate arrays */
        int_arr = (int *) calloc(int_arr_size, sizeof(int));
        double_arr = (double *) calloc(double_arr_size, sizeof(double));
        Node_num = mynode();
        Host_num = myhost();
        crecv(1, int_arr, int_arr_size * sizeof(int));
        crecv(2, double_arr, double_arr_size * sizeof(double));
        num_nuclides = int_arr[0];
        num_cells = int_arr[1];
        num_surfaces = int_arr[2];
        num_materials = int_arr[3];
```

```
source_cell_index = int_arr[4];
num_particles = int_arr[5];
tally_surface = int_arr[6];
num_distr_levels = int_arr[7];
tot_material_nuclides = int_arr[8];
total_cell_refs = int_arr[9];
total_regs = int_arr[10];
total_surfs = int_arr[11];
fl_tot = int_arr[12];
Seed = int_arr[13];
num_phases = int_arr[14];

erg_rng = double_arr[0];
source_point_x = double_arr[1];
source_point_y = double_arr[2];
source_point_z = double_arr[3];
max_ergval = double_arr[4];
ERG_THRESHHOLD = double_arr[5];
ERG_CUTOFF = double_arr[6];
erg_phase_ubnd = exp(max_ergval);
erg_phase_lbnd = exp(max_ergval - erg_rng);
srand((2 * Node_num) + Seed);
incff_tot = num_nuclides * NINCFF;
cohff_tot = num_nuclides * NCOHFF;
incff = (double *) calloc(incff_tot + cohff_tot + fl_tot, sizeof(double));

crecv(3, incff, sizeof(double) * (incff_tot + cohff_tot + fl_tot));
cohff = incff + incff_tot;
fl = cohff + cohff_tot;
cross_refs = (int *) calloc(total_cell_refs, sizeof(int));
crecv(4, cross_refs, sizeof(int) * total_cell_refs);
reg_arr = (region *) calloc(total_regs, sizeof(region));
crecv(5, reg_arr, total_regs * sizeof(region));
surf_arr = (int *) calloc(total_surfs, sizeof(int));
crecv(6, surf_arr, total_surfs * sizeof(int));
atomic_numbers = (int *) calloc(num_nuclides, sizeof(int));
crecv(7, atomic_numbers, num_nuclides * sizeof(int));
material_num_nuclides = (int *) calloc(num_materials, sizeof(int));
crecv(8, material_num_nuclides, num_materials * sizeof(int));
material_nuclides = (nuclide *) calloc(tot_material_nuclides, sizeof(nuclide));
crecv(9, material_nuclides, tot_material_nuclides * sizeof(nuclide));
total_offsets = (3 * num_nuclides) + num_materials + num_surfaces;
offsets = (int *) calloc(total_offsets, sizeof(int));
crecv(10, offsets, total_offsets * sizeof(int));
nfls = offsets;
material_num_nuclides = nfls + num_nuclides;
cross_ref_nums = material_num_nuclides + num_materials;
ps = (surf *) calloc(num_surfaces, sizeof(surf));
crecv(11, ps, num_surfaces * sizeof(surf));
stats_on_cells = (cell_specs *) calloc(num_cells, sizeof(cell_specs));
crecv(12, stats_on_cells, num_cells * sizeof(cell_specs));
source_distribution = (e_p_pair *) calloc(num_distr_levels, sizeof(e_p_pair));
crecv(13, source_distribution, num_distr_levels * sizeof(e_p_pair));
phase_num_xsects = (int *) calloc(num_phases, sizeof(int));
crecv(14, phase_num_xsects, num_phases * sizeof(int));
curr_phase = num_phases - 1;
num_xsects = phase_num_xsects[curr_phase];
phase_nxsects = (int **) calloc(num_phases, sizeof(int *));
```

```
	phase_nxsects[0] = (int *) calloc(num_phases * num_nuclides, sizeof(int));
	crecv(15, phase_nxsects[0], num_phases * num_nuclides * sizeof(int));
	for(i = 1; i < num_phases; i++)
		phase_nxsects[i] = phase_nxsects[i - 1] + num_nuclides;
	phase_xsect = (xsect *) calloc(num_xsects, sizeof(xsect));
	crecv(16, phase_xsect, sizeof(xsect) * num_xsects);
	num_xsects_next = phase_num_xsects[curr_phase - 1];
	next_phase_xsect = (xsect *) calloc(num_xsects_next, sizeof(xsect));
	rqstd_xsect_msg_id = irecv(17, next_phase_xsect, sizeof(xsect) * num_xsects_next);
	cell_entries = (int *) calloc(2 * num_cells, sizeof(int));
	cell_collisions = cell_entries + num_cells;
	if(incff == NULL || reg_arr == NULL || phase_nxsects == NULL
			|| phase_nxsects[0] == NULL || surf_arr == NULL
			|| phase_num_xsects == NULL || next_phase_xsect == NULL
			|| stats_on_cells == NULL || source_distribution == NULL
			|| ps == NULL || material_nuclides == NULL
			|| offsets == NULL || material_num_nuclides == NULL
			|| int_arr == NULL || double_arr == NULL
			|| cell_entries == NULL
			|| material_nuclides == NULL || cross_refs == NULL) {
		printf("\n********************************\n");
		printf("Array in main() not malloced");
		printf("\n********************************\n");
		fflush(stdout);
		exit(0);
	}
	init_data_arrays();
	mcp();
	csend(23, tally_v, NUM_ERG_LEVELS * sizeof(double), Host_num, 17);
	csend(24, tally_m, NUM_ERG_LEVELS * sizeof(double), Host_num, 17);
	csend(25, cell_entries, 2 * num_cells * sizeof(int), Host_num, 17);
}
```

References

[1] T. H. Dunigan, Performance of the INTEL iPSC/860 and NCUBE 6400 Hypercubes, Oak Ridge National Laboratory, Oak Ridge, TE, 1991.

[2] R. J. Howerton, R. J. Doyas, T. C. Michels, and S. T. Perkins, An Integrated System for Production of Neutronics and Photonics Calculational Constants, Vol. 4, Evaluated Nuclear Cross Section Library, USAEC Report UCRL-50400, University of California Lawrence Radiation Laboratory, 1971.

[3] Intel Supercomputer Systems Division, A Touchstone Delta System Description, Intel Corporation, Beaverton, OR, 1991.

[4] Malvin H. Kalos and Paula A. Whitlock, Monte Carlo Methods, Volume I: Basics, John Wiley and Sons.

[5] M. E. McArdle, C. L. McCreary, and J. D. McCreary, Broadcast Communication Delay Metric for the iPSC/2 and iPSC/860 Hypercubes, Auburn University, AL.

[6] Roger Prynn, Neutron Scattering A Primer, Number 19, Los Alamos Science, Los Alamos National Laboratory, 1990.

[7] James L. Tomkins and John P. VanDyke, Massively Parallel Computing and the Mid-Course Tracking Problem, Sandia National Laboratories, Albuquerque, NM.

[8] James L. Tomkins and John P. VanDyke, SDI Tracking and Correlating, Massively Parallel Computing Research Laboratory Research Bulletin, Sandia National Laboratories, 1991.