



Initial Experiences Porting Hydra_MPI, which requires MPI-2 remote memory access calls, to HPCx

Paul Walsh and Gavin J. Pringle

January 28, 2004

Abstract

This paper describes the initial experiences of porting and optimising Hydra_MPI on to HPCx. Hydra_MPI is an astrophysical code which belongs to the Virgo Consortium. The code is a portable parallel N -body solver, based on an adaptive P^3M , written in Fortran90, which utilises several of the remote memory access routines of MPI-2, where MPI-2 is a set of extensions to the MPI standard. We have ported the code to HPCx and have investigated tuning the code for this platform, with a view to introducing self-tuning.

Contents

1	Introduction	1
1.1	Hydra_MPI	1
1.1.1	Self-tuning	2
1.2	What MPI-2 routines does Hydra_MPI utilise?	2
1.3	The Background Technical Details	2
2	Porting	3
2.1	Debugging Tools	3
2.1.1	Using TotalView on HPCx	3
2.2	Problems and Issues	3
2.2.1	MPI Trace	3
2.3	The evaluation of HPCx	4
3	Profiling	4
3.1	Profiling Hydra_MPI	5
3.2	Timing codes on HPCx	7
4	Introducing self-tuning to Hydra_mpi	7
4.1	Tunable variables	7
4.1.1	Primary Tunables	7
4.1.2	Secondary Tunables	8
4.1.3	Tertiary Tunables	8
5	Conclusions	10
5.1	Future Work	11

List of Tables

1	List of primary tunables	7
2	List of secondary tunables	8
3	List of tertiary tunables	8

List of Figures

1	Xprofiler calling tree display for Hydra_MPI	6
2	Comparison of Tuned and Untuned code for data set d6065.1998	10

1 Introduction

This report describes our experiences in the porting of Hydra_MPI [1] to HPCx. The Hydra_MPI code is an astrophysical code which belongs to the Virgo Consortium [5]. The code is a portable parallel N -body solver based on an adaptive P³M algorithm which utilises several of the Remote Memory Access (RMA) routines of MPI-2. MPI-2 is a set of extensions to the MPI (Message Passing Interface) standard.

As far as we are aware, this is the only industrial-strength research code which utilises the RMA routines of MPI-2, namely, MPI_Put/Get, MPI_Accumulate and the communications epochs MPI_Fence and, most pointedly, MPI_Lock, we thought it would be of general interest to port the code to HPCx to gauge the RMA performance of the IBM MPI library, within a real code.

Furthermore, we considered the possibility of introducing self-tuning to Hydra_MPI.

This work was carried out by an EPCC summer intern, Paul Walsh, supervised by Dr. Gavin Pringle, over a 12 week period.

1.1 Hydra_MPI

As previously stated, Hydra_MPI is a portable parallel N -body solver based on an adaptive P³M (AP³M) algorithm which utilises several of the RMA routines of MPI-2.

A brief description is now given, for the purposes of this technical report, however, a more detailed description, along with details of its parallelisation can be found in [2].

The N -body solver itself is constructed out of an adaptive Particle-Particle, Particle-Mesh (PPPM, or P³M) algorithm.

The force on each particle is determined by the location and strength of all the other particles. The force from close particles is computed directly. This is known as the particle-particle, or PP, section. The force from distant particles is computed, via a Fast Fourier Transform, by convolving the density-mesh with a Green's Function. This is known as the particle-mesh, or PM, section. Hydra_MPI employs FFTW [4] for the PM sections. Both the PP and PM sections together form the PPPM, or P³M, algorithm

When the particles are smoothly distributed, the algorithm is “non-adaptive” and is simply a P³M algorithm. However, when the distribution is clustered, these clusters are “removed” and treated as separate P³M problems. In turn, these clusters can contain clusters themselves (*parent* and *child*) and so the overall simulation becomes hierarchical in nature. Hence, for general clustered distributions of matter we require an *adaptive* P³M code.

Clusters are located within the code by means of a *refinement placing algorithm*.

The performance of this parallel code depends on many factors, such as the number of particles in the simulation and their level of clustering, the number of processors and the amount of memory available, the speed of communications, etc. Further, we expect the refinement placement algorithm to have a strong influence on the overall performance of the code.

