LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JRNL-208745

Comparison of Four Parallel Algorithms For Domain Decomposed Implicit Monte Carlo

T. A. Brunner, T. J. Urbatsch, T. M. Evans, N. A. Gentile

December 22, 2004

Journal of Computational Physics

Comparison of Four Parallel Algorithms for Domain Decomposed Implicit Monte Carlo

Thomas A. Brunner
Sandia National Laboratories
Document Number: SAND-2004-6694J

Todd J. Urbatsch and Thomas M. Evans
Los Alamos National Laboratory
Document Number: LA-UR-05-0290

Nicholas A. Gentile
Lawrence Livermore National Laboratory
Document Number: UCRL-JRNL-208745

Submitted to Journal of Computational Physics, January 2005

Comparison of Four Parallel Algorithms for Domain Decomposed Implicit Monte Carlo

Thomas A. Brunner *

Sandia National Laboratories, PO Box 5800, Albuquerque, NM 87185-1186[1]

Todd J. Urbatsch, Thomas M. Evans

Los Alamos National Laboratory, PO Box 1663, Los Alamos, NM 87545[2]

Nicholas A. Gentile

Lawrence Livermore National Laboratory, 7000 East Ave., Livermore, CA 94550[3]

Abstract

We consider two existing asynchronous parallel algorithms for Implicit Monte Carlo (IMC) thermal radiation transport on spatially decomposed meshes. The two algorithms are from the production codes KULL from Lawrence Livermore National Laboratory and Milagro from Los Alamos National Laboratory. Both algorithms were considered and analyzed in an implementation of the KULL IMC package in ALEGRA, a Sandia National Laboratory high energy density physics code. Improvements were made to both algorithms. The improved Milagro algorithm performed the best by scaling nearly perfectly out to 244 processors.

Key words: Monte Carlo methods, Parallel computation, Radiative transfer

* Corresponding author.

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

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

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

1 Introduction

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

Four different algorithms are outlined, and the performance of each is measured on several different test problems. These algorithms are implemented in the KULL IMC package[1] running inside of ALEGRA[2]. This package implements the Implicit Monte Carlo (IMC) scheme for thermal radiation transport of Fleck and Cummings[3].

These algorithms only address the scalability for domain decomposed meshes that have reasonable particle load balancing. If one processor has significantly more particles than the others, all of the algorithms presented here will scale poorly.

2 Algorithms

In domain decomposed Monte Carlo, two sets of data must be communicated between the processors. The nearest neighbors must exchange particles that cross processor boundaries. A global communication operation must also be performed so that all the processors know when all the other processors are finished moving all the particles. The four algorithms for a time step in the IMC package vary in how they perform each of these two tasks. Specific MPI calls in the algorithms are shown.

2.1 The KULL Algorithm

Algorithm 1 shows the original communication method used by the KULL IMC package[1]. The number of particles that need to be exchanged is not known, so this information must be sent before allocating memory for the receive buffer. It is critical to have a sorted list of the neighbors, otherwise it is possible to get locked cycles of processors, each waiting on another to exchange particles. For example, in a three-processor problem in which all three must communicate, you can get processor one waiting on processor two, processor two waiting on processor three, and processor three waiting on processor one.

Algorithm 1 KULL

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

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

2.2 The ALEGRA-KULL Algorithm

The blocking sends and receives in the original KULL IMC Method lead to non-scalable behavior, like the serialization in the example above. Algorithm 2 is an improved version of Algorithm 1 that uses nonblocking communication and combines the buffer size with each buffer, which eliminates a separate message.

