Final Report

Development of Parallel Computing Framework to Enhance Radiation Transport Code Capabilities for Rare Isotope Beam Facility Design

Funded by DOE under **DE-FG02-07ER41473**

M. A. Kostin (PI)[1], N. V. Mokhov[2] and K. Niita[3]

[1]Facility for Rare Isotope Beams, Michigan State University, 640 South Shaw Lane, East Lansing, MI 48824, USA

[2]Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

[3]Research Organization for Information Science and Technology, Tokai-mura, Naka-gun, Ibaraki-ken 319-1106, Japan

Funding received in k$

	FY 07/08
Michigan State University	**98**
Total	**98**

Abstract

A parallel computing framework has been developed to use with general-purpose radiation transport codes. The framework was implemented as a C++ module that uses MPI for message passing. It is intended to be used with older radiation transport codes implemented in Fortran77, Fortran 90 or C. The module is significantly independent of radiation transport codes it can be used with, and is connected to the codes by means of a number of interface functions. The framework was developed and tested in conjunction with the MARS15 code. It is possible to use it with other codes such as PHITS, FLUKA and MCNP after certain adjustments. Besides the parallel computing functionality, the framework offers a checkpoint facility that allows restarting calculations with a saved checkpoint file. The checkpoint facility can be used in single process calculations as well as in the parallel regime. The framework corrects some of the known problems with the scheduling and load balancing found in the original implementations of the parallel computing functionality in MARS15 and PHITS. The framework can be used efficiently on homogeneous systems and networks of workstations, where the interference from the other users is possible.

Existing Parallel Computing Capabilities of Radiation Transport Codes Used for RIA R&D and FRIB

It is a conventional knowledge that calculations performed for shielding applications can be time consuming. A number of reasons may be responsible for that, for example the amount of shielding material in models, and comprehensive modeling of the physics processes. Various biasing and variance reduction techniques have been developed over the time to deal with this problem. Another way to mitigate the problem is to use parallel computing capabilities which become available with rapidly growing computational resources. Most of radiation transport codes today are capable of performing calculations on parallel computers.

This project was developed for needs of design of RIA and FRIB project. Two main radiation transport codes that we used are MARS15 [1]-[3] and PHITS [4].

In the original MARS15 parallel framework, described in reference [5], only a linear topology of processes implemented, where one *master* process receives data consequently from a number of *workers*. When the data are sent is determined by the *master* process only. Each *worker* process does not have any predetermined number on maximum number of events it needs to process. This configuration allows a great flexibility when the system is used on a network of workstations where the calculation speed is different on each workstation. Despite of this flexibility, a problem arises sometimes at the end of calculations when a total requested number of events (N_{total}) is almost reached. Since each *worker* does not have a pre-set limit on the number of events it is allowed to calculate, it is possible that the number of events a *worker* processed since the last data exchange is too great. It this case this local data cannot be used anymore. In this case this process is dropped out, and the rest of calculations is finished using a smaller number of *worker* processed. And if the number of dropped processed is large, the rest of the calculations must be carried out by only a small fraction of processes. In some instances we found that this scheduling artifact leads to a significant increase in the computation time. An illustration of this problem is shown in Figure 1.

The scheduling in the PHITS code is performed in a different way. All calculations are carried out in batches. Each process has a fixed number of events to process. When that number of events is processed, the data are sent to the *master* process. The only function of the *master* process is the bookkeeping, it does not run calculations. This is illustrated in Figure 2 which shows a load of Linux cluster. This *self-scheduling* approach eliminates the *load balancing* problem like the one described above that persists in the MARS15. But the drawback of this approach is inability to use all resource efficiently on a network of workstations where each process can have a different execution speed. In this case, since the data is received from the *worker* processes consequently (linear topology), the *master* process has to wait for the slowest process, and effectively the execution speed of all the processed becomes that of the slowest one.

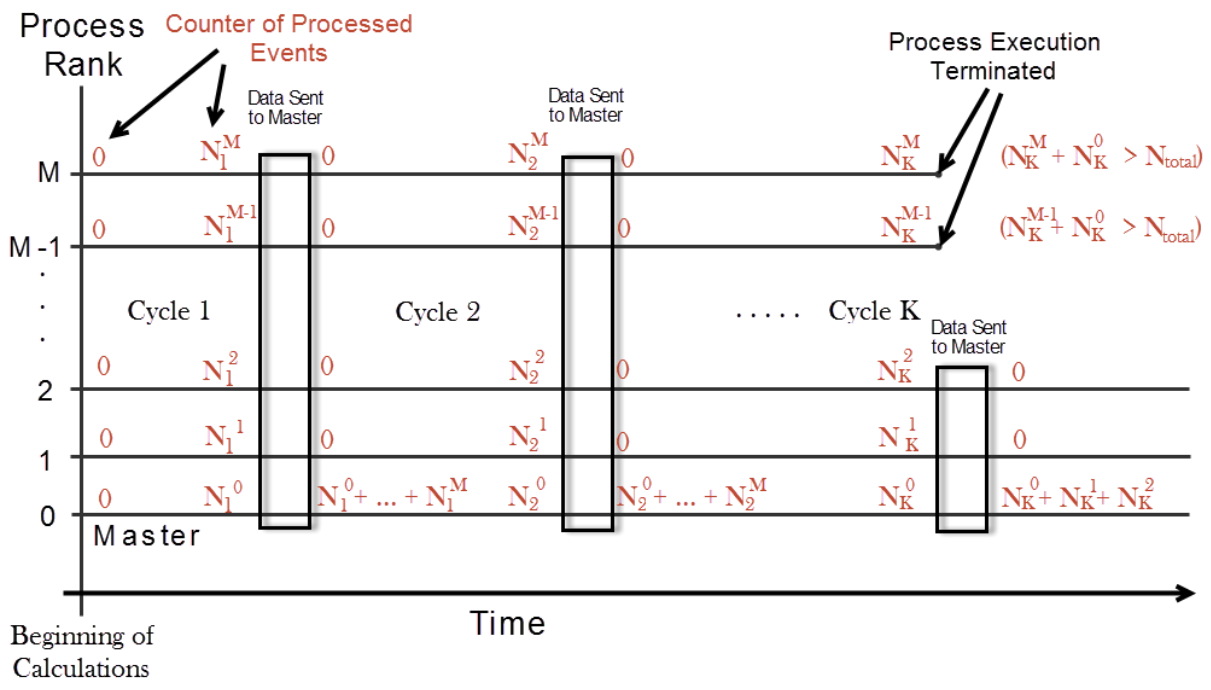


Figure 1 Schematic of data exchanges that illustrate the problem of worker processes terminated if the number of calculated events exceeds a limit. N_i^k is the counter of processed events in each process where *i* is the index of exchange cycle and *k* is the rank of the process (the master process is assumed to have the rank 0). The counter of processed events is reset in the workers after each data exchange. N_{total} is total number of events requested by the user to complete the calculations.

