

Development of an MPI profiling library on
the IBM BlueGene system

Timothy Hayes
Supervisor: Gilles Civario

August 24, 2009

Abstract

The aim of the project is to develop a light-weight MPI profiling library that differentiates between local communication (intra-node or nearby nodes) and distant communication (> 1 hop) within the 3D torus of BlueGene systems. The library should use the standard PMPI profiling interface and be as portable as possible. It involves the instrumentation in C - all the point to point MPI functions (MPI_[ISB]Send and MPI_[I]Recv) and communicator creation (to keep track of the actual MPI ranks). It is also necessary to identify a suitable trace file format, and developing a tool to parse/analyse the trace files with a high-level programming language (e.g. Python). The impact of process placement on torus will be assessed on systems such as BG/L, BG/P and Cray XT4 (Hector, if the library is portable).

Chapter 1

Introduction

This project has been undertaken as part of a 10 week summer scholarship programme at the Irish Centre for High-End Computing (ICHEC).

Many complex applications, especially in the area of simulation in computational science, can be parallelised in order to reduce the implied long-execution times. As high-performance computing machines increase the available number-crunching power in the form of more processors, the application shifts from a CPU-bound problem to an I/O-bound problem. It then becomes imperative to optimise the communication aspect of a parallelised program.

The mapping of virtual processes onto physical nodes is a very significant issue in high-performance computing. Depending on the topology of the machine's intra-connection, a poor layout of processes can lead to network congestion, increased latency, decreased bandwidth and additional CPU time spent on networking rather than computation. An ideal mapping should aim to optimise the overall process-to-process communication in all of these areas.

The overall goal of the project was to design and implement a toolset to evaluate the process placement of MPI jobs on a variety of high-performance computing machines. Specifically, to permit the user to observe the exact point-to-point bandwidth matrix; to infer the breakdown of bandwidth ordered by hop-count and also to provide estimations of a better process placement when possible. Although the project had been originally intended for the BlueGene family of supercomputers, it has been designed to be as general

as possible in order to make it portable over a large range of high-performance architectures.

The project has been implemented as two separate but symbiotic tools; a profiling library called `prmpi` and an evaluation application called `pranalysis`. This paper can be read verbatim on its own; as a complement to reading the source code or for guidance when using the tools. Chapter 2 gives a detailed look at the design and implementation of the tools; chapter 4 looks at some of the issues and caveats of the software; chapter 3 gives a critical evaluation of the utilities with some experiments and chapter 5 gives a comprehensive set of instructions for using the tools.

Chapter 2

Design & Implementation

This section gives a detailed description of the design and implementation of `prmpi` and `pranalysis` respectively.

2.1 `prmpi`

`prmpi` (profile mpi) is a very light-weight library that collects various statistics of an MPI job. It is constructed as a static library that is linked in to MPI programs and overloads many of the MPI functions using a technique called interposition[7]. The MPI standard ensures that there are two definitions for every function in the API; the `MPI_*` functions and the `PMPI_*` methods. The `PMPI_*` functions contain the actual implementation of the routines whereas the `MPI_*` functions are by default merely a wrapper which call their `PMPI_*` equivalent. Using interposition, a program can redefine the `MPI_*` calls to perform extra logic, then instead of having to find the original `MPI_*` symbol, the program merely calls the matching `PMPI_*` function. Provided the user-code uses `MPI_*` calls and not `PMPI_*` calls, they will go through this library. This separates the utility from any specific MPI implementation and makes the library very portable.

`prmpi` records four pieces of information: the amount of information exchanged between unique processes; the frequency of individual exchanges between unique processes; the total amount of time spent in MPI commu-

nication routines and the overall time of the application. The library was designed to log data accumulatively rather than event-based; this implies one can determine that process a sends process b x bytes of information in the scope of the program but cannot determine how many bytes had been transferred in a particular MPI call. It also implies one can determine the total amount of time process a spends waiting for all MPI functions to complete but not the amount of time any individual function took to complete.

The rationale behind this is that we are not interested in the communication time between processes per se, rather we are interested in the time spent performing busy waits in the function calls themselves. For example, it may take a chunk of data a large amount of time to transfer between processes, but if it is performed asynchronously and `MPI_Wait` is called some point after the data has arrived, it will return immediately having only spent a short time in the function e.g. see Figure 2.1. However if `MPI_Wait` is called prior to the arrival of the data, it will block e.g. see Figure 2.2. This is the time that is of interest to us. Event-based tracing is not the goal for this profiling tool and there already exists a variety of tools to achieve this.

