

Terascaling techniques on HPCx

S. Booth, A. Jackson

April 15, 2005

1 Introduction

One of the key objectives of HPCx is to support Grand-challenge science and to allow UK academics to perform computations that are larger than can be performed on any other platform that they have access to. This will only be possible if the simulation codes they use can efficiently utilise large numbers of processor and achieve aggregate performance in Teraflops rather than Gigaflops. The difficult process of optimising a code to efficiently use large numbers of processors is called “Terascaling”. This report reviews a number of Terascaling techniques that might be used to improve the scaling of application codes on HPCx and other systems with a similar “Clustered SMP” architecture.

As with all types of optimisation Terascaling requires the implementation of a code to be adapted to make the best use of the capabilities of the target platform. It is therefore important to have a good understanding of the hardware architecture of the target platform.

2 Architecture

HPCx like many current HPC systems has a clustered SMP architecture. It is constructed out of nodes connected by a communication network where each node contains multiple processors attached to a shared memory system. It is therefore a Hybrid of two more traditional HPC architectures

- Distributed memory, where each processor has its own address space.
- Shared memory, where all processors have access to a single shared address space.

The current popularity of the clustered SMP architecture is essentially due to cost and pragmatism. Though HPC systems are individually quite large the global market for HPC systems is relatively small so there are potential cost saving if HPC systems are built out of off the shelf mass market components.

Shared memory systems are difficult to build and this gets harder the more processors the system contains. Outside of the HPC community there is very little commercial demand for shared memory systems with more than about a hundred processors.

Large distributed memory systems are potentially easier to build but when they are constructed out of commodity parts then large numbers of network cables are required to connect them together and this introduces practical difficulties to building large systems.

A clustered SMP can have a large total processor count and still have a manageable number of nodes each of which is a high volume product. The disadvantage of this approach is that the architecture of the system is more complicated and therefore it can be much harder to understand the scaling behaviour of programs running on this type of system.

2.1 HPCx

HPCx is a clustered SMP system built out of 50 IBM P690+ SMP systems connected by the IBM “High performance switch”. Each P690+ contains 32 Power4+ processors. These processors come in pairs (There are two processors per chip). Each processor has its own Level-1 cache but Level-2 is shared by the pair. This shared Level-2 cache is a 1.5 Mb 8-way associative cache divided into three 512Kb sections. The Level-2 cache has 128 byte cache lines. Four chips (8 processors) form a Multi-Chip-Module (MCM), with all the processors in a MCM sharing a Level-3 cache. The Level-3 cache is also 8-way associative but has 512 byte cache lines and a capacity of 128Mb.

On HPCx each node has two connections to the switch fabric which is organised as a omega-like network with three levels of switch hierarchy.

3 Programming models

HPCx and similar systems support a number of programming models. The primary model is *message passing*. In this model the calculation is divided into a number of communicating processes (Units of execution each with their own separate address space). These processes communicate by making calls to a *message passing library* like MPI. These message passing calls essentially copy the required data from the address space of the source process to somewhere in the address space of the destination process. This is the only way that processes on different nodes can communicate.

Another important programming model is *shared-memory* programming. In this model the entire program is a single process with a single address space. It is still possible to use multiple CPUs by having multiple *threads* of execution running within the process. The *OpenMP* language extensions are probably the easiest way of writing shared memory programs. Shared memory programming is only possible within a single node.

It is also possible to allocate shared-memory *segments* on a node which are regions of memory that exist independently of the existence of any process and multiple processes can map the segment into their own address space to create regions of shared memory in addition to their normal private address space.

As shared memory techniques can only be used within a node, applications that need to utilise a very large number of processes need to use message-passing or a hybrid model where message passing is used between nodes and some shared memory techniques are used within a node.

3.1 Tools and Techniques

Profiling and analysing applications is the cornerstone of terascaling applications. Without a good understanding of the performance characteristics of the program you are trying to optimise, it will be difficult to make any performance improvements.

This can only be achieved through the use of Profiling and Performance Analysis tools that allow you to discover the sections of code that take the largest proportion of the execution time of an application, and therefore which areas of the program it would be beneficial to focus optimisation effort on. It is more efficient to make a small improvement in the section of code that dominates the runtime of an application, than to double the performance of a section of code that only accounts of a small amount of time.

There are a number of generic, and IBM specific, tools available on HPCx and they are described in the HPCx User Guide[1], under the “Tools” section.

There are two main areas to profile for any code aiming to be terascaled, serial performance, and parallel performance. The serial performance of a program can be analysed using profiling tools such as gprof and xprof, and IBM’s HPM¹ toolkit. These tools will provide information on execution count and time for functions and sections of code within an application, the cache and memory usage of a code, and other hardware metrics. This information can then be interpreted to identify bottlenecks within an application, and suggest methods for improving performance.

There are also a set of tools that will provide information on the parallel performance of a program, VAMPIR and MPI Trace being the most useful. VAMPIR is a profiling tool which allows users to record the MPI communications associated with an application, and view the data using a graphical user interface. MPI Trace is a low-level tool which will produce detailed profiling information on all MPI calls made within an application. There is even a MPIHPM library that combines the functionality of MPI Trace with that of HPM.