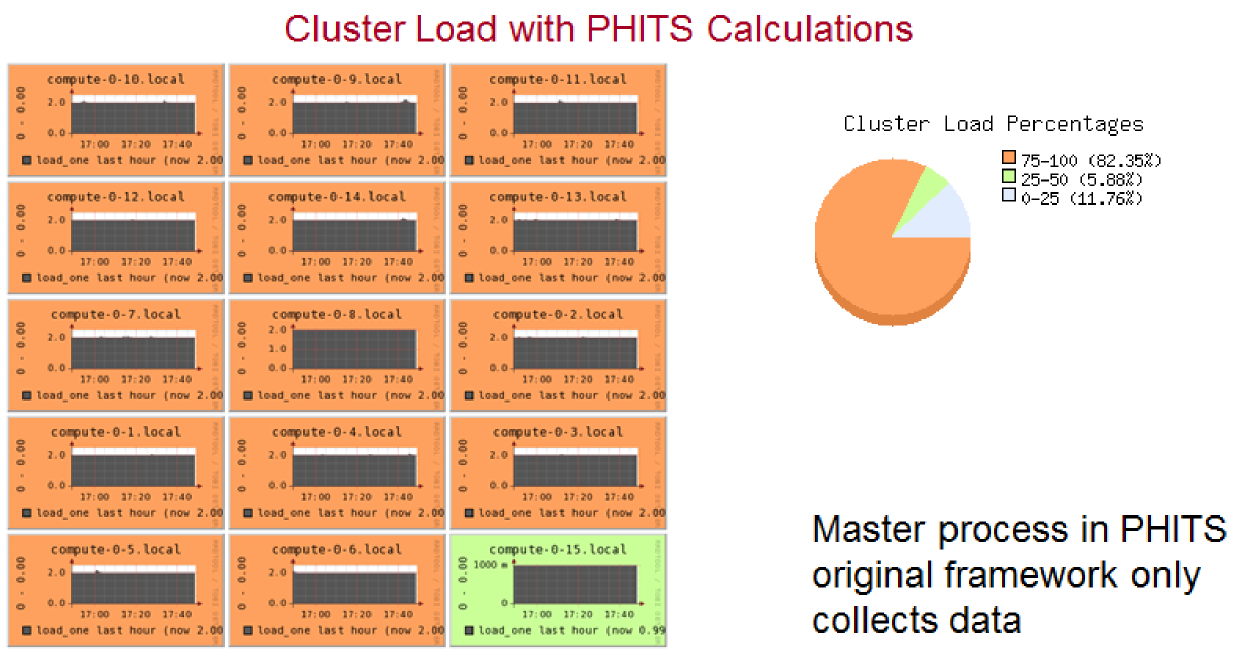


Figure 2 Chart that shows a load of a Linux cluster with PHITS calculations. The master process in PHITS original framework only collects data.

Another radiation transport code FLUKA [6] (not used in RIA R&D or FRIB design) does not offer a parallel computing at all. FLUKA users need to run a number of single-CPU calculations and then to post-process and merge the results of these calculations.

The code MCNP/MCNPX [7] uses a framework by concept similar to that of PHITS. It employs *self-scheduling* approach.

Out of all these codes only MCNP offers a *checkpoint facility* which allows restarting calculations from a *checkpoint*. Whereas MARS and PHITS save some intermediate results during calculations, these cannot be used to restart the calculations if those are prematurely terminated.

All the discussed radiation transport codes use a linear topology of processes in which the master process receives the data from the workers consequently one by one. The communications between the processes become a bottleneck if a number of processes is large (the effect is expected to be significant at a few thousands processes) and the efficiency of the parallel computing degrades. This could be mitigated with more advanced topologies (for example with a tree-like topology), but it was not yet implemented in any of these codes.

Description of New Framework

Implementation Language

The original idea was to make this new framework as much independent of radiation transport codes it could be used with as possible. Most of these codes were developed over long periods of time, and were implemented in FORTRAN77 for the most part. FORTRAN77 data structures must be handled explicitly in a parallel code which negates the whole idea of the framework independence from any particular radiation transport code. For example, the data handling in the original MARS15 framework needed to pack/unpack data or create *derivative types* require code that does it by calling MPI functions for each MARS15 data array explicitly. And any time the list of arrays is changed, or dimensions of the arrays are changed, these functions must be modified in order to accommodate the recent update. This is inconvenient and error prone. An alternative way for the framework implementation is to segregate the parallel computing functionality into a software module written in a language that supports the mechanism of pointers. This way, any action with data structures such as packing and unpacking buffers or creating *derived types* can be done in a single loop over the pointers. This concept is illustrated in Figure 3 and Figure 4. C++ was chosen among several considered candidates because there are freely distributed compilers for C++, it can be mixed with FORTRAN77, and the language features allows a good code structure for the framework. FORTRAN90/95 would be another appropriate choice, but there were not freely available compilers when the project was started. A commonly used in the scientific community compiler *gcc* [8] did not support FORTRAN90/95 at the time.