The code has been tuned to run quickly on a Cray T3E-900, which was housed at EPCC [6]. However, when the code is ported to other platforms, we expect that the performance may be poor. Thus, we have also viewed this project as an opportunity to investigate the introduction of self-tuning to the code so that Hydra_MPI can perform competitively on any platform, “straight out of the box”.

Apart from this IBM, the code has been executed on a wide range of machines, namely the Cray-T3E, Sun E3500 SMP, a Compaq SMP, SGI Origin 2000 and Beowulf systems. The Beowulf systems included Sun systems, an SGI-1200 and a generic alpha cluster using the MPI libraries from Sun, SGI and LAM.

1.1.1 Self-tuning

The human effort required by the manual tuning process makes it a slow and expensive exercise, thus a code which *self tunes* can be of great interest.

A self-tuning code is useful for the following reasons [3].

- Significant performance improvement is achieved.
- It relieves the user from substantial time and effort spent in tuning.
- Several tuning iterations can be performed in an interactive manner, and at a modest price.
- Heavy reliance on ingenuity and expertise is not mandatory for success.
- In contrast to a simulation, the design cycle is short and economical.
- Complete transparency to the user is feasible if the technique is completely automated.

It can be argued that a portable high-performance code, which has been tuned for a particular machine, is, in fact, not portable. A self-tuning code can overcome this handicap. There are two ways to achieve self-tuning: a code can

- probe the hardware, at the point of installation, to determine various machine characteristics, and then generate code that takes advantages of this particular architecture.
- monitor the time for previous iterations and alter parameters within the code depending on past performance.

Of course, these two techniques are not mutually exclusive and can be used in conjuncture to produce code. Indeed, the FFTW library [4] employs both these techniques to great effect to produce FFT routines which can be highly competitive with the vendor's own FFT routines.

1.2 What MPI-2 routines does Hydra MPI utilise?

The main MPI-2 routines employed by Hydra MPI come from the remote memory access, or RMA, section of the MPI-2 standard which allow for single-sided communications to take place. These are `MPI_Win_create/MPI_Win_free`, `MPI_Put/MPI_Get/ MPI_Accumulate` and the communications epochs `MPI_Win_fence` and `MPI_Win_lock/MPI_Win_unlock`.

The following constants, defined within MPI-2, are also employed by Hydra MPI: `MPI_Mode_nostore`, `MPI_Mode_noput`, `MPI_Mode_nosucceed`, `MPI_Info_null`, `MPI_Mode_noprecede`, `MPI_Lock_shared`, `MPI_Mode_nocheck`, `MPI_Info_null` and `MPI_Address_kind`.

1.3 The Background Technical Details

The code was unpacked and was compiled using the re-entrant compiler, namely `mpxlf90_r`, as the default compiler, namely `mpxlf90`, does not contain any MPI-2 routines.

The code was then run on a range of processor numbers and two data sets were employed. These were as follows:

- a huge, highly-clustered data set, called `d8117.back`, containing 65,536,000 particles.
- a very large, well-clustered data set, called `d6065.1998` containing 524,288 particles.

Hydra_MPI has an internal Initial Conditions Generator (ICG) which creates a smooth, non-clustered data distribution. The ICG was used with $N=65450827$ and 262144 , when determining the initial guess for the tunable parameters, for each of the clustered data sets.

A large number of development tools were investigated for producing performance timings for the code, namely TotalView, xprofiler (and pdbx), MPI_Trace and hpmcount, along with Hydra_MPI's inbuilt timers.

A list of Hydra_MPI's tunable parameters is given in [7] and this report describes the effect of the primary, secondary and tertiary tunables, as categorised in [7].

2 Porting

2.1 Debugging Tools

2.1.1 Using TotalView on HPCx

Throughout the course of this project, extensive use was made of the debugger, TotalView [8]. Etnus TotalView is a debugger for sequential, parallel and threaded programs, and it has a powerful graphical interface. It is available to buy on most parallel supercomputer platforms.