The library is portable relying on standard C libraries and standard MPI functions for implementation. It provides bindings for C, C++ and Fortran. The library itself is managed with GNU Autotools[8], this is particularly important for Fortran bindings which have their function names mangled in different ways depending on the environment. Autotools implies that an administrator/user follow the conventional `configure`, `make` & `make install` process. The library itself is implemented in C and uses `MPI_*_c2f` and `MPI_*_f2c` functions for Fortran conversions. The library has overloaded all point-to-point and collective functions however this information remains permanently disjoint as the particular architecture of the machine may or may not use separate networks for each kind of communication.

`MPI_Init` is used to initialise structures for logging data. To avoid excessive space, a simple hash table is used that keeps a structure containing point-to-point bytes sent & call frequency as well as collective bytes sent & frequency. The bucket size is $\log_2(\text{number-of-processes})^2$ which was empirically derived as a good value. Whenever an MPI communication function is called, it first converts the destination rank into the equivalent rank in `MPI_COMM_WORLD` (which will be the same if the communicator is `MPI_COMM_WORLD`); it then uses this global rank as a key for the hash table in order to get a structure containing statistics for that particular process. The structure's bytes sent value is then updated and frequency incremented. `PMPI_Wtime` is

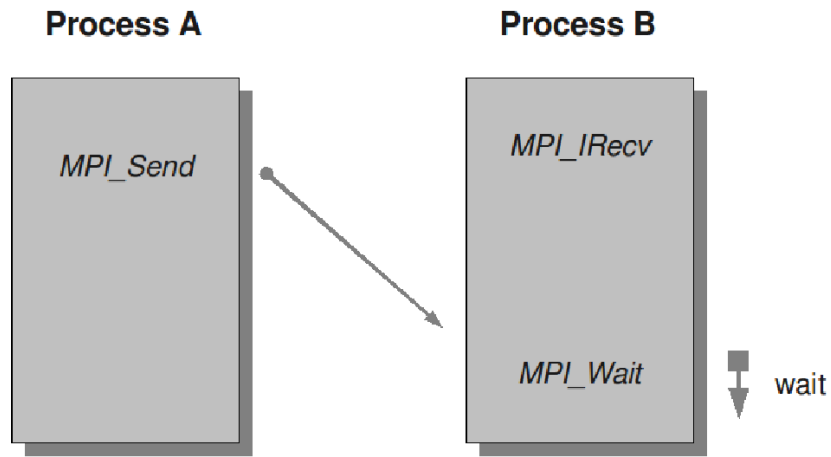


Figure 2.1: The `MPI_Wait` is invoked just as or after the `MPI_Irecv` has completed

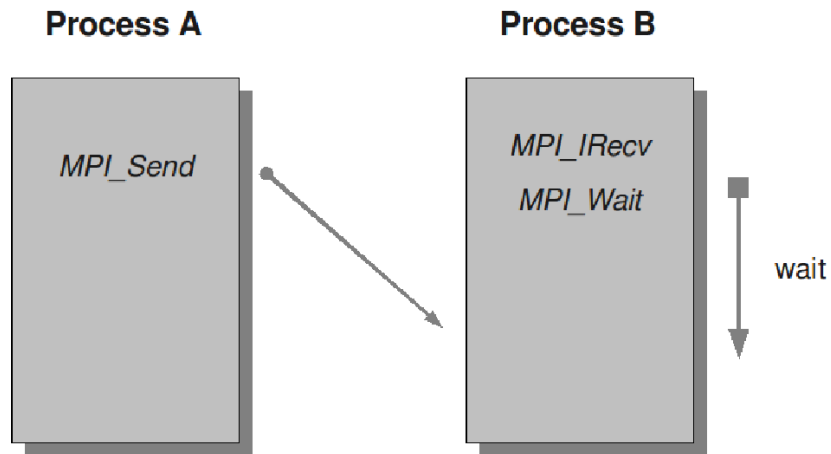


Figure 2.2: The `MPI_Wait` is invoked before the `MPI_Irecv` has completed leading to an undesired latency

called immediately before and after the real `PMPI_` function is invoked and the difference is then added on to a cumulative time value for that process (separate ones for point-to-point and collective operations). The main overhead of the library is contained in the `MPI_Finalize` function that uses several gather operations to collect the data into the root process (i.e. rank 0) and then outputs the results as a formatted XML file.

The library is activated by default and can be deactivated without re-linking the programming by exporting the environment variable `PRMPI_NOLOG`. This will inhibit any any extra overhead and should perform as well as the application running without the library linked in.

The default naming scheme for the XML files is `profile_*.xml` where `*` is from 0 to 999. The library will look for the first file-name that does not exist in the working directory. For example, if `profile_0.xml` exists and `profile_1.xml` does not, it will write the profiling data to `profile_1.xml`. This was implemented for convenience and safety for the end user so as not to accidentally overwrite important profiling data.

