

Emulation of PHASTA for PETA Applications Project

Joshua Nasman

Adviser: Chris Carothers

Co-adviser: Kenneth Jansen

May 4, 2008

Abstract

The goal of this project is to be able to predict results on supercomputers before such computers are available. PHASTA has been run on many supercomputers including RPI's CCNI Blue Gene. It has seen excellent scaling results on up to 32,000 processors. This project's goal is to predict how well PHASTA will scale on larger computers than those currently available. This is being done by intercepting the MPI calls so that in the first stage statistics can be obtained about PHASTA and in stage 2 the results can be adjusted according to information about an alternate architecture. This involves making wrappers for both the calls to MPI in FORTRAN and C and getting this to link properly.

1 Introduction

1.1 PHASTA

PHASTA is a parallel flow solver developed at RPI which can solve both compressible and incompressible flow problems. It uses the finite element method to solve a variety of problems using an unstructured grid (Peta apps proposal). One goal of it is to obtain sustained petaflop performance. This emulator is being designed to be a predictive tool on how PHASTA will perform on future supercomputers.

1.2 Motivation

Currently, the tools for emulating programs on supercomputers are limited and often tied to certain architectures. The goal of this project is to be able to simulate the performance of PHASTA on various architectures on systems with larger processor counts than are currently available. By using a lookup table for the communication, it should be possible to emulate various systems by only changing a few parameters. These parameters should ultimately be able to be inputted either as values (which can be obtained from the stage 1 emulator) or mathematical functions (dependent upon processor count and message size) for

future systems where the architecture is unavailable to be tested at this time. At the conclusion of this semester it is likely the emulator will only have been tested on very small systems, but a short term goal will be to get a system like the Blue Gene to emulate a system like TACC (Texas Advanced Computing Center), with a longer goal of being able to emulate systems with a larger processor count such as Blue Waters. In a final version of the emulator as accurate a prediction as possible would like to be obtained. This will likely mean finding a more accurate way to account for cache performance as well as emulating the network in more depth (either by emulating the whole network or using a function of congestion to adjust the times of communication).

This approach is especially ideal when wanting to emulate systems of completely different network types. Many large systems such as TACC are based on one large network (with a switch in the center). The Blue Gene on the other hand has 3 independent networks: a collective operation network, a barrier network, and a point to point network. By simply using lookup tables the type of network used becomes completely irrelevant.

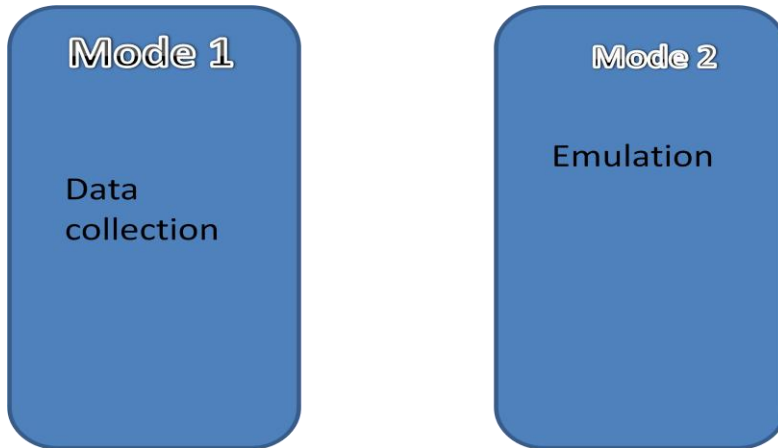
2.1 Phase 1 of project (current semester) - Two phase wrapper:

2.1.1 Overview and goals

While the goal of the project is ultimately to simulate many more cores than available on a system, it was decided that it would be a worthwhile goal to first develop a simulator which can simulate the same number of cores as a different system and make the adjustments based on the network and architecture for a new system. By doing this we could verify the accuracy of the approach before developing an extensive control structure to keep all the emulated objects time in line.

The first stage would simply be a data collection program. This program would measure the amount of time spent in different stages of computation as

well as measure the number of each type of MPI call made. The actual implementation of this will be discussed at a later point in this paper. For certain MPI calls the size of the transfer was also collected (those which were called more than once which were dependent upon this information: isend and allreduce etc).