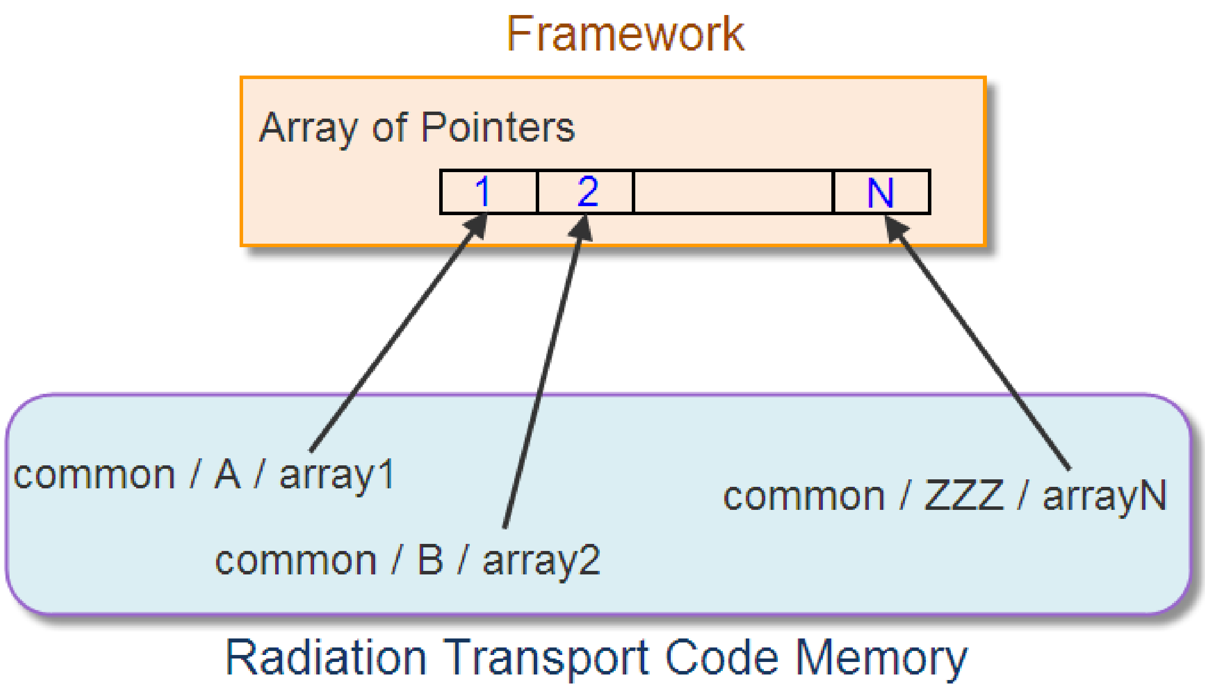


Figure 3 Schematics of the data array arrangement in a FORTRAN77 legacy code with associated array of pointers in the new framework. Any data manipulation in the framework can be performed as a loop over the pointer instead of doing it for each array separately and explicitly.

```
position = 0

call MPI_PACK(APP_1, block_length_1,  MPI_type_1,
                    buffer, buffer_size, position,
                    MPI_COMM_WORLD, error_code)

call MPI_PACK(APP_2, block_length_2,  MPI_type_2,
                    buffer, buffer_size, position,
                    MPI_COMM_WORLD, error_code)

.....

call MPI_PACK(APP_N, block_length_N,  MPI_type_N,
                    buffer, buffer_size, position,
                    MPI_COMM_WORLD, error_code)
```

```
position = 0

do i=1, N
     call MPI_PACK(APP(i), block_length(i),  MPI_type(i),
                    buffer, buffer_size, position,
                    MPI_COMM_WORLD, error_code)
end do
```

Figure 4 Two examples of pseudo-code. The part of the left shows an example of packing MARS15 data arrays into MPI buffer. Any time arrays added or removed, or their order is changed, or array sizes are changed, one must explicitly make changes in this code. The code on the right does the same packing of the data into a buffer but uses the pointers to the arrays (not available in standard FORTRAN77). The entire structure on the left is now collapsed into a single loop. The order in which the arrays will appear in the buffer and their sizes are specified at initialization.

Middleware

A comprehensive study of the available middleware was performed during our previous research work [5]. To summarize, a number of candidates were considered: MPI [9],[10]; CORBA [11]; sockets and PVM [12]. CORBA provides extensive functionality and is appropriate for distributed computing applications, but it is also relatively hard to use and its communication overhead may be significant. Sockets involve little overhead for communications but much of the necessary high-level functionality is absent. PVM has a long and successful history. The MCNP collaboration [7] however recommended MPI over PVM, having had substantial experience with both the packages. Also, given our experience, we could conclude that MPI is the best choice for this particular type of application. Besides being considered as a standard for programming parallel systems, the MPI functionality seems to match data structures and the code structure of the considered radiation transport codes quite well. It can use low level protocols such as TCP/IP which imposes little communication overhead. MPI is available for machines of all architectures – massive parallel clusters, commodity clusters and networks of workstations. Freely distributed implementations of MPI such as Open MPI [13] and MPICH [14] are available together with a number of commercial implementations with performance optimized by the vendors for their systems.

Code Architecture

The current version of the framework offers only a linear topology of processes. There is only one group of the processes with one *master* process and an arbitrary number of *worker* processes. Each process replicates the entire geometry and uses the same settings of a studied system. The parallelization is job-based, i.e. processes are running independently with different initial seeds for the random number generator. The exchange of results between the master and workers is initiated by the master according to the scheduling algorithm which will be described later. Besides performing the control task, the master also runs event histories. This is different from the framework originally implemented in the PHITS code. This feature is important for systems with a small number of processors.

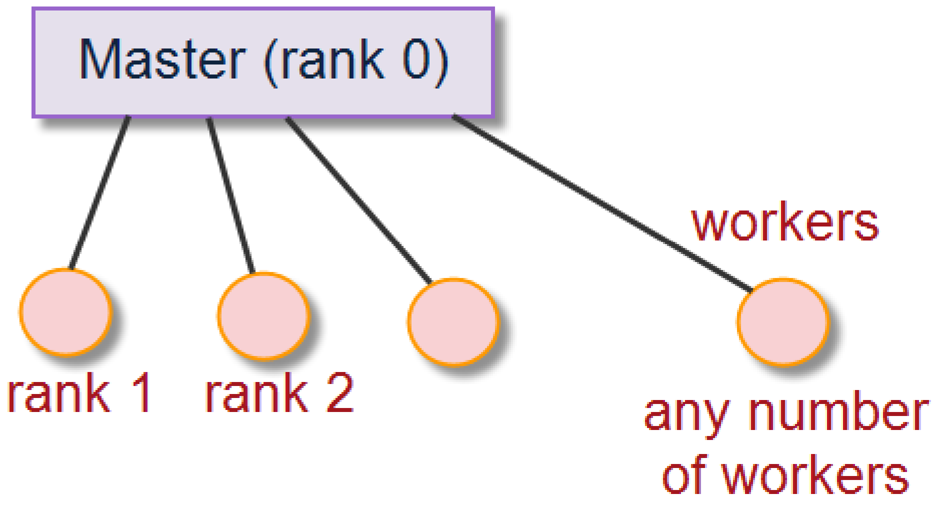


Figure 5 'Linear' topology of processes. There is one *master* process and an arbitrary number of *workers*.