Setting up TotalView in graphical mode needed several prerequisites. Most importantly, the user's session on HPCx must allow X-forwarding, generally achieved using the `-X` option to ssh. Starting TotalView on a parallel job is non-trivial because of the current limitations on HPCx for such programs to run under the control of LoadLeveler. For example, there is an upper level on the number of processors one can debug on. This limit is currently 32 processors. To facilitate this, the `runtv` command is used.

This script runs the debugger as a *poe* job in the appropriate interactive LoadLeveler queue. The command `poe` invokes the Parallel Operating Environment (POE) for loading and executing programs on remote processor nodes. As a result, `poe` command line options cannot be used but must instead be defined as exported environment variables in the user's login script.

For example, to debug Hydra_MPI on 16 processors, a LoadLeveler script must exist for interactive jobs (`tv_debug.ll.16`):

```
#@ job_type = parallel
#@ job_name = hydra_MPI
#@ tasks_per_node = 8
#@ node = 2
#@ node_usage = shared
#@ requirements = (Machine == "l2f35")
#@ class = inter2_1
#@ network.MPI = csss,shared,IP
#@ wall_clock_limit = 00:30:00
#@ account_no = z004-ssp
#@ notification = never
#@ queue
```

The script is then evoked using `runtv tv_debug.ll.16 hydra_mpi`. For more details, see [12]

2.2 Problems and Issues

2.2.1 MPI Trace

In order to gain more detailed information about MPI communication times, we attempted to profile the code using `mpitrace` [9].

Unfortunately, we have found that the `mpitrace` library is not MPI-2 aware. Further, the library can be temperamental as the compilation of the code is sensitive to the location of `-lmpitrace` when linking. Indeed, we were simply unable to invoke the library for `Hydra_MPI`. This was not investigated further due to time constraints.

2.3 The evaluation of HPCx

During the course of the project, a significant reduction in wall clock time per time step was observed: the wall clock time was almost halved. This was due to the underlying communication harnesses LAPI and HAL being updated on HPCx, where LAPI is a low level, efficient one-sided communication API library and HAL (Hardware Abstraction Layer) is an even lower-level interface to the network hardware.

A considerable amount of time was spent debugging the code. We encountered many issues running the code on 64 processors but were unable to investigate further as the current configuration of HPCx only allows users to interactively debug using TotalView for up to 32 processors. Furthermore, when attempting to debug on 32 processors, we occasionally found that there were insufficient resources to run the debugger.

One line in a particular routine, which is only invoked under debugging conditions, causes the code to hang on HPCx. Specifically, if the code is compiled using the preprocessor flag `DEBUG`, then the code hangs at the line

```
icount(int(ref_data(itype_off,i)))=icount(int(ref_data(itype_off,i)))+1
```

of `par_gravsph_noll_ref.F`. For the purposes of this exercise, we simply commented this line out.

3 Profiling

The majority of the times used in the project were gathered using the code's internal timing routines, however, employing other profilers allow the user greater freedom, such as developing a deeper understanding of the structure of the programme itself.

We profiled `Hydra_MPI` using `Xprofiler` [10]. `Xprofiler` is one of the family of IBM XL profilers and is a graphical interface to `prof` and `gprof`.

`Xprofiler` can be used with any program that has been compiled with an IBM XL compiler with the `-pg` flag. This includes MPI programs, as the IBM compile scripts (`mpicc`, `mpxlf`, and `mpCC`) call IBM XL compilers. If the program is in multiple files, only those that are compiled with `-pg` will be profiled. The code that is produced by this re-compilation invokes special runtime libraries.

The code in the libraries causes the Unix `SIGPROF` signal to be sent periodically. It interrupts program execution and jumps to a special subroutine (`.mcount()`) that updates pointers to nodes in a call graph that includes all subroutines that were compiled with `-pg`.

An interrupt handler determines which subroutine is active and its call chain. It updates records of cumulative CPU time for that routine and its parents.

At the end of the job, the invocation count data is written to a file (`gmon.out`). The user can then post-process this profile data with `Xprofiler`, `gprof` or `prof` to produce the final report.

For parallel programs, one `gmon.out` profile file is created per process. These can be viewed individually or as merged data.

N.B., as with all profilers, running a profiler-enabled version of the code causes the code to run slower to some degree.