The details of using the HPM toolkit are discussed in the technical report HPCxTR0307 [2], and various specific examples of applications being profiled using the tools available on HPCx are included in the following technical reports: HPCxTR0410 [3], HPCxTR0413 [4], HPCxTR0414 [5], HPCxTR0415 [6].

¹Hardware Performance Monitor

4 Causes of poor scaling

The aim of parallel programming is to improve the time to solution of a calculation by decomposing the problem into a number of tasks that can be performed in parallel by utilising multiple CPUs, memory-systems etc. at the same time. Unless these tasks are completely independent they will need to exchange data (communication) at various points in the calculation. This communication always has some cost associated with it. In addition it introduces synchronisation between the tasks. If a task has reached the stage where it needs a particular result but the task that produces that result has not yet made it available then the first task must sit idle waiting for the data.

We can therefore identify a number of common causes for lack of scalability in parallel programs:

- Lack of parallelism.
- Communication overheads.
- Poor Data decomposition or load balance
- Contention for resource.

we will look at some of these issues in greater detail in the following sections.

In addition to understanding how scaling problems occur it is also important to understand how their significance varies with the number of processors. For example on many systems it is possible to characterise the cost of sending a message using a latency/bandwidth model. In this model the cost of sending a message is proportional to the sum of two contributions.

1. A constant cost independent of the size of the message. This is normally called the message **latency**.
2. A cost proportional to the size of the message being sent.

For many codes the number of messages stays constant or increases as the number of processors is increased but the typical size of message reduces. Therefore the relative importance of the two types of overhead changes.

On the other hand with some types of simulation it is more productive to scale the simulation by increasing the size of the problem being simulated rather than trying to run a fixed size simulation in a shorter time. This problem size scaling is usually easier to achieve. If the communication patterns of applications are relatively local then the size and number of messages will stay roughly the same as the problem size and number of processors are increased. This will in turn result in good scaling behaviour as the communication overhead also remains roughly constant.

4.1 Lack of parallelism

Any significant stage of the program that is only performed on a single processor or that is calculated redundantly on all processors will result in a lack of scalability.

This is the effect described in Amdahls law:

“The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial”

More specifically if a fraction α of the run-time is completely serial then the speedup is limited to $\frac{1}{\alpha}$ no matter how many processors are used.

Redundantly calculated sections are slightly better than sections that only utilise a single processor because there is no need for additional communication steps to re-broadcast the result. However any but the very smallest serial sections can seriously impair the scalability of a code.

All codes will have some fraction that is essentially serial even if it is only the overhead of making a subroutine calls.

4.2 Decomposition and load imbalance

The parallel decomposition used in an application can have a big effect on its scalability. The aim is to give an equal amount of work to each processor so that they all complete their allotted tasks at the same time without lightly loaded processors having to wait for the heavily loaded ones. This is relatively straightforward when the available parallel tasks all take the same time to complete. All that is required is for each processor to be given an equal number of tasks. Things become more complicated when the tasks are not equivalent. When the cost of the tasks can be estimated in advance it is possible to construct a static decomposition scheme to balance out the available work. A good example of this kind of static load balancing technique can be seen in the HPCx technical report: HPCxTR0415 [6]

This report is a case study of the optimisation of a coastal ocean modelling code where the load imbalance results from the geography of the area being simulated.

On clustered SMP systems like HPCx there is an additional factor to consider when choosing a decomposition strategy. Communications within a node are usually significantly faster than communications between nodes so the time taken to complete a communication step may depend on the decomposition strategy. For example if an application uses a two dimensional decomposition but has a large number of communication steps that only communicate in the vertical direction then it makes sense to decompose the vertical dimension within a node and the horizontal direction across nodes.

4.3 Mixed mode codes

While it is not possible to use a full shared memory programming style on clustered SMP architectures it is possible to adopt a hybrid style where some

shared memory techniques are used within a node and message passing is used to communicate between nodes.

The hope is that by writing such a “mixed-mode” code it should be possible to take maximum advantage of the clustered SMP architecture.

The HPCx technical report: HPCxTR0403 [7] investigates this approach. Unfortunately it turns out to be quite difficult to make a good case for recommending mixed-mode programming.

One of the biggest advantages of shared memory programming models like OpenMP is their relative ease of use compared with message passing. This advantage is not shared by mixed mode codes as it is effectively necessary to parallelise the program twice once using OpenMP and once using message passing.

Within a SMP node message passing is usually implemented by copying data too and from a shared memory segment. This is usually significantly faster than off-node communications but introduces additional memory copies compared to the direct reads and writes to fully shared data. However much of this advantage is lost in mixed mode codes because off-node communication tends to be slower than in the pure-MPI case because the data being communicated is less likely to be in the cache of the processor making the MPI communication calls, resulting in costly cache misses.