The second part of data collection was finding the times for the MPI calls used on that architecture on the new architecture. This was done using a test program which simply calls certain MPI calls on the alternate architecture many times and takes an average of the results.

Mode 2 actually does the Emulation. Mode 2 substitutes the newly acquired values for MPI times while PHASTA is being run. It also multiplies the execution time in the computational part of PHASTA by a factor of the CPU times. A notable problem which is soon to be discussed is that of the call MPI_Wait. MPI_Wait waits for all current communication to complete before continuing. It has been discovered that in certain cases this call takes up to half of the time the program spends in MPI. This is not entirely dependent upon the number of processors, nor message sizes, but instead what order things complete in. In order for the timing of MPI to be more accurate, this issue will have to be addressed.

2.1.2 Implementation details of phase 1.

In order to meet the goal of an emulator which will ultimately be portable to other petascale programs it was necessary to find a way to have emulator run outside of the actual PHASTA code itself. It was for this reason that it was decided the best level at which to enter is the MPI level. This emulator is designed in the form of an ‘MPI wrapper’. This effectively means that when MPI calls are made from PHASTA it produces a way to intercept the program without modifying PHASTA. By modifying these MPI calls it is possible to hijack the program.

Conveniently, when MPI was implemented, it was implemented as a combination of two libraries: MPI and PMPI. When making a call from a parallel program it is standard procedure to make the call using the MPI library. A standard MPI call usually looks something like the following (in C).

```
int MPI_Waitall(int a, MPI_Request * b, MPI_Status * c)
{
    return PMPI_Waitall(a, b, c);
}
```

As you can see the implementation is not in the MPI library, but in the PMPI library. This provides an easy way to use your own MPI call, call PMPI at some point in your call and be able to do other things as well (such as timing calls).

This is the approach that was used for stage 1. One of the challenges was to figure out how to have the wrappers MPI calls called instead of the standard ones. Ultimately, it was discovered that MPI uses weak links in their calls. This means that if somewhere a strong link is used on a function definition the new definition will be used. Effectively this means if you link against your own object file or library file before mpi is linked in, the new one will be used.

MPI is implemented in both Fortran and C++ (and both types of calls are used within PHASTA). The wrapper was first tested on an 8 core opteron system. On this system, it was sufficient to include C++ wrappers. We inferred from this that the Fortran calls simply called the C calls in this implementation. When the wrapper was ported over to a Blue Gene an interesting problem was encountered. Only the MPI calls made from C were being captured. The Fortran calls were still doing the MPI operations, but were not calling our wrappers. From this we inferred that the implementation must be in Fortran for the Blue gene MPI calls.

In order to catch the Fortran calls as well, it was necessary to add those calls to the wrapper. The code in the wrapper could still be written in C, so long as the PMPI calls were to the Fortran version. Bob Walkup's Blue gene profiling tool was very useful in figuring out how to do this.

Linking was also a challenge as where the link was done before did not intercept before the fortran mpi calls were bound. This was solved by putting the object file in the phasta/phSolver/phSolver/common/obj/ibm-bg/ directory. This caused the wrapper to be linked in with the common library in PHASTA.

In order to measure the timing two libraries were experimented with. The first one was the rdtsc call. This call uses the system clock to measure time. It seems to work fairly well but it is necessary to include the code yourself. The wrapper later migrated the the MPI_Wtime call. This call uses MPI to measure time in seconds and you must take a difference of two to get the appropriate time. This has worked well with one exception. You cannot call MPI_Wtime before MPI_Initialize. For this reason, the wrapper does not currently correctly account for time before the MPI_Initialize call and ultimately the simulator may go back to using the rdtsc call.