2.2 pranalysis

`pranalysis` (profile analysis) is a complementary tool to `prmpi` which parses and analyses the information contained in the XML trace files. It is a relatively small command-line utility written in C. The tool can do a variety of tasks including outputting the statistics of the program in terms of hop counts; displaying best/worst/average timing information; exporting the communication matrix in various formats and suggesting a better process placement scheme that may lower the overall time of the application.

Because `prmpi` was made to be portable, all architecture-specific processing had to be contained in this auxiliary tool. In order to keep `pranalysis` portable as well, a modular design was chosen based on the concept of plugins. The idea is that any part of the utility which can't be generalised should be isolated away from the main area of the program. For example, determining the hop-counts between processes is not going to be the same on the BlueGene/L as it is on an SGI Altix. This led to the definition of an interface header file that is implemented by architecture-specific plug-ins. This ultimately means the main program does not need to be updated and

recompiled whenever a new architecture is introduced; instead, a loadable module for that specific architecture can be created from a template and then dynamically loaded into the main application.

Internally the program works with two matrices; a static data matrix derived from the statistics of the program itself, and a re-arrangeable cost matrix based on the topological location of the processes. The data matrix simply takes the lump-sum of bytes sent between processes, the rows represent the sending process and the columns represent the receiving process, all ordered by `MPI_COMM_WORLD` rank. The cost matrix holds hop-counts and is derived from the string retrieved by `MPI_Get_processor_name`. Because this string is going to be different depending on the architecture, the cost matrix should be constructed in the plug-in rather than main program. The program could easily be modified to take a different form for both matrices e.g. the data matrix could contain frequency of MPI calls instead.

The program relies on two libraries; `libxml`[9] for parsing the XML files and `argtable`[10] for parsing the command-line arguments. `libxml` is a very typical library to have installed and is therefore not bundled with the tool; however `argtable` is a little more unusual and so is built into the tool. Both libraries are released under the GNU public licence.

The utility's main objective is to provide locality statistics of an MPI job. Therefore invoking the tool with some `prmpi` XML file and an architecture-specific plug-in will output something similar to Listing 2.1. The main interest here is the total bandwidth of the application which has been organised in terms of hop-counts. In addition, also shown is the process with the best (shortest) busy-wait time, the process with the worst (longest) busy-wait time, the average of all busy-wait times and the total running time of the application.

The application can also output both the communication matrix and the hop-count matrix for a detailed inspection by the end-user. The format of the outputted file can be modified with a command-line argument; currently the options are a tab-delimited plain-text file or a CSV file. The main application has been written so new output methods can be easily added.

The secondary goal of this tool was to provide some hints for a better process placement. Initially it was assumed that diagonalisation algorithms such as the CutHill-McKee Algorithm[1] would suffice, however it became apparent that most high-performance topologies are not flat and therefore

```

*****
*  pranalysis  *
*  v0.1        *
*  ICHEC 2009  *
*****

Total application time is: 305.380048
Average process waiting time is: 8.169779
Worst case process waiting time is: 9.424892
Best case process waiting time is: 6.909324

Original process layout has the following:
Hop 0: 0 bytes of 33731832 bytes - 0.000000%
Hop 1: 1108732 bytes of 33731832 bytes - 3.286901%
Hop 2: 8803884 bytes of 33731832 bytes - 26.099632%
Hop 3: 12929900 bytes of 33731832 bytes - 38.331449%
Hop 4: 8693800 bytes of 33731832 bytes - 25.773281%
Hop 5: 2195516 bytes of 33731832 bytes - 6.508736

```

Listing 2.1: Sample pranalysis output

trying to diagonalize the communication matrix would be in vain since there is not necessarily one single perfect gradient originating from the central diagonal.

An alternative approach is to associate a cost value between two processes based on their physical locations and try and find an arrangement that minimises the summation of cost values. Since we have already derived a hop-count between processes, it seemed natural to use this to build a cost function. For example, if process i sends process j x bytes of data and there are 3 physical hops between them, we can say the cost between these two processes is $3 * x$. The whole cost function could then be $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} hop(i, j) * sent(i, j)$ where n is the amount of processes. This function straight forward and easy to calculate; the problem is then finding a good way to search for a process arrangement that finds the minimum value given by the cost function. If we naively tried to permutate every possible arrangement, the problem becomes NP-hard since permuting n processes over n physical locations is $n!$. For a small job of 32 processes, this is already $2.63130837 * 10^{35}$ permutations and grows exponentially as n increases.