There are restrictions one should be aware of if using `Xprofiler/gprof` on a parallel code. Profiles of parallel codes on HPCx typically claim that large amounts of time have been spent in routines `kickpipes()` and `readsocket()`. These routines run when the programme is waiting for a message to be sent and/or

received. Indeed, the sum of these routines can give a useful estimate of the total communication cost. However, the statements or routines in the application program that caused `kickpipes()` and `readsocket()` to be called are not identified in the profile. Thus, one cannot trace the source of one's communication overhead.

3.1 Profiling Hydra_MPI

The following Xprofiler table displays the most expensive routines, in terms of computational time, in descending order.

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.par_green	29.4	34.30	34.30	2148266168	0.0000
.kickpipes	14.9	17.37	51.67		
.par_gcnvl	9.6	11.23	62.90	539334164	0.0000
.shgrav	6.1	7.10	70.00	8408523288	0.0000
._sqrt	5.3	6.15	76.15		
._sin	5.1	5.90	82.05	69759322	0.0001
._cos	4.8	5.60	87.65	2554467	0.0022
.par_mesh	4.1	4.81	92.46	13015689770	0.0000
.__mcount	2.5	2.94	95.40		
.update_pm_mesh	2.2	2.62	98.02		
.get_neighbors_noll	2.0	2.30	100.32		
.mpci_recv	1.6	1.86	102.18	7066162384	0.0000
.__par_ref_MOD_par_p	1.5	1.69	103.87	3229182190	0.0000
.z_fft_ldeven	1.2	1.35	105.22	10	135.0
.par_igreen	0.8	0.98	106.20	2148874824	0.0000
._fill	0.8	0.93	107.13		
.refine	0.7	0.83	107.96	8409314958	0.0000
.__par_ref_MOD_set_c	0.7	0.76	108.72	2	380.
.mpci_wait	0.5	0.63	109.35	7063965560	0.0000
.update_boundary	0.5	0.60	109.95	2	300.
.par_imesh	0.3	0.40	110.35	16279741572	0.0000
.group_particles_in_	0.3	0.36	110.71	1	360.
.__itrunc	0.3	0.30	111.01		
.accel	0.2	0.26	111.27	537435009	0.0000
.__mcount	0.2	0.25	111.52		
.gettime	0.2	0.19	111.71		
.sph	0.2	0.18	111.89	540780158	0.0000
.fftwi_no_twiddle_32	0.2	0.18	112.07		

This code profile lists routines which are within Hydra_MPI and without. The following routines are employed by Hydra_MPI: `par_green`, `par_gcnvl`, `shgrav`, `par_mesh`, `update_pm_mesh`, and `get_neighbors_noll`.

Those routines, listed by Xprofiler in the above listing, that are not related to Hydra_MPI, are as follows:

- `kick_pipes` is system routine, called when waiting on communications to complete.
- `_sqrt`, `_sin` and `_cos` are all calls to the mathematical library.
- `__mcount` is an Xprofiler routine, used when time-sampling.

- `mpci_recv` and `mpci_wait` are the primary interface to the point to point message-passing protocols that support the SP Switch and the SP Switch 2.
- `_fill` is currently not understood.
- `_litrunc` is a subroutine that rounds floating-point numbers to signed integers.

Xprofiler produces a graphical depiction of the code calling tree, as shown in Figure 1, which can be extremely useful for a complicated parallel code such as Hydra_MPI.