It is quite obvious that each event in a radiation transport code is independent from other events. This differs from calculations on meshes where each new round does depend on results of previous iterations. This feature makes the processes in the framework loosely-coupled, and allows information exchange sessions (*rendezvous*) as often or rare as we choose without performance penalties. This also results in a better scalability compared to tightly-coupled calculations on meshes.

During the information exchange sessions the worker processes are inquired consecutively according to their ranks. In order to avoid possible interference with the running event histories, the workers probe the signals from the master in an *asynchronous mode*. A *worker* starts processing the next event if no signal from the *master* is received at the time of the probing. The *master* can also send out a signal to stop calculations if the required number of events has been reached. All the intermediate information is transferred from a *worker* to the *master* in a single round. The *worker* will pack and send a buffer with service information containing the number of processed events and the seeds for the random number generator (needed for the *checkpoint* facility), with the contents of arrays, and with the contents of data containers.

The data containers are another part of the framework. They were developed to replace the deprecated single-precision HBOOK histogram package [15] used in the MARS15 code. There are no limitations on the buffer size because it is dynamically allocated in the framework. All communications are performed in the MPI *standard mode* except for the signal probing mentioned above. The order of all the corresponding 'send' and 'receive' function calls is carefully matched in order to avoid a *deadlock*.

We were previously experimenting with various methods of information exchange between the processes [5]. Among the methods considered there were sending the data with MPI functions for array elements positioned contiguously in the memory, sending with the MPI functions for packing and unpacking buffers, and sending data with *derived types*. The data arrays in the radiation transport codes are generally positioned in a number of common blocks, therefore they are not in contiguous memory. The use of the first method would result in excessive communications, since each array should be sent separately. The other two methods can be used for data located in arbitrary places. The sending data with *derived types* involves some overhead to build such types, but this happens only once before the calculations are started. This is an appropriate method in case when the communications occur frequently. In our case, however, the processes are loosely coupled with information exchange sessions to be rare. In addition to that, the *checkpoint* facility requires functionality to pack and unpack the data to and from a buffer anyway. This is the same functionality as required for the second method. Therefore the second method is used in the framework.

Scheduling and Load Balancing

Scheduling is an important issue that is directly related to performance, scaling and fault tolerance. Since the communications are quite expensive for the current generation of commodity clusters, an obviously simple approach to the design of the framework would be reducing the amount of communications as much as possible. In the most extreme case this means collecting the information from the worker processes only once at the end of calculations. Moreover, the performance can only be good if the communication time, T_m (the subscript 'm' stands for 'messages'), is much smaller than the computation time, T_c. On the other hand, the *rendezvous* must be frequent enough in order to provide some fault tolerance – it is important to be able to restart the computations from the last *checkpoint* if a system failure occurs. The frequent *rendezvous* are also useful to obtain most recent information about the calculation speed of each process in order to adjust the load of the process and to achieve a better load balancing.

The scheduler in the framework compromises between these two issues. It decides when to suspend the computation and to start a *rendezvous*. The decision is based on the knowledge of an estimated time needed for the *rendezvous* and time needed to process one event, T_1. The *master* process may wait for a response from a *worker* for a long time during *rendezvous* in case of long histories. This waiting time may significantly prolong the *rendezvous*. For the framework to be effective, the time between the *rendezvous* has to be significantly longer than $max\{T_m , T_1\}$.

In the first implementation of the parallel computing functionality in the MARS15 code [5], a *worker* process is terminated at a *rendezvous* if the number of locally processed histories combined with the number of events already collected by the *master* is in excess of the total number of requested events. This would most probably to happen when the jobs are close to their end. Time to the next *rendezvous* has to be shortened to avoid that and to use the resources more effectively. In the opposite case, the *master* or a smaller group of the remaining processes will have to process the rest of histories by themselves. This may lead to a sizeable computation time increase if the number of terminated processes is large and the balance of events is still significant. This problem was illustrated in section Existing Parallel Computing Capabilities of Radiation Transport Codes Used for RIA R&D and FRIB. An attempt was made to take into account this effect by adjusting the two previous time conditions:

$$T = min\{100 \times max\{T_1 , T_m \},0.8 \times T_{end} , 1\ hour\}, \quad (1)$$

where T is the time to the next *rendezvous*, T_{end} is estimated time to the end of calculations taking into account the speed of each process. The requirement of *1 hour* is based on a human factor. The exchange time must not exceed a sizeable fraction of a working day. This is to let people deal with potential problems in their models. The numerical factors 100 and 0.8 are parameters that can easily be changed. We have found however after several years of use that the situation when the master finishes the calculations itself still occurs. Further improvements are required despite the fact that the above scheduling algorithm offers a great load balancing.

The scheduler in the parallel computing implementation in the PHITS code does not have this problem at all. All calculations are performed in batches where each worker gets a fixed number of events to process, and the situation when the master must process an excessive number of histories is not possible. Each process ‘knows’ ahead of time how many events it needs to process, and stops when this number is reached. This is an example of self-scheduling also implemented in the MCNP code. This approach works well on homogeneous systems, but is quite inefficient on network of workstations and systems where each processor may have different performance or interference from the other users is possible.

The new framework implements features from both scheduling mechanisms. The scheduling still works as defined in (1), but each process now also knows the maximum number of events it is allowed to process. This mixed scheduling still provides a great flexibility in load balancing that allows the framework to be used on a network of arbitrary workstations, and also deals with the scheduling problems described above.

Figure 6 illustrates a communication time scheme. The first *rendezvous* occurs in a fixed time period of 10 s. The fixed time period is needed to calculate trial values for T_1, T_m and T_{end}. These values are calculated at the end of each rendezvous later on, with T_1 calculated for each process. Schematics of the data exchange between a *worker* and the *master* process is shown in Figure 7 and is explained above. The buffer used to send data between the processes is exhibited in Figure 8. It normally includes three parts: process header with a number of processed events and seeds for the random number generator; contents of data arrays (these arrays are typically placed in the common blocks); and contents of the data container (which can be used to replace the HBOOK histograms in MARS15). The same buffer is also used by the *checkpoint* facility which will be described later. In the latter case, the buffer will include the number of processes in the job. There is also space reserved for data structure that would describe a topology of the processes. This space is not currently used because only one topology has been implemented, and these data are not needed at this moment.