In order to accurately measure the time of various MPI routines a few statistics are needed. These include the number of processors being used by the job and the size of messages sent (or being used in reduce or allreduce). In order to get these sizes, in stage one of the emulator for every MPI call of the type

ISend or allreduce, the number of bytes used is discovered using MPI_Type_Size (or its fortran equivalent). Once this size is obtained, it is sent to a function: either inIsendList or InAllReduceList. These functions check the size given to see if the value is already in the linked list. If it is, 1 is returned, if it is not, 0 is returned and the new value is added to the linked list. By this approach a cohesive list of sizes used in this case is collected by the end of the collection phase of the emulator. To give you an idea of a typical call within the wrapper the MPI_ISend call is shown below.

```
int MPI_Isend(void* a, int b, MPI_Datatype c, int d, int e, MPI_Comm f, MPI_Request *
g)
{
    getTimeAndReset();
    int size;
    MPI_Type_size(c, &size);
    if(!inIsendList(size))
        printf("In Isend: size: %i number: %i \n",size , b);
    retVal = PMPI_Isend(a, b, c, d, e, f, g);
    getMPITime(ISEND);
    return retVal;
}
```

The two notable calls in here which have not been mentioned yet are getTimeAndReset and getMPITime. These calls are responsible for dividing the time for data collection of the emulator as well as adjusting it in the later stage of the emulator. All timings are stored in linked list in this implementation. The getTimeAndReset call stops the timer for computation and adds that time to list of computation times. It also starts the MPI timer. The MPI timer is then on until after the PMPI call is made. When the getMPITime is called you will notice there is an enumerated parameter that is passed in. This parameter is necessary because each type of MPI call has its own linked list of times. This parameter determines which list the time for the last MPI call is added to.

2.1.3 Additional benefits of phase 1

While developing the emulator in this fashion, it was natural at one point for the wrapper to provide statistics for running PHASTA on the system it was currently running on. By running the initial wrapper, it was possible to discover the number of any type of MPI calls made, as well as the time spent in each type of MPI call. This helped us discover that more MPI_AllReduce calls were being made than we were aware, which also helped us understand why it is such a bottleneck on systems like TACC.

2.2 Future work

As this is a long term project, much future work will be done in this area. The initial hope was that by the end of the semester the simulator would be able to simulate TACC using CCNI and vice versa. As time ran out in the semester, and only a smaller test was run, this is the immediate direction this research will take.

Once the testing on this emulator is completed, the project will be expanded to simulate systems with larger processor counts than the machine emulating it. This will involve having nodes be divided into two types: emulation control nodes, and PHASTA nodes.

The PHASTA nodes in this version will no longer hold simply one version of PHASTA per core. PHASTA instances will now have to be switched in and out. This will likely be based on a Round Robin methodology where a message sent to a processor may preemptively force a switch to it.

The emulation control nodes will be the ones responsible for telling PHASTA instances when to switch in and out. They will also be responsible for keeping track of time and emulating the network. Ultimately it will probably be necessary to have one of the cores per node. This could work well on a system like TACC where there are 16 cores per node. The emulator control node could take control of all internode communication and will communicate with other emulator nodes to do communication off-node. This can effectively be done by

dividing the MPI_COMM_WORLD into two comms: and emulator comm and a phasta comm.

4 Ongoing issues

4.1 MPI_Wait call

Currently it has been discovered that the MPI_Wait call ends up taking up to 50% of the MPI time on many PHASTA cases. This is problematic in that there is no easy way to adjust the MPI_Wait time based on an architecture. MPI_Wait waits for all current communication to complete. Currently a bad approximation hack is used to estimate MPI_Wait time (based on the time of MPI_Isend). This will have to be adjusted for the wrapper to give reliable times.

4.2 MPI call time vs. Computational call time

In many cases (seemingly especially the small ones) the MPI time is less by close to an order of magnitude to the computation time. This makes it very difficult to measure the effectiveness of the wrapper when the computational time is estimated using a simple ratio of processor speeds and the MPI call time is what we have focused our energy upon. In order for the emulator to have a time which is close to accurate a heavy emphasis will have to be put on how fast the computational part of PHASTA performs.

5 Data so far

5.1: Data from wrapper collection stage:

So far all test done so far have been on a 4 processor case: PIG4. So far this has been tested to the point of being run on 4 processors on the BlueGene L system. The output from that system has then been inputted into a quad-core opteron system to measure the opteron system's network. The results from the Opteron system were ported back to the Blue Gene and used as input for the final part of the simulation. This procedure is discussed in more details in other parts of this paper.