The problem with searching for a global optimum has to do with the nature of greedy algorithms[6]. A greedy algorithm will attempt solve the search space by heuristically choosing a local optimum at every stage in the hope of finding an overall global optimum. For many problems this approach is too naive; a global optimum can only be found by exploring outside of local minima. Bandwidth optimisation problems frequently have this issue.

A paper from IBM[2] looks at a similar issue for their BlueGene toroid topology and suggests the use of simulated annealing to reduce the bandwidth by pseudo-randomly mutating the physical location of processes in order to find a global minimum cost. Their implementation uses the Metropolis-Hastings Algorithm[3] to search the domain. The benefit of using this method over greedy search algorithms is that greedy searches can be trapped in local-minima whereas simulated annealing has the potential to explore the entire space by allowing for some cost increases instead of solely cost decreases. This method was chosen for pranalysis as it could be used as a general bandwidth optimisation solution rather than one specifically for the BlueGene.

The general algorithm of simulated annealing is analogous to annealing metals in order to strengthen them; it searches for a global minimum cost using a Markov chain Monte Carlo method. It has four parameters: an initial temperature; a final temperature; a temperature attenuation and an iterations per temperature. The idea is that the algorithm starts with a high temperature which allows more negative (cost increasing) changes to be accepted, the temperature is gradually decreased to a point where only positive (cost decreasing) changes are accepted. The system is given an initial default state and the cost is evaluated. Two randomly chosen processes are switched and the cost is evaluated again. If the cost is lower, the change is accepted, otherwise the change is accepted if $r < e^d$ where r is a random real number uniformly distributed in $(0,1)$ and d is $\frac{(newcost-oldcost)}{temperature}$. After so many iterations at a given temperature, the temperature is attenuated. This process continues until the temperature reaches a lower threshold.

The algorithm was implemented similarly to the IBM paper making a few changes. A Mersenne Twister[4] was used as the pseudo-random number generator. The GNU C library's implementation of `rand` uses a linear congruential generator which is not generally suitable for Monte-Carlo simulation, the Mersenne Twister algorithm has a longer period and higher order of equidistribution than many other pseudo-random number generators. The Mersenne Twister algorithm is particularly fast when it comes to number generation and has a very low memory footprint.

Instead of reevaluating the entire cost function each time two processes are switched, only the change in cost is calculated. This implies that if we switch process i and process j , we can calculate the costs *only* at rows and columns i and j , subtract this value from the already known total cost, switch the values and calculate the new cost at rows and columns i and j and finally add this to the total cost. This means that the overall total cost only has to be evaluated once at the beginning of the algorithm which cuts down processing time significantly.

When evaluating if a change to the matrix arrangement is accepted (based on the old cost, the new cost and a heat value which allows more significant cost increases to be accepted at high values) it was found that using a relative change produced better mappings than using an absolute change. This ultimately meant replacing a subtract with a divide which is quite significant for a function that is called very frequently; however this sacrifice is made for the potential of getting a better process placement.

The simulated annealing process requires four parameters from the user which are not always intuitive let alone easy to guess. In order to relieve them of any guesswork, all of the values can be automatically found. This should give *good* results but perhaps not the *optimum*. [5] proposes some ideas to find a temperature that will yield some desired acceptance rate and [2] suggests the initial temperature have an acceptance rate of $\sim 20\%$ and a final temperature have an acceptance rate of $\sim 0\%$.

To achieve this, an algorithm to find a temperature based on a desired acceptance rate was implemented. Initially the temperature is taken as 1.0 and a finite number of samples are taken giving the acceptance rate for 1. If the acceptance rate is not within some threshold of the desired acceptance rate we then multiply the temperature by $\frac{\ln x}{\ln y}$ (where x is the current acceptance rate and y is the desired acceptance rate) and then the space is sampled again. This continues until a suitable temperature is found, or the acceptance rate becomes invariant of the temperature itself.

The automatic system will try to find an initial temperature and final temperature with acceptance rates of 20% and 0% respectively. By default, the amount of iterations per temperature is 1024. This was empirically derived as a good value that is not too computationally demanding. The annealing rate is calculated so that there are 64 attenuations from the initial temperature to the final temperature. This was simply calculated as $\sqrt[64]{\frac{x}{y}}$ where x is

the final temperature and y is the initial temperature.

2.3 BlueGene/L plug-in

Because the original goal of the project was to develop evaluation tools for the BlueGene architecture and also given the fact that the BlueGene's point-to-point toroid network is a particularly interesting one, an extra section is given to look at the design of the pranalysis plug-in for the BlueGene/L.