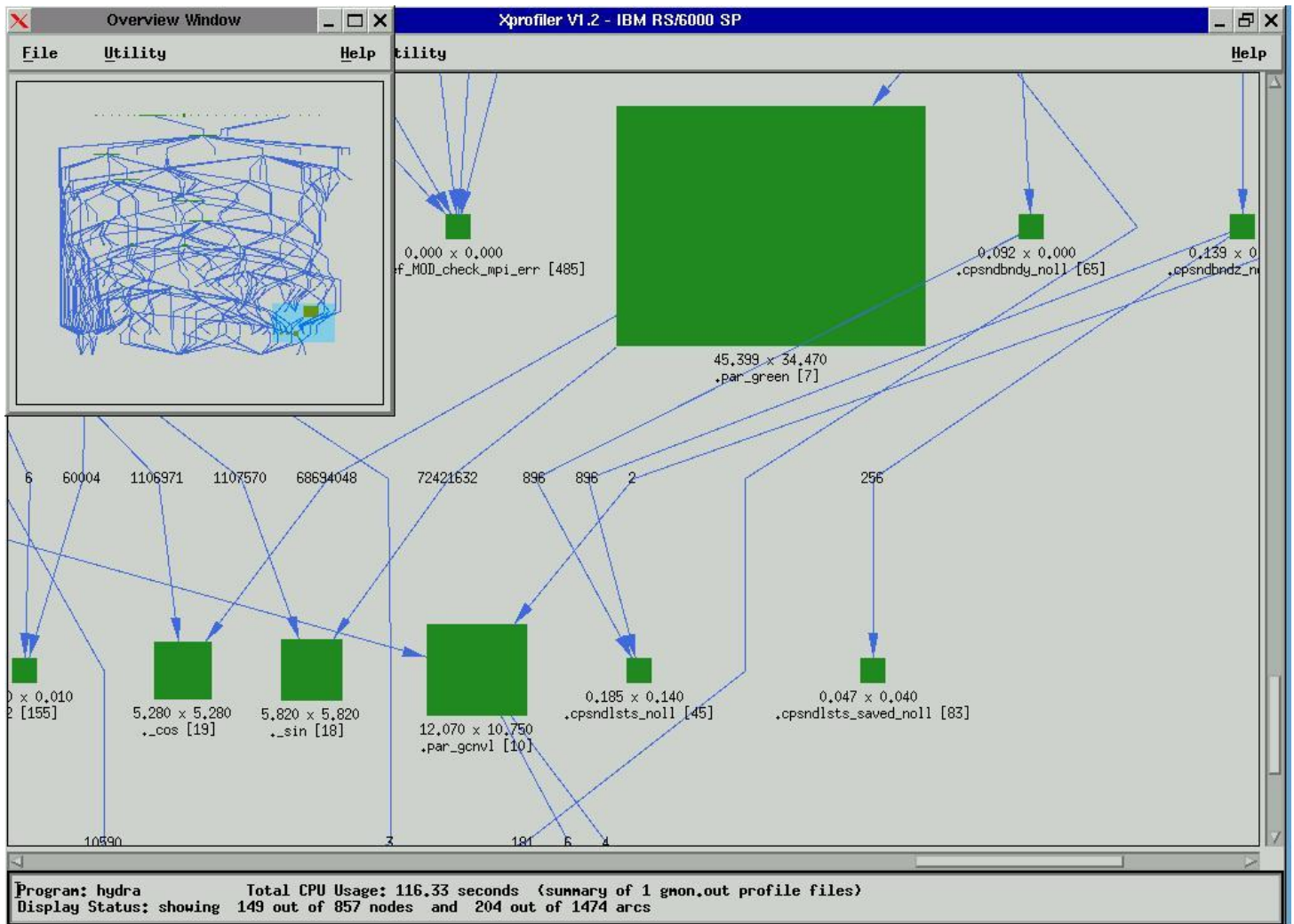


Figure 1: Xprofiler calling tree display for Hydra_MPI

The Overview Window shows the complete hierarchical code display. Dragging the blue box allows one to control the view in the larger Xprofiler Window. The Xprofiler Window shows a box for each subroutine, followed by its inclusive time (time spent in that subroutine plus all subroutines called from it) and exclusive time (time spent in that subroutine, only). The width and height of the box are proportional to these times. Lines between boxes connect callers to callees; the number printed on the line is the number of calls to the subroutine.

3.2 Timing codes on HPCx

Care must be taken when timing codes [11] on HPCx. Firstly, HPCx is a shared resource and producing accurate execution timing figures is therefore difficult.

It was found that timing the wall clock times on Hydra_MPI fluctuated considerably. These fluctuations occur due to other users placing demands on the interconnect, or when system daemons wake up and check to see whether the LPAR is functioning correctly. When we were testing the performance of the code on 16 processors, we were employing two LPARS. The times here were found to fluctuate quite dramatically, since any communications had to be transmitted through the communications' switch. Thus, the speed of communications depended on how busy the switch is, either by MPI messages (belonging to this code or another code running elsewhere on HPCx) and I/O to disk. Further, if a packet is dropped, the message is resent.