Stage 1 Results:

```
11111 - reduce
8
22222 - allreduce
4
8
33333 - allgather
4
44444 - isend
3000
15600
1440
7488
600
3120
360
1872
-1 -flag for end of file
```

These are the results from the first stage of the emulator. They are the sizes in bytes of the various data sent through the program.

Data: part 2 (call timing collection)

This is the output from the runs on area51 which measured its communication time. For each of the calls the size of the call is displayed first and the timing (in seconds is displayed after).

```
-10-reduce
8
0.000058
-20-allreduce
```

4
0.000137
8
0.000136
-30-allgather
4
0
-40-isend
3000
0.000154
15600
0.000165
1440
0.000147
7488
0.000160
600
0.000042
3120
0.000153
360
0.000042
1872
0.000149
-50-barrier
0.000109
-60

Data part 3: Simulated results

MPI NAME: ABORT NUMTIMES: 0 TIME: 0.000000

MPI NAME: ALLREDUCE NUMTIMES: 1340 TIME: 0.182243

MPI NAME: REDUCE NUMTIMES: 1 TIME: 0.000058

MPI NAME: REDUCESCATTER NUMTIMES: 0 TIME: 0.000000

MPI NAME: BARRIER NUMTIMES: 7 TIME: 0.000763

MPI NAME: GATHER NUMTIMES: 0 TIME: 0.000000

MPI NAME: ALLGATHER NUMTIMES: 1 TIME: 0.000000

MPI NAME: ALLGATHERV NUMTIMES: 1 TIME: 0.000000

MPI NAME: SEND NUMTIMES: 0 TIME: 0.000000

MPI NAME: RECEIVE NUMTIMES: 0 TIME: 0.000000

MPI NAME: ISEND NUMTIMES: 124 TIME: 0.012318

MPI NAME: IRECEIVE NUMTIMES: 112 TIME: 0.000000

MPI NAME: WAITALL NUMTIMES: 118 TIME: 0.000000

MPI NAME: COMMSIZE NUMTIMES: 1 TIME: 0.000000

MPI NAME: FINALIZE NUMTIMES: 1 TIME: 0.000000

MPI NAME: INIT NUMTIMES: 1 TIME: 0.000000

As you can see not all calls have been accounted for, but the ones that dominate timewise have been included. The MPI_Isend accounts for both Isends and Ireceives. The next stage will be to verify these results but time has not allowed for this in the scope of this semester.

References:

Jansen, Kenneth. Computational Fluid Dynamics lectures: Spring 2008.

PetaApps proposal: RPI faculty.

Walkup, Bob. MPI Trace library.

Appendix: Flow of programs and what needs to be changed to compile them.

The program as it stands now consist of 3 parts (and therefore 3 files) they are as follows:

This part is done on the computer doing the simulation

mympi.c is the initial wrapper. Its purpose at this point is to run phasta and discover the sizes of memory used in all the mpi calls. It's output goes to intermedresults.out. This is to be run on the system doing the simulation.

Intermediate step: the intermedresults.out file must be copied to the computer being simulated and renamed intermedresults.x.out for 1 to Z where Z is the number of processors you are running phasta on.

tester.c is the file which contains the program which measures the timing for the simulation. It is run on the simulated system.

intermedresults2.out must be copied back to the first system. Like the last case Z copies must be made and put in the cwd for PHASTA in the form on intermedresults2.x.out .

adjustedmympi.c:

This runs PHASTA with the wrapper but uses the times from intermedresults2.n.out.

Compilation instructions: all the .c files use mpi and therefore must be compiled using mpicc. mympi.c and adjustedmympi.c do not have a main and therefore must be compiled using the -c option. When this is done these files produce mympi.o and adjustedmpi.o respectively. In order for the wrapper to be linked into PHASTA, the .o file simply must be included in phSolver/phSolver/common/obj/ibm-bg/mympi.o for the blue gene or the corresponding directory on another system. Once this is put there PHASTA must

be recompiled using a normal makefile (I believe gmake is usually used in each directory in a 'normal' make file. The useful output from the wrapper is at the end of the PHASTA output.

To compile the tester.c program you must simply do 'mpicc tester.c -o executable'. This can then be run using mpirun as any other mpi program.