The plug-in interface has three function prototypes. `create_cost_matrix` which returns a `square_matrix` data-type filled with the hop-count between any two global ranks. `create_mapping` which outputs a mapping file for this specific architecture based on an arrangement passed as a parameter. `use_collectives` which returns either true or false based on whether this architecture should be concerned about collectives in its output/optimisations.

The `create_cost_matrix` function relies heavily on the information obtained from `MPI_Get_processor_name` which is contained in the XML file in the `<cpu>` `</cpu>` tags. The format of the BlueGene/L's string is similar to: `Processor <0,0,0,0>` in a `<4, 4, 2, 1>` mesh *or* `Processor <0,0,0,0>` in a `<8, 8, 8, 1>` `torusX=1, torusY=1, torusZ=1`; from this we can get the mesh/torus shape and also the position of the process in the mesh/torus. The mesh/torus takes the format of `<x, y, z, t>` where xyz represent the particular node and t represents the thread on this node; it should be clear that `<4, 4, 2, 1>` has only one thread per node.

A node is adjacent to another node if any *one* of its xyz values is ± 1 different from the other e.g. `<0,1,0,0>` is adjacent with `<0,0,0,0>` and also `<0,2,0,0>`. Because of the nature of a torus, the BlueGene's network also wraps around on itself, so the first and last nodes are adjacent too e.g. `<0,0,0,0>` is adjacent with `<0,3,0,0>` and also `<0,1,0,0>` but specifically in this `<4, 4, 2, 1>` mesh. For processes on the same node we make one further distinction; we assume a process sending to itself is 0 hops and a process sending to another process on the same node is 1 hop; therefore we can also assume adjacent nodes are 2 hops.

From this we can derive a simple algorithm to get the distance between two processes knowing the mesh and their positions on the torus. See Listing

2.2. `create_cost_matrix` calculates the distance between every pair of processes in n and returns a matrix data-type to be used by the main program.

```
int distance(xyzt mesh, xyzt a, xyzt b)
{
  if ( (a.x = b.x) & (a.y = b.y) & (a.z = b.z) )
  {
    if (a.t = b.t)
      return 0;
    else
      return 1;
  }

  int x :=
    min
    (
      ((a.x - b.x) + mesh.x) MOD mesh.x,
      ((b.x - a.x) + mesh.x) MOD mesh.x
    )

  int y :=
    minimum
    (
      ((a.y - b.y) + mesh.y) MOD mesh.y,
      ((b.y - a.y) + mesh.y) MOD mesh.y
    )

  int z :=
    minimum
    (
      ((a.z - b.z) + mesh.z) MOD mesh.z,
      ((b.z - a.z) + mesh.z) MOD mesh.z
    )

  return x + y + z + 1;
}
```

Listing 2.2: Calculate the distance between two processes

`create_mapping` outputs a mapping file specifically for the BlueGene/L.

The format is `x y z t` per line, where the line represents the global rank. So by placing `1 1 1 1` on the first line, it will map rank 0 to `<1, 1, 1, 1>` on the torus. The input simply takes an array of integers representing the desired arrangement (this is derived from the simulated annealing process) along with the XML document. The function will match the ranks in the array with the processor name in the XML document, extract the `xyzt` values, and write them in this order to a file.

`use_collectives` is the simplest function of them all, it simply returns true unconditionally. This is because the BlueGene/L's collective network is completely separated from the point-to-point torus network and therefore there is no need to consider the collectives when mapping processes.

Chapter 3

Results

This section primarily looks at the effects of remapping processes after a pass through the pranalysis simulated annealing algorithm. All tests were run on the BlueGene/L.

3.1 Simple Hop Optimisation

The first benchmark was to test effectiveness of simulated annealing on an incredibly poor process placement. Since any well known scientific application is not going to exhibit signs of extremely low performance, a dummy application was created. The notion was that each process in an MPI job would send a very large chunk of data to one other process in the mesh simultaneously. This would create a lot of traffic on the torus network and if the hop-count is high for each send, it should take a longer amount of time to complete than if the hop-count was low. By examining the hop-count matrix of a BlueGene/L job with 512 processes in co-processor mode, it can be inferred that sending to a process's own global rank + 292 would send to a process between 11 and 13 hops away (mostly 12).