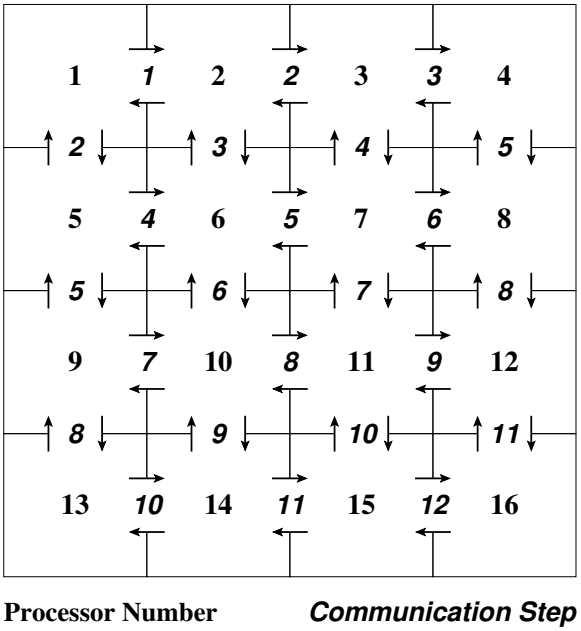

Fig. 1. Communication pattern for the KULL IMC algorithm for a square domain of sixteen processors. It takes twelve communication steps to fully exchange all particles when it should only take four steps.

Algorithm 2 ALEGRA-KULL

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

2.3 The Milagro Algorithm

An algorithm based on the Milagro code[4] is outlined in Algorithm 3. For particle transport on a spatially decomposed domain, each processor continuously loops over mutually exclusive options until every particle finishes.

After simulating each particle, the communicator is checked to see if any parti-

cles have arrived from neighboring domains on other processors. This frequent checking was necessary in the original implementation in Milagro on the SGI Octane "Bluemountain" supercomputer (now decommissioned) at Los Alamos National Laboratory. Skipping even a small number of checks would occasionally lock the processors on that machine.

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

When a processor has no more source particles or incoming particles, it deliberately flushes its buffers, sending only the number of bytes needed to transfer the partially full buffer. The number of particles that are being sent is encoded into the beginning of the message buffer, and extracted when the buffer is received. When there appears to be no more incoming particles, the processor makes any updates to the global count of finished particles. When all particles have been completed, the master processor broadcasts the finished status to all the processors.

2.4 The ALEGRA-Milagro Algorithm

Identifying the deficiencies in the Milagro algorithm, we may propose an improved algorithm. While the Milagro algorithm avoids any global synchronizations during the time step, its scalability is limited in three key areas. The improved version of the Milagro algorithm is shown in Algorithm 4. We refer to this improved version as the ALEGRA-Milagro Algorithm.

The Milagro algorithm uses a fat communication tree for the "particles completed" messages, where processor zero is the parent of all other nodes. The master processor checks for many "particles completed" messages, which causes a load imbalance and poor scaling. The improved Algorithm 3 uses a binary tree communication pattern[5], which is optimal for short messages[6], for the asynchronous "particles completed" communications and the finished message broadcast. This pattern ensures that each processor has an even and minimal workload for incoming message tests.

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

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

Algorithm 3 Milagro

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

3 Performance Results

The four algorithms have been tested on different problems. The first is a perfectly load balanced problem, for which constant work and constant work per

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

Table 1
Run time in seconds as a function of buffer size and message check period.

processor scaling studies are done. The second is a load-imbalanced problem, for which only a constant work scaling study is performed.

All timings include only the particle transport section of the code and do not include things such as input, output, or startup costs. The simulations were run on a Linux cluster with dual 3.05 GHz Pentium Xeon nodes and Myrinet interconnects between the nodes. Each simulation was run three times, and the minimum time was used to calculate the efficiencies.

For all except the one-processor runs, both processors on a compute node were used. In all the results, there is a drop in efficiency from one to two processors mainly due to the fact that the memory bandwidth is shared between the processors.

3.1 A Hot Box

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

3.1.1 Buffer Size and Message Check Frequency

The maximum buffer size and message check period, N in Algorithm 4, was varied to find the best values for the remainder of the tests. Sixty four processors were used to simulate 69,120,000 particles. Table 1 shows the run time