4 Introducing self-tuning to Hydra_mpi

As a first step towards a self-tuning version of Hydra_MPI, it was decided to investigate how the execution time was affected by varying Hydra_MPI's numerous variable parameters.

4.1 Tunable variables

This subsection contains a detailed listing of the tunable parameters given in [7] that we examined, along with references to code sections of Hydra_MPI. All of the parameter's optimal values are machine dependent. It should be noted that by varying these particular parameters, we are only varying the refinement placing algorithm.

For each case, the ICG was run to produce a non-clustered distribution of matter and the times for the 2nd time-step were noted. The first time step is ignored as the time includes the time to load the data sets into memory and compute a large array of constants. (The data set (binary) holds information about the velocities and positions of the particles, along with information about their nature, their past positions, their energy, etc.)

The initial guess for each tunable parameter is then formed in terms of the execution times for certain sections of the code and the values of other parameters.

For the following section, we will use the dataset d6065.1998 as an example, running on 8 HPCx processes on one LPAR.

4.1.1 Primary Tunables

Variable	Description	File	T3E Value	New Value
rtp	Cost per particle of the parent PP	par_refine.F	6.00e-7	6.37e-6
rtn	Cost per particle in the refinement PM	par_refine.F	1.00e-5	5.60e-5
rtl	Cost per FFT point in the PM	par_refine.F	1.80e-5	8.74e-7

Table 1: List of primary tunables

In the standard output, the times for `shgrav` and for `mesh` were monitored, where the `shgrav` and `mesh` values are the total times spent performing the top level PP and PM work, respectively. Both these times were divided by N_{par} , where N_{par} is the total number of particles in the simulation.

shgrav / particle / unit time = rtp: 6.37e-6
 mesh / particle / unit time = rtn: 5.60e-5

To determine a default value for `rtl` for the `d6065.1998` data set on HPCx, the time spent on the top level PM work (the mesh time) was divided by the number of points in the top level FFT mesh.

mesh / Number of FFT points = `rtl`: 8.74e-7

We began using these re-calculated values on the datafile `d6065.1998` to determine how sensitive the wall clock time per time step was to changes in the primary tunables. Each parameter was increased and decreased by a factor of ten, and the wall clock time per time step for each adjustment was measured.

It was found that `Hydra_MPI`, running on either 8 and 16 processors, with the data set `d6065.1998`, was insensitive to changes in the primary tunables. For example, the wall clock time per time step for the untuned code on 8 processors was approximately 33.1 seconds, whereas the same time for tuned code was approximately a second quicker at 32.2 seconds. Indeed, the difference here is in the realm of noise, as HPCx is a shared resource.

4.1.2 Secondary Tunables

Variable	Description	File	Value
<code>rtc</code>	Factor of clustering in refinement PP	<code>refine.F</code>	8.0
<code>min_save</code>	Minimum cost saving needed to place a refinement	<code>par_refine.F</code>	0.1

Table 2: List of secondary tunables

The values shown in table 2 are the secondary tunables employed on the Cray T3E-900 [6].

The method of testing used against the primary tunables was also used to investigate the sensitivity of the code to the secondary tunables, namely, the varying of each parameter by a factor of 10.

The dependence of the wall clock time on the secondary parameters was found to be negligible for this data set, namely `d6068.1998`. It should be noted that this is unlikely to be the case for every data set as the level of clustering will not be constant.