The experiment was to see if simulated annealing could reduce some or all of these high hop-counts. Calling the simulated annealing with the automatic values estimated a potential reduction in waiting time by a factor of 3.7x. By tweaking the simulated annealing parameters manually this was increased to

Hop-count	Percentage	Hop-count	Percentage
0	0.0%	0	0.0%
1	0.0%	1	0.0%
2	0.0%	2	63.87%
3	0.0%	3	22.1%
4	0.0%	4	11.33%
5	0.0%	5	2.34%
6	0.0%	6	0.2%
7	0.0%	7	0.2%
8	0.0%	8	0.0%
9	0.0%	9	0.0%
10	0.0%	10	0.0%
11	31.25%	11	0.0%
12	43.75%	12	0.0%
13	25.0%	13	0.0%

(a) Original breakdown (b) Remapped breakdown

Figure 3.1: Simple Hop Breakdowns

Measure	Original	Remapped	Speedup
Total	361.16	161.79	2.23x
Average	236.48	72.45	3.26x
Worst	327.39	127.99	2.56x
Best	119.99	54.05	2.22x

Figure 3.2: Simple Hop Timings

a factor of 4.7x. Figure 3.1 shows the old and new hop-count breakdowns. When run with the new mappings the overall application time decreased by a factor of 2.23x. Taking an average of multiple runs, the results in Figure 3.2 were obtained. From this we can deduce that simulated annealing can make optimisations that can significantly reduce the timing of an application. The thing to bear in mind, however, is that this dummy application has very little processing overhead and performs poorly *strictly* due to its point-to-point communication.

3.2 Chaotic Optimisation

The previous test had been a proof of concept test designed specifically to see if simulated annealing could optimise a domain that is quite obviously optimisable. The next experiment tests simulated annealing where the optimisation is not so obvious by observation. Once again an application was designed for this purpose and works by using a pseudo-random number generator to find every process 12 (not strictly unique) processes to send a large chunk of data to. The same seeds were used each time so that the tests could be repeated with the new process mapping.

Because of the chaotic nature of the random process selection, a bandwidth matrix was produced that would be far too difficult to optimise by hand. The simulated annealing function found a potential reduction in waiting time by 1.28x. Figure 3.3 shows the hop-count breakdown of the original and optimised mappings. While high distance communication is not eradicated completely, it is reduced significantly. Figure 3.4 shows the timings of the original and optimised layout. While it hasn't been optimised as much as the simple hop experiment, there is still an overall optimisation of 1.23x. This shows there is potential to use simulated annealing successfully with very random communication patterns.

3.3 DL_POLY Optimisation

The previous experiments showed good results, but unfortunately the applications benchmarked had specifically been designed to illustrate the effectiveness of the optimisation process and do nothing useful in themselves. For an honest comparison, the molecular dynamics simulation package DL_POLY3[11] was benchmarked. The benchmarks included with the package were used; the results here are for benchmark 4 - DMPC (dimyristoylphosphatidylcholine) in water (413896 atoms) run on 32 processes in co-processor mode. The original and remapped versions were run 10 times taking an average of the results obtained.

The simulated annealing algorithm seemed to be able to rearrange the processes quite well, bringing the amount of adjacent communications from $\sim 50\%$ to $\sim 90\%$ and predicting a potential communication reduction of

Hop-count	Percentage	Hop-count	Percentage
0	0.13%	0	0.13%
1	0.0%	1	0.0%
2	1.06%	2	5.71%
3	3.24%	3	10.92%
4	7.18%	4	16.10%
5	11.57%	5	18.03%
6	16.32%	6	17.80%
7	18.21%	7	14.16%
8	16.28%	8	9.51%
9	13.10%	9	4.70%
10	7.78%	10	2.07%
11	3.68%	11	0.73%
12	1.2%	12	0.10%
13	0.16%	13	0.03%

(a) Original breakdown

(b) Remapped breakdown

Figure 3.3: Chaotic Breakdowns

Measure	Original	Remapped	Speedup
Total	7.81	6.36	1.23x
Average	7.07	5.63	1.26x
Worst	8.88	6.98	1.27x
Best	5.19	3.71	1.40x

Figure 3.4: Chaotic Timings

Hop-count	Percentage	Hop-count	Percentage
0	0.0%	0	0.0%
1	0.0%	1	0.0%
2	50.12%	2	90.37%
3	40.53%	3	8.96%
4	8.92%	4	0.47%
5	0.33%	5	0.17%
6	0.10%	6	0.03%

(a) Original breakdown (b) Remapped breakdown

Figure 3.5: DL_POLY Breakdowns

Measure	Original	Remapped	Speedup
Total	298.64	298.04	1.002x
Average	8	7.56	1.06x
Worst	9.26	8.93	1.04x
Best	6.73	6.2	1.09x

Figure 3.6: DL_POLY Timings

1.23x, see Figure 3.5. The actual speedup was not as significant as this; Figure 3.6 shows the timings of the original and remapped runs. Although the average, best & worst times decreased a little bit, the overall time barely decreased at all. Considering the average communication waiting time is only 2.68% of the overall time of the application, it is likely that optimising the mapping isn't going to have as significant an affect as the previous tests had.