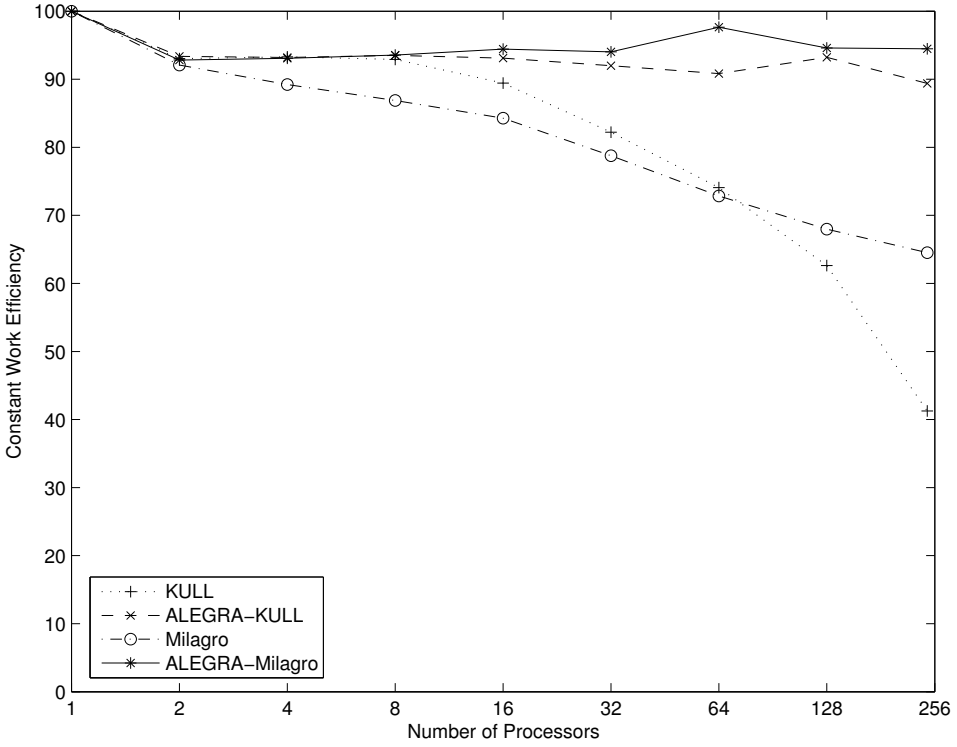

Fig. 2. Parallel Efficiency, ϵ_{CW}, for the constant work hot box problem.

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

3.1.2 Constant Work Scaling

Figure 2 shows the constant work efficiency of the four algorithms with four million particles. The constant work parallel efficiency is

$$\epsilon_{\mathrm{CW}} = \frac{t_1}{nt_n}, \tag{1}$$

where n is the number of processors, t_1 is the serial run time, and t_n is the run time of the n processor run. The mesh was decomposed into roughly cubic chunks. The original KULL and Milagro algorithms do not scale well, each for a different reason. The KULL algorithm has a serialized communication pattern, as discussed in Section 2.1. In the Milagro algorithm the master processor spends a lot of time checking for messages from all other processors. This leads to a significant load imbalance as the number of processors is increased. The ALEGRA-KULL algorithm scales very well, but begins to suffer from the multiple global communications within each time step. The ALEGRA-Milagro algorithm scales nearly perfectly to 244 processors. In fact, the biggest performance decrease happened between one and two processors and is more a result of machine architecture than of the algorithm behavior.

3.2 A Vacuum Box

This is the same mesh as the hot box problem, but there is no material in the mesh. It is initially cold, with a uniform isotropic source of $T = 3.5e5$ K on one side. Initially the load balance is not good, but by the end of the time step, the box is uniformly filled with particles. Only one time step of $\Delta t = 3 \cdot 10^{-9}$ sec was run.

Figure 3 shows the efficiency results from a constant work study using one million particles. The efficiency actually improves for this problem as the number of processors increases. As the number of processors is increased, particles traverse the mesh on each processor more quickly, and must be transfered to other processors more often. Particles get transfered to more processors sooner, so the work can be more evenly shared. In the extreme case of two processors and the KULL and ALEGRA-KULL algorithms, processor one must move all the particles from the source boundary to the sub domain boundary while processor two waits to receive some particles. Once processor one is finished, it sends the particles to processor two, which then moves the particles to completion while processor one sits idle. The Milagro and ALEGRA-Milagro algorithms do not suffer from this problem as severely because they exchange particle buffers more frequently. Similar things have been noticed before in discrete ordinates simulations, where decomposing a three dimensional mesh into two dimensional columns can dramatically improve performance[7] because information can be exchanged more often allowing otherwise idle processors to do work.