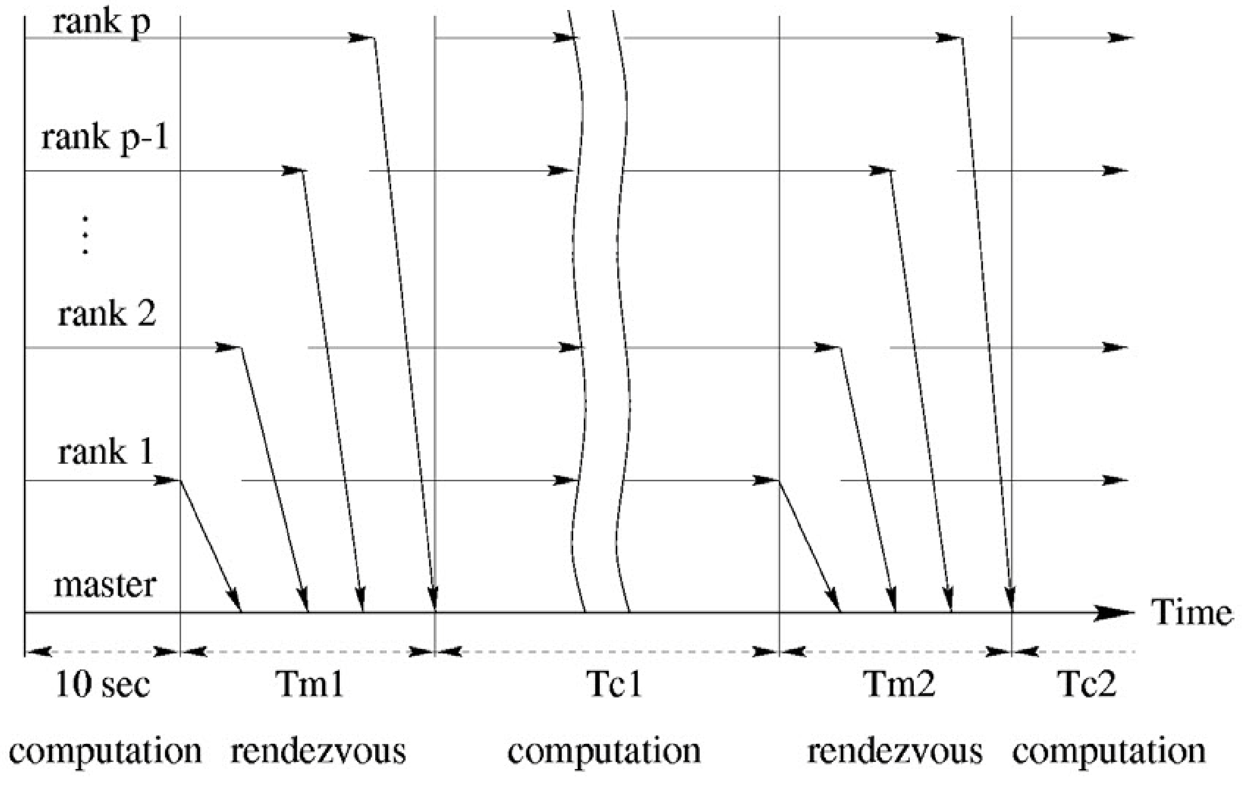
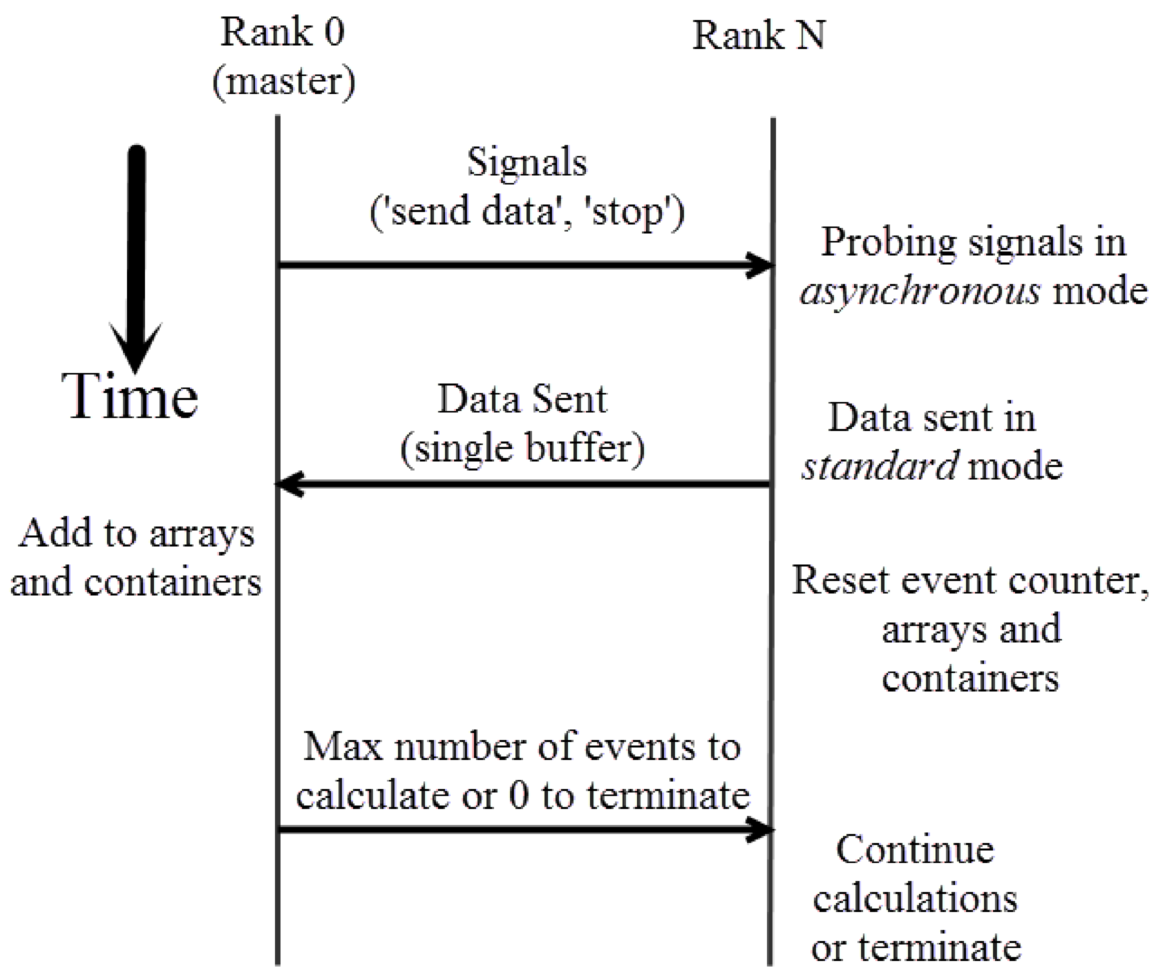


Figure 6 Time scheme of communications between the processes.