Another thing to note is that the standard deviation/standard error for this experiment was 2.39/0.76 and 2.05/0.65 for the original and remapped versions respectively. These errors are greater than the time delta between the averages - 0.60. The source of this deviation is probably due to the program's reliance on hard disk I/O. The program relies heavily on a large input file $\sim 30\text{Mb}$ and progressively writes its results to another large output file $\sim 115\text{Mb}$. For this fact it can be assumed the biggest bottleneck is not in the program's process mapping.

Chapter 4

Issues Encountered

This chapter looks at various issues encountered in the design/implementation stages.

4.1 BlueGene/P Processor Names

The BlueGene/L was a relatively simple architecture to derive processor placement information. Calling `MPI_Get_processor_name` returns a string similar to the format `Processor <0,0,0,0>` in a `<4, 4, 2, 1>` mesh. The second set of numbers is particularly important for deriving the distance between processes in the job due to BlueGene's torus topology. For example, in a `<4, 4, 2, 1>` mesh a process located at `<0,0,0,0>` will be adjacent with a process located at `<3,0,0,0>` due to the wraparound nature of a torus. However, if the mesh was `<8, 4, 2, 1>` the processes can no longer be considered adjacent.

BlueGene/P has a different format for its `MPI_Get_processor_name` implementation and returns a string similar to `Rank 0 of 512 <0,0,0,0> R00-M0-N00-J23`. Immediately it can be seen that the mesh information is no longer present. For this reason, the BlueGene/P plug-in can only be used if a full job is run i.e. there is a process on every node of the mesh given.

4.2 Reductions

There were a variety of collective operations that had to be overloaded; each time the a collective was called the communicator had to be queried for the processes contained inside and then the logging information be updated for those processes in particular. For example, an `MPI_Scatter` will log transfers from the root process to every process in the communicator. For the majority of collectives this is fine, but for reductions the MPI standard defines no rules as to how they should be implemented. For this reason, `MPI_Reduce`, `MPI_Reduce_scatter` & `MPI_Allreduce` merely log the time spent in the function and increment the count of the collectives frequency but do not record process-to-process bytes sent.

4.3 SGI Altix ICE

Due to the limited information obtained for the SGI Altix ICE 8200EX[12], certain simplifications had to be made. The general hierarchy is: rack \Rightarrow integrated rack unit \Rightarrow node \Rightarrow processor \Rightarrow core. Calling `MPI_Get_processor_name` will retrieve a string similar to: `r2i1n15`. The processor number and core number are not included in this string and cannot be obtained using standard MPI calls. Another quirk is that nodes are inter-connected within a single integrated rack unit in a hypercube topology. This topology cannot be derived from the node numbers themselves so a simplification was made and all intra-IRU communication is taken to be 2 hops.

Chapter 5

Usage

This chapter provides comprehensive instructions to using `prmpi` and `pranalysis`.

5.1 `prmpi`

`prmpi` comes packaged as a tarball, to untar issue the following command

```
tar -xzvf libprmpi-0.1.tar.gz
```

`prmpi` must be configured for a specific platform which can be achieved by calling the `configure` script. Because many architectures contain multiple compiler suites, the `CC` and `F77` flags can be used to specify the desired compilers. To make things simple, the MPI compiler wrappers should be used here rather than the compilers themselves. A Fortran 90 compiler can be used in lieu of a Fortran 77 one.

```
./configure CC=mpicc F77=mpif77  
./configure CC=mpixlc F77=mpixlf77  
./configure CC=mpicc F77=mpif90
```

Calling `make` will build a static library called `libprmpi.a` in the `src` directory. Calling `make install` will install copy this to the default library

location (usually `/usr/local/lib/` unless the configure script is invoked with `--prefix=$HOME` or something similar.

```
$ make
# make install
```

To build your MPI applications with the library, one simply needs to append an extra argument to the compilation string. It is important to remember this argument must be the *final* statement of the linking command.

```
mpicc -o program.exe file1.c file2.c -lprmpi
```

One executes the program as normal and if the program exits normally there will be a file `profile_0.xml` in the working directory that contains the profiling data. If the program is run again without deleting `profile_0.xml`, then the data will be written to `profile_1.xml` instead (this pattern is consistent up to 999).