4.1.3 Tertiary Tunables

Variable	Description	File	Value
<code>default_thresh</code>	A starting value for the threshold	<code>refine.F</code>	0.16
<code>THRESH_MEAN</code>	Attempts to cater for sub-refinements	<code>par_refine.F</code>	defined
<code>PROMOTE</code>	Reduces the number of single processor refinements	<code>par_refine.F</code>	defined
<code>maxtrial</code>	Maximum number of placing trials	<code>refine.F</code>	20
<code>allow_splitting</code>	Allow refinements to be split	<code>par_refine.F</code>	.TRUE.
<code>max_asym</code>	Maximum asymmetry for splitting	<code>par_refine.F</code>	3
<code>min_split</code>	Minimum width of ref before splitting occurs	<code>par_refine.F</code>	6
<code>min_pop</code>	Minimum fraction of above thresh cells in a refinement before splitting occurs	<code>par_refine.F</code>	0.25

Table 3: List of tertiary tunables

It was found that varying the tertiary tunable parameters, from their default (T3E) values, had no effect on performance for the d6065.1998 data set. (Again, the numerical tunables were increased and decreased by a factor of 10; those tunables which were set as defined were set as undefined; and the `allow_splitting` tunable was set as `.FALSE.`)

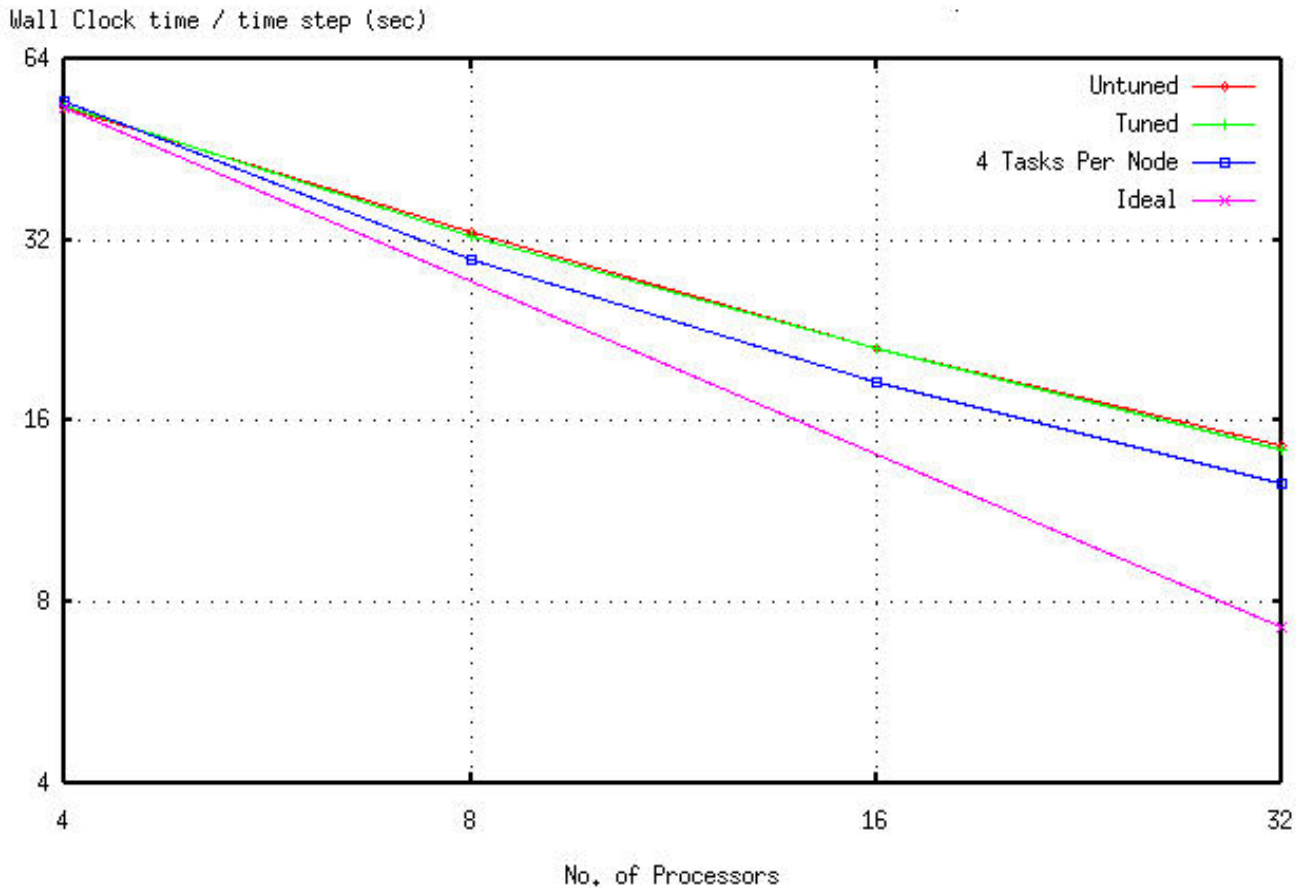


Figure 2: Comparison of Tuned and Untuned code for data set d6065.1998

The difference between the times for the tuned and the untuned codes can be seen from this graph. There is very little difference between them. This is probably due to the fact that the code is not especially sensitive to any of the modified parameters.

The line showing “4 tasks per node” denotes the execution time when running on only 4 nodes per LPAR, rather than the full 8 nodes, i.e. each node runs only 4 MPI processes on its 8 processors. The execution time is faster than for fully populated LPARs, however, it is not a cost effective saving as the user pays for full LPARs, and the time saved is not more than 50%.

It is interesting to note that, when running 8 processes on 8 processors, we saw around 20 MPI threads running on each LPAR. These extra threads are associated with the RMA routines.

The tuned code is approximately a factor of two slower than the ideal time for 32 processors.

5 Conclusions

We had hoped to port Hydra.MPI to HPCx and investigate self-tuning. Due to time constraints, the code was not fully ported, i.e. some workarounds were introduced to allow the investigation of self-tuning to begin timeously.

Unfortunately, the initial investigation showed that the so-called tunable parameters had little to no effect on the code's performance.

5.1 Future Work