Rank 0 (master)
Rank N
Time
Signals ('send data', 'stop')
Probing signals in *asynchronous* mode
Data Sent (single buffer)
Data sent in *standard* mode
Add to arrays and containers
Reset event counter, arrays and containers
Max number of events to calculate or 0 to terminate
Continue calculations or terminate

Figure 7 Schematics of the data exchange between the *master* process and a *worker* process.

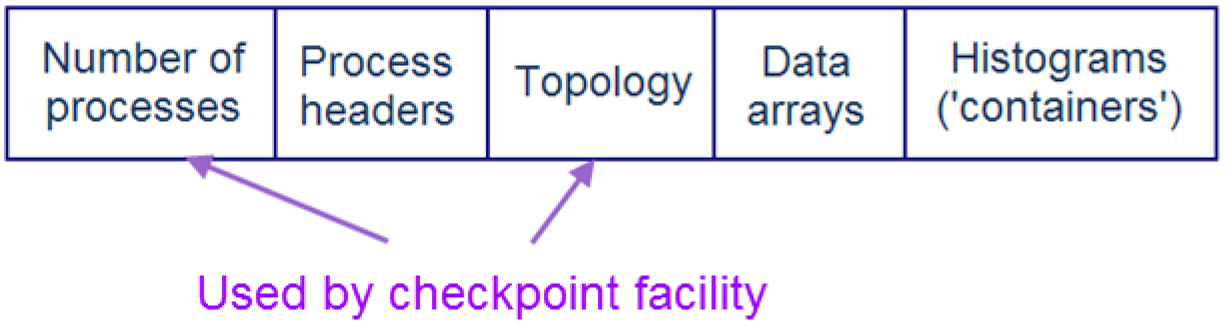


Figure 8 The structure of the buffer used to send data from a *worker* process to the *master* process. The same buffer is also used by the *checkpoint* facility. In this case the number of processes in a calculation job and the topology of processes is included. The space for the topology is currently reserved but not used because only one topology has been implemented in the framework.

Checkpoint Facility

It is not unusual that a computational job terminates prematurely and has to be restarted from the beginning. This may occur due to a variety of reasons such as power outage, problems with batch systems, etc. This may cause a significant loss of time since the computational jobs are time consuming and may take days to complete. Moreover, the radiation transport codes are under constant development. The recent code modifications sometimes make the codes unstable which may results in job terminations. Regular debugging techniques are not appropriate in this case because it may take many days to reach a problematic event.

These problems can be mitigated with a *checkpoint* facility that was developed as a part of the framework. The facility allows restarting calculations from a recent *checkpoint* and not from the very beginning. A *checkpoint* is represented with a file with all the information necessary to restart calculations. The information is saved into two files alternately, so that if the system fails during the framework saving the *checkpoint* file, then the previous file would still be available. The *checkpoint* facility works both in the parallel and single process regimes. The files are saved after each information exchange session in the case of parallel computation. In the single process regime, however, the user defines how often he is willing to use this functionality. The framework was designed to allow merging several files into a single one, thus allowing combining results from several calculations as long as the machine word formats are the same. This option is not completely functional at this time and will require a further development. Currently the facility imposes no limitations in what regime a *checkpoint* file can be used regardless of how it was created. For example, it is possible to obtain a *checkpoint* file in the parallel computation regime, and to use it to restart calculations in the single process regime.

It is appropriate to say now that the existence of the *checkpoint* facility changes the strategy of how the calculations can be performed. The number of events needed to obtain statistically significant results is rarely known a priori. It is customary to estimate the required number of events using short test calculations. If these estimates are not accurate enough, the required statistical significance is not achieved, and the calculations must be restarted from the very beginning after appropriate corrections are made. The necessity to restart the calculations from the beginning may cause significant delay in obtaining results is some instances. With the *checkpoint* facility, however, it is not an issuer anymore, because the statistics can always be improved 'on-the-fly'.

Handling FORTRAN Arrays

The part of the framework that works with FORTRAN arrays through the pointers is hidden from the FORTRAN users. The only function that is available to the FORTRAN users is registration of arrays. These arrays must have a global scope (be in common blocks), otherwise the pointers to these arrays will not be valid. All other functions are used by the framework only (although they can be accessed from a C++ code if added to the radiation transport codes). This part of the framework is organized as a module. There is a single pointer manager that is built as a global object that cannot be copied. The manager keeps track of all the registered arrays. The manager recognizes the following FORTRAN types: INTEGER, REAL, DOUBLE PRECISION. The functions that can be used from the inside of the module include the registration of an array; reset the array contents to zero; read the contents from a buffer and save it into the array; read the content from a buffer and add it to the existing contents; write the contents to the buffer; and calculate the size of the array and memory size needed for this array.

Data Containers/Histograms

The data containers module was originally designed to replace the deprecated third party library HBOOK [15] in MARS15. The HBOOK is single precision software that is no longer officially supported. The data containers (or histograms) are organized as a module with a single global manager object. The users can book the containers and access their functionality via interface functions. The containers are all in double precision (although other types can be added), and are available as 1-D (one index), 2-D (two indexes) and 3-D (three indexes, for example X, Y and Z). There is also one-dimensional container with variable bin size. Each bin in a container keeps both the mean value and variance of a stored value (for example particle flux), and thus can also be used as a bookkeeping tool for the radiation transport codes. For example, two arrays in MARS15 that keep the mean values and variances of the star density can be replaced with a single container. In principle, all the bookkeeping of most the values that the radiation transport codes collect and keep in the common blocks can be outsources to the data container module in a uniform manner. The containers are stored in the dynamic memory therefore there is no hardcoded limit on the number and the size of the data arrays that can be stored. The data containers can be saved in a separate file which can be further converted into an HBOOK file or a ROOT file or any other format. Converter programs are outside of the scope of this project, and are needed to be developed separately.