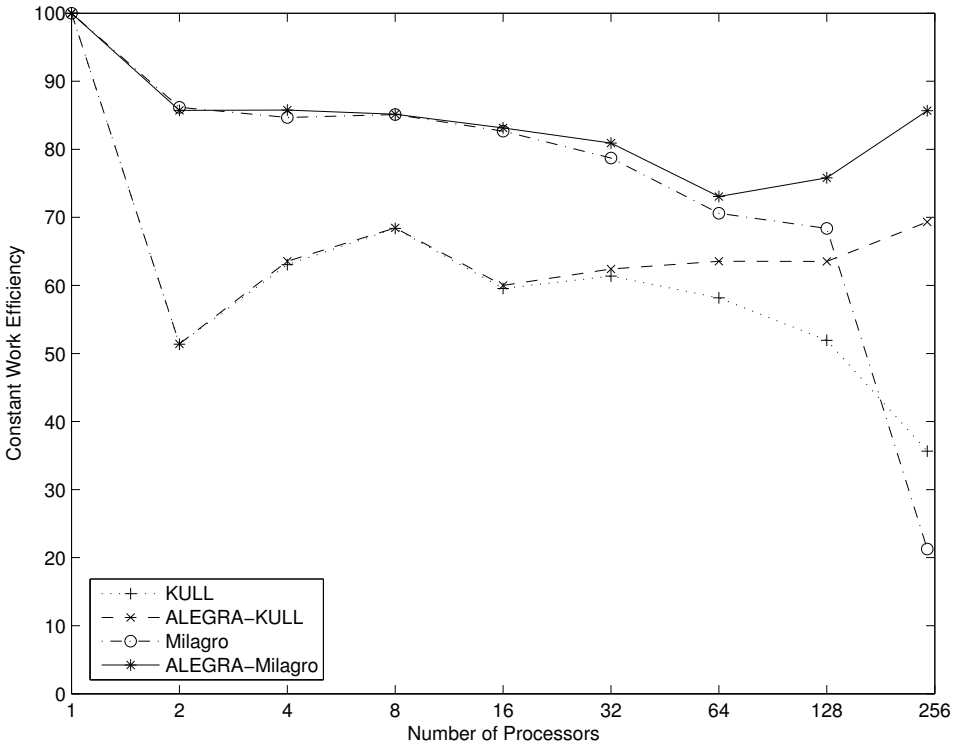

Fig. 3. Parallel Efficiency, ϵ_{CW}, for the constant work vacuum box problem.

4 Conclusions

Two production algorithms for asynchronous parallel Implicit Monte Carlo radiation transport were analyzed and improved. The improved version of the Milagro algorithm, the ALEGRA-Milagro algorithm, performed the best by scaling nearly perfectly out to 244 processors on a Linux cluster. The improvements were to check for messages less frequently and to use a scalable, nonblocking version of the standard reduce and broadcast functions. It is critical not to have one processor do more work than the others, even if it seems like it is a trivial amount of work, such as checking for incoming messages. The algorithms that used blocking communication do not perform well due to unnecessary contention for processor time.

References

[1] N. A. Gentile, N. Keen, J. Rathkopf, The KULL IMC package, Tech. Rep. UCRL-JC-132743, Lawrence Livermore National Laboratory, Livermore, CA (1998).

[2] T. A. Brunner, T. A. Mehlhorn, A user's guide to radiation transport in ALEGRA-HEDP, version 4.6, Tech. Rep. SAND2004-5799, Sandia National Laboratories, Albuquerque, NM (Nov. 2004).

[3] J. A. Fleck, Jr., J. D. Cummings, An implicit monte carlo scheme for calculating time and frequency dependent nonlinear radiation transport, Journal of Computational Physics 8 (1971) 313–342.

[4] T. J. Urbatsch, T. M. Evans, Milagro version 2, an implicit Monte Carlo code for thermal radiative transfer: Capabilities, development, and usage, Tech. Rep. LA-14195-MS, Los Alamos National Laboratory (Jan. 2005).

[5] Derrick Coetzee, Binary tree, Wikipedia article.
URL `http://en.wikipedia.org/wiki/Binary_tree`

[6] Rolf Rabenseifner, Optimization of Collective Reduction Operations, in: M. Bubak, G. D. v. Albada, P. M. A. Sloot, J. J. Dongarra (Eds.), International Conference on Computational Science, Krakow, Poland, Vol. 3036 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 1–9.

[7] R. Baker, K. Koch, An S_n algorithm for the massively parallel CM-200 computer, Nuclear Science and Engineering 128 (3) (1998) 312–320.

Algorithm 4 ALEGRA-Milagro

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