4.4 Communication overheads

The performance of any communications within a parallel program can have a large impact on performance, and the larger the number of the processors used, the greater the impact.

The majority of parallel applications on HPCx use IBM’s MPI library to perform all their communications. This library has been optimised for both the shared-memory and switch communication hardware used in HPCx, and will give better performance than public domain MPI libraries.

MPI communications can be a bottleneck for parallel applications, especially when using large numbers of processors, as standard MPI functions can impose unnecessary synchronisation, and synchronisation is one of the main constraints on tera scaling. There are two main kinds of communication functions in MPI; point-to-point functions, and collective functions. Point-to-point functions enable one process to send a message (i.e. some data) to another process (i.e. there are only two processes involved in a point-to-point communication). Collective communications involve all processes, and can be operations such barriers (global synchronisation), broadcasts (one process sending data to all processes), or other more complex group behaviour.

Collective communications are very costly operations, from a time point of view, as they enforce global synchronisation (i.e. all processes have to wait for the slowest process). This means they should only be used where really necessary. In particular, the barrier operation should not be used unless it is explicitly needed, and collective operations to compute “global values” (i.e. data about the simulation, or program, as a whole) should only be performed when that

calculated data is to be used (either by the program or as part of some data output). It is common for many applications to compute global values to check that the application is proceeding correctly when the application is being developed, but these checks should be disabled for production runs of the applications unless they are necessary for the output of the program or in the operation of the program. On the other hand if an application requires a collective operation the standard collectives in the MPI library should usually be used in preference to hand crafted routines built out of point to point communication. This is because the the MPI standard is written in such a way to allow many kinds of optimisation in the collective calls that cannot be used for individual point to point communications.

The simplest MPI point-to-point optimisation that can be performed is replacing a series of blocking MPI communication calls with their non-blocking equivalents running in parallel. This technique, discussed in Technical Report HPCxTR0308 [8], is particularly effective where there are regular, common, communication patterns (i.e. when, for instance, each process performs halo or boundary data swapping). Replacing blocking function calls with non-blocking function calls enables processes to perform a number of communications without having to specify what order the communications will take place. Thus if each process is communicating with two other processes, it can send both it messages straight away. Then it can wait for both the messages it has sent to “complete” (i.e. finish sending). Whichever peer processor becomes ready first is then able to receive its message immediately. If blocking communication functions were used for this communication pattern the process would have to wait for the first communication to complete before starting the next communication, which could be inefficient if it was the second peer process that became available for communication first. The same argument applies when a process has to receive data from many sources.

As mentioned previously, the MPI library on HPCx makes use of both shared-memory, and switch, communication methods. If a process is sending a message on the same node as itself then the message is sent purely through memory. If it is sending a message “off-node” (i.e. to a process on another node) it is sent through the switched network. This optimisation is important as on-node memory communications are considerably fast than off-node switched communications. The latency of a message on-node is approximately $3.4e^{-06}$ seconds, whereas the off-node latency is approximately $5.5e^{-06}$ seconds.

This difference in communication latencies can be used to optimise some collective communications, such as `MPI_Alltoall` and `MPI_Alltoallv`. The AlltoAll operation involves all processes communication with all other processes, thus it is a very costly operation. By introducing additional communications within a node we can reduce the number of messages that have to pass between nodes. The same amount of data has to moved in both cases so the messages are larger as well as being fewer in number. In the case of `MPI_Alltoallv` this optimisation has to be performed explicitly. The MPI library cannot do this for you as it does not know beforehand what the global communication pattern is. Two different methods for performing this optimisation are discussed in

technical report HPCxTR0401 [9] and HPCxTR0409 [10].

4.5 Contention

Another potential cause of scaling problems is contention for some shared resource. On clustered SMP systems there are two main resources that are shared by processors.

1. The memory system within a node
2. The interconnect fabric connecting nodes.

In an SMP system all the processors within a node share the main memory system, caches are usually either dedicated to a single processor or shared by a small subset of the processors. The cache utilisation of an application can therefore have an impact its scalability within a SMP node. If the majority of memory accesses are from cache then most memory accesses are being handled by independent hardware. If a large fraction of the memory accesses need to go to main memory then the CPUs are competing for the memory bandwidth available from main memory and scalability will be impaired. This can be particularly bad in shared memory applications if the CPUs are competing for access to the same data.

Similar contention problems can occur in the interconnect fabric. For example if all CPUs in a node are attempting to perform inter-node communications at the same time then there usually some aggregate bandwidth limit on each node that results in each individual communication running more slowly than it would have done if performed in isolation.

4.6 Libraries

Scientific applications use libraries for all aspects of their working, from I/O to Eigensolvers, and FFTs to communications. These library functions, especially some of the numerical subroutines, can constitute a large portion of a programs run time, so it is important to ensure that the most efficient libraries and routines are being used.

On HPCx, there are a number of IBM specific libraries that have been created and optimised for IBM hardware. These include the ESSL² and PESSL³ [11] libraries, which contain a subset of the LAPACK[12] and ScaLAPACK[13] libraries respectively, and the