Each data container/histogram is identified with two parts: a numerical identifier and a character identifier. For the histogram identifier to be valid, it must have a positive numerical part and any (empty or not empty) character part, or the numerical part equal to zero and the character part which is not empty. The numerical part cannot be negative. It would normally be sufficient to keep only the character identifier. The numerical part was introduced for consistency with the HBOOK package in order to minimize code changes where HBOOK is used. The character part of the identifier was included for compatibility with the ROOT package.

The following functionality is currently supported. The FORTRAN users can check if a histogram with a specified identifier exists, book or delete a histogram, access histogram attributes, fill histogram, reset the contents to zero. There is also additional functionality that is available to the framework exclusively: read or write a whole histogram or contents only from/to a buffer, read the contents from a buffer and add to the existing contents, calculate the memory size needed to store the histogram.

Performance Test

A performance test was conducted on the NSCL's DOEHPC Linux cluster. The cluster consists of 16 dual CPU nodes. Each CPU is a 2.6 GHz AMD Opteron™ 252 processor with 1024 kB of cashe memory. The nodes are equipped with 4 GB of memory and connected via a 1 Gb/s network. The MPICH2 implementation of MPI was used for the message passing. The test jobs were managed by the TORQUE Resource Manager [17]. The test was conducted using a MARS15 model consisted of one hundred geometry zones. The performance test measured the speedup as a function of the number of used processors. The test demonstrated the speedup close to linear on a small number of processes (up to

32) (see Figure 9). The test could not be conducted on a larger number of processors due to the limited resources.

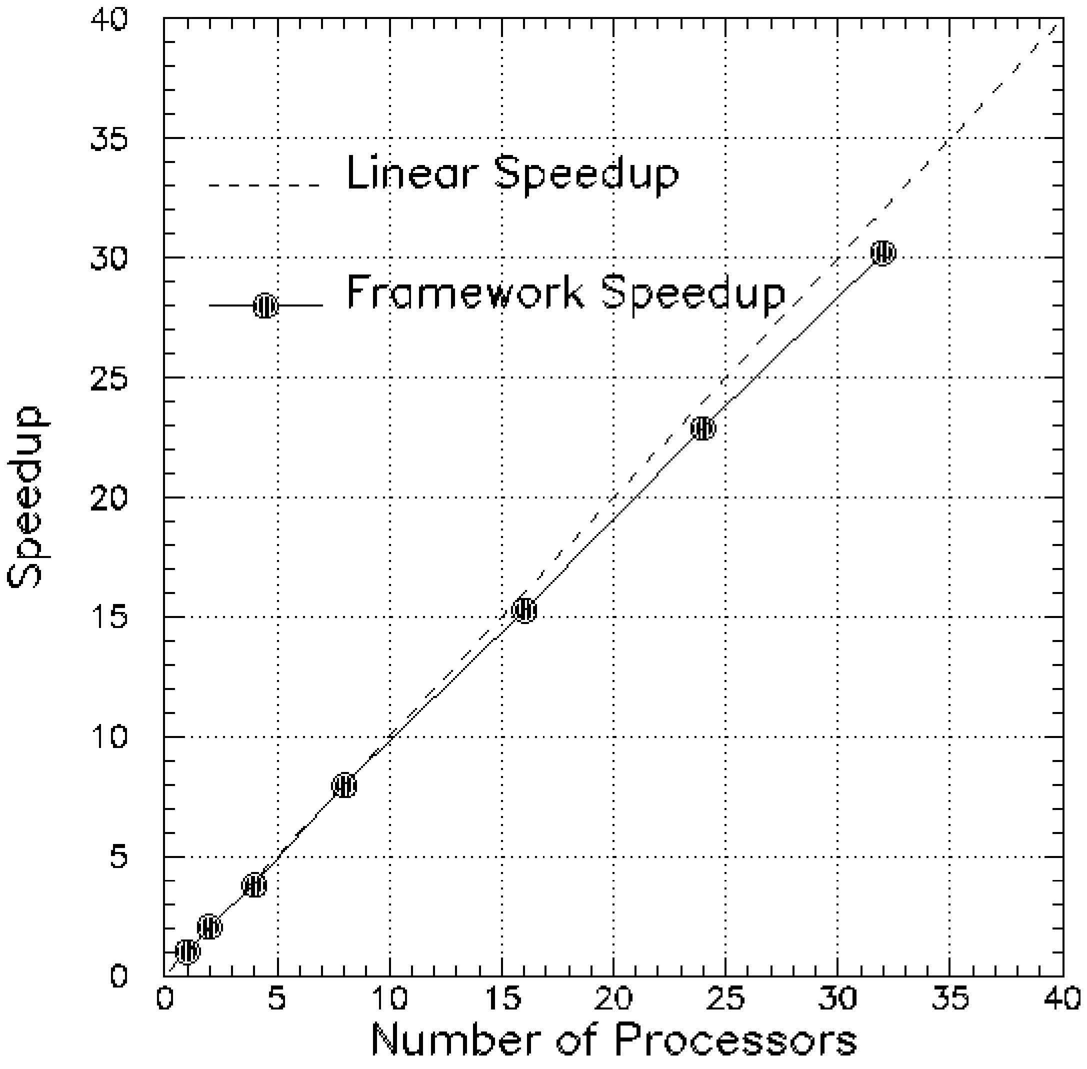


Figure 9 Speedup of the framework in the tested configuration.

Framework Limitations

The framework is meant to be used in situations in which only the *master* process generates the output of results. The framework must be used with care in other scenarios. For instance, the user may want to generate a list of particles crossing a surface. In this case the output files with the particle lists generated by each process must be named uniquely to avoid corruption of the output data. The framework does not provide any facility in which such lists can be redirected to and saved by the *master* process.

The framework also assumes an unlimited input for all the nodes and processes, *i.e*. each process on each node must be able to read input files required by radiation transport codes. Although it is technically possible to implement a framework where only the *master* process reads in the input files and distribute the parameters to the *worker* processes, this configuration is considered outside of the scope of this work.