It would be of interest to list all the particle refinements in order of size: currently, all large refinements are handled by all the processors and all small refinements are handled by single processors. There are some refinements which are neither large nor small, and they are computing using $\sqrt{N_{\text{pes}}}$ processors, where N_{pes} is the total number of processors. An investigation of what exactly the size of this intermediate refinement is, and what the optimum number of processors for job is, remains to be performed.

As 5.3% of the code was spent calling the square root function, it would be a good idea to review the use of the square root with a view to removing the computation.

Another interesting thing to investigate would be whether the wall clock time per time step increased when the code was using 7 of the 8 processors in an LPAR. When running the code on 8 processors, the code must share the LPAR with the operating system. However, this overhead is a constant time penalty. If the code employs seven processors, the operating systems daemons are free to run on the one remaining processor, which may well improve the code's efficiency. One drawback of this approach is that the user is charged for all eight processors, however, the overall execution time may be reduced.

Hydra_MPI appears to hang for a large number of HPCx processors (64+) for more than 2 levels of refinement for the large data set only. This bug was not found due to time restrictions. It could be that, since this code has run successfully on many other environments, that the bug lies in IBM's MPI-2 RMA implementation, however, this remains to be proven.

References

- [1] <http://www.epcc.ed.ac.uk/t3e/virgo>
- [2] "Towards a portable, fast parallel AP³M-SPH code: Hydra_MPI", Pringle, G. J., Booth, S. P., Couchman, H. M. P., Pearce, F. R., Simpson, A. D., Euro PVM-MPI 2001, Santorini (Thera) Island, Greece, 23-26 September 2001
- [3] "Applied Numerical Libraries for Parallel Software", Antonioletti, M., Darling, G., Ashby, J. V., Allan, R. J., UKHEC Technical Report, 2002 <http://www.ukhec.ac.uk/publications/reports/numlib.pdf>
- [4] <http://www.fftw.org>
- [5] <http://star-www.dur.ac.uk/~frazierp/virgo>
- [6] <http://www.epcc.ed.ac.uk/t3e>
- [7] "How to tune the refinement placing of Hydra_MPI for each new platform", Pringle, G. J., VIRGO internal technical report, 4 July 2002
- [8] <http://www.etnus.com>
- [9] <http://www.hpcx.ac.uk/support/documentation/IBMdocuments/mpitrace>
- [10] <http://www.pdc.kth.se/training/Talks/Performance/ParallelPerfTools/xprof.more.html>

[11] <http://www.hpcx.ac.uk/support/FAQ/timing.txt>

[12] <http://www.hpcx.ac.uk/support/FAQ/totalview/>