In order to turn off profiling without relinking, the environment variable `PRMPI_NOLOG` must be set. This must be done based on the system's particular configuration. For LoadLevel submissions, something like following should work

```
/bgl/BlueLight/ppcfloor/bglsys/bin/mpirun \  
-np 512 \  
-mode CO -cwd /icheck/home/users/user/program \  
-exe /icheck/home/users/user/program/program.exe \  
-env 'PRMPI_NOLOG=1'
```

5.2 pranalysis

pranalysis also comes packaged as a tarball, to untar issue the following command:

```
tar -xzvf pranalysis-0.1.tar.gz
```

The package should be configured, made and if possible, installed. The following will make & install the main program and all of the plug-ins.

```
./configure
$ make
# make install
```

To use the program, `pranalysis` must be invoked with a minimum of two arguments: the architecture specific plug-in and the XML file from `prmpi`. The plug-in is specified with the `-l|--lib=<file>` flag and the XML file is taken as the main argument. To output general statistics of a program run, issue something like:

```
pranalysis -lbluegenel profile_0.xml
```

The program will search for plug-ins in the `/usr/local/lib/` directory by default, if a full install hasn't performed, there is a requirement to set the environment variable to search additional directories. There are currently plug-ins for BlueGene/L (`bluegenel`), BlueGene/P (`bluegenep`).

To output the cost matrix or data matrix, the flags `-c|--cmatrix=<file>` or `-d|--dmatrix=<file>` can be used respectively. The default output format is a comma delimited CSV file; in order to change this the flag `-f|--matformat=<string>` can be used. Currently the options are either CSV or TXT. Files are written to the current working directory. For example:

```
pranalysis -lbluegenel profile_0.xml -cmatrix.txt -fTXT
```

Collective statistics are merged with the point-to-point statistics based on the plug-in used. Because the BlueGene systems have a separate collectives network these are not counted in the statistics. The user can force the collectives to be considered by setting the `-o|--forcecol` flag.

In order to perform simulated annealing to possibly find a better arrangement of the processes, the `-a|--anneal` should be set. This does not require any additional arguments and will try and automatically find good values for the initial temperature, final temperature, attenuation rate and iterations per temperature. The user can fine-tune these values with the `-w|--anmax=<double>`, `-x|--anmin=<double>`, `-y|--anrate=<double>` & `-z|--anits=<int>` flags respectively. By default, annealing will only output the hop-count statistics of the new arrangement. To write to a mapping file specific to the machine, the `-m|--map=<file>` flag should be used. A typical annealing invocation may look like this:

```
pranalysis -lbluegenel profile_0.xml \  
-a --anmax=0.35 --anmin=0.012 \  
--anits=16384 --anrate=0.99 \  
--map=bg.map
```

In this example, the mapping file is made specifically for the BlueGene/L system. In order to use it, it should be uploaded to some directory in the BlueGene/L system and the LoadLeveler script will take this form:

```
/bgl/BlueLight/ppcfloor/bglsys/bin/mpirun \  
-np 512 \  
-mode CO -cwd /ichec/home/users/user/program \  
-exe /ichec/home/users/user/program/program.exe \  
-env 'BGLMPLMAPPING=/ichec/home/users/user/bg.map'
```

Bibliography

- [1] E. Cuthill and J. McKee. “*Reducing the bandwidth of sparse symmetric matrices*” In Proc. 24th Nat. Conf. ACM, pages 157-172, 1969.
- [2] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, R. Walkup, (2005), “*Optimizing task layout on the Blue GeneL supercomputer*”
- [3] W. R. Gilks, S. Richardson and D. J. Spiegelhalter, (1996) “*Markov Chain Monte Carlo in Practice*”, Chapman and Hall/CRC, pages 5-17
- [4] M. Matsumoto and T. Nishimura, “*Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*”, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998)
- [5] Walid Ben-Ameur, “*Computing the Initial Temperature of Simulated Annealing*”, Computational Optimization and Applications, v.29 n.3, p.369-385, December 2004
- [6] Paul E. Black, (2005), “greedy algorithm in *Dictionary of Algorithms and Data Structures*”, U.S. National Institute of Standards and Technology
- [7] Yukon Maruyama and Marty Itzkowitz, (2009), “*Profiling MPI Applications*” available at http://developers.sun.com/sunstudio/documentation/techart/mpi_apps/
- [8] “*Autoconf - GNU Project - Free Software Foundation (FSF)*” available at <http://www.gnu.org/software/autoconf/>
- [9] “*libxml - The XML C parser and toolkit of Gnome*”, available at <http://xmlsoft.org/>

- [10] Stewart Heitmann, “*Argtable - ANSI C command line parser*”, available at <http://argtable.sourceforge.net/>
- [11] W. Smith, T.R. Forester and I.T. Todorov, (2009), “*The DL-POLY Molecular Simulation Package*”, available at http://www.cse.scitech.ac.uk/ccg/software/DL_POLY/
- [12] Irish Centre for High-End Computing, (2009), “*Stokes on SGI Altix ICE 8200EX*” available at <http://www.ichec.ie/infrastructure/stokes>