The authors realize that the performance tests were carried out using limited available resources. The framework speedup with currently implemented linear topology of processes may significantly degrade or even plateau out on massive parallel systems when the number of used processes is very large. This speedup degradation due to increased communication time can be significantly mitigated with considered but not yet implemented multilevel ('tree-like') topology. In the 'tree-like' topology, the *master* process (level 0) gathers results from a limited number of level 1 processes, which also act as masters for their own groups of processes (level 2), and so on (see Figure 10 for example). The number of levels in principal is not limited. For practical purposes, however, it is most likely that the number of levels can be limited to two: one *master* process of level 1 manages *worker* processes on a single computer node, and sends processed data to the *master* process of level 0 (main *master* process).

The following pieces of software need further development: a program that mixes a number of checkpoint files and produces a single output checkpoint file; a converter of a file with data containers into a HBOOK file; a converter of a file with data containers into a ROOT file. These programs are free standing and are used separately from any of the radiation transport codes.

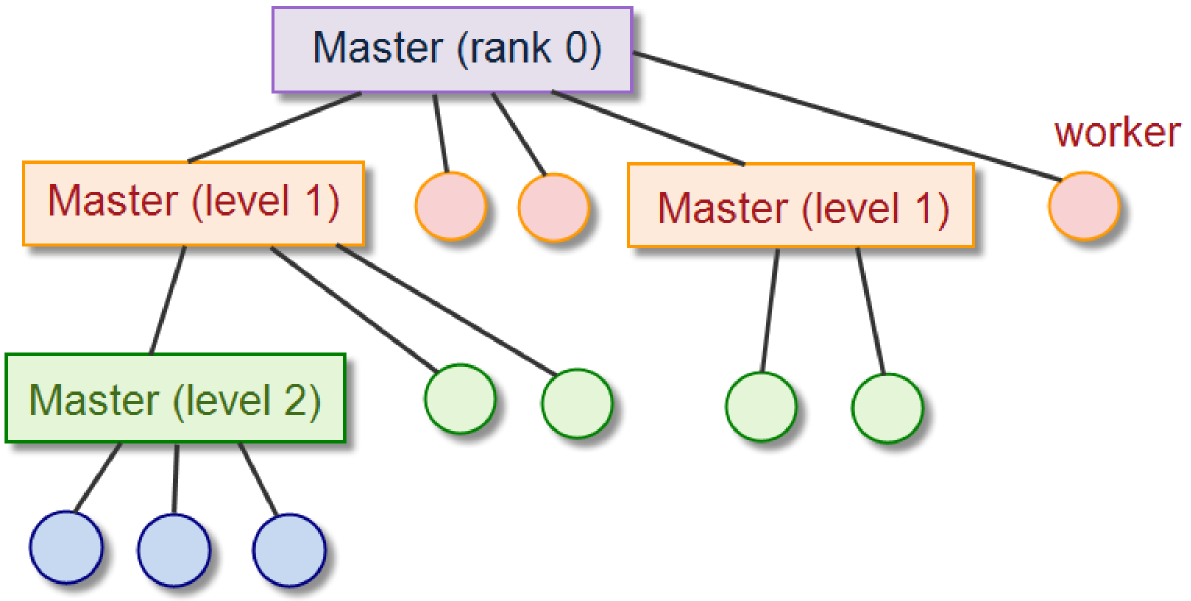


Figure 10 Example of an arbitrary tree-like topology of processes.

Conclusion

The new parallel computing framework was designed, implemented and tested with the MARS15 code. It was tested on the NSCL small commodity cluster and demonstrated a good performance on a small number of processors. The framework offers a good load balancing for each process so that it can be used effectively not only on homogeneous systems but also on networks of workstations. The framework performs better than the original implementations of parallel computing in MARS15 and PHITS. It also offers the checkpoint facility that can be used both in multiple and single process calculations. There is a potential to increase the efficiency of the framework through a new multilevel topology of processes (needs development), although this improvement may only be noticeable for very large parallel systems.

References

[1] N.V. Mokhov, “The MARS code system user's guide”, Fermilab-FN-628, Fermi National Accelerator Laboratory (FNAL) (1995)

[2] N.V. Mokhov and S.I. Striganov, “MARS15 overview”, Proc. Hadronic Shower Simulation Workshop, Fermi National Accelerator Laboratory (FNAL), September, (2006); AIP Conf. Proc., 896, 50-60 (2007)

[3] MARS code system, http://www-ap.fnal.gov/MARS

[4] K. Niita, T. Sato, H. Iwase, H. Nose, H. Nakashima and L. Sihver, "PHITS-a particle and heavy ion transport code system", Radiat. Meas., 41, 1080-1090 (2006).

[5] M.A. Kostin and N.V. Mokhov, “Parallelizing the MARS15 code with MPI for shielding applications”, Radiat. Prot. Dosim., 116, 297-300 (2005); FERMILAB-CONF-04-054-AD, Fermi National Accelerator Laboratory (FNAL) (2004); hep-ph/0405030, http://arxiv.org

[6] G. Battistoni, S. Muraro, P.R. Sala, F. Cerutti, A. Ferrari, S. Roesler, A. Fasso`, J. Ranft, “The FLUKA code: Description and benchamarking”, Proceedings of the Hadronic Shower Simulation Workshop 2006, Fermilab 6--8 September 2006, M. Albrow, R. Raja eds., AIP Conference Proceeding 896, 31-49, (2007)

[7] A General Monte Carlo N-Particle Transport Code (MCNP), http://mcnp-green.lanl.gov

[8] Gnu Compiler Collection, gcc, http://gcc.gnu.org

[9] Message Passing Interface forum, http://www.mpi-forum.org

[10] Message Passing Interface standard, http://www.mcs.anl.gov/mpi

[11] Object Management Group (OMG), http://www.omg.org

[12] Parallel Virtual Machine (PVM), http://www.csm.ornl.gov/pvm/pvm_home.html

[13] Open MPI, http://www.open-mpi.org

[14] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2

[15] HBOOK, http://cernlib.web.cern.ch/cernlib/packlib.html

[16] ROOT, An Object-Oriented Data Analysis Framework, http://root.cern.ch

[17] TORQUE Resource Manager, http://www.clusterresources.com/products/torque-resource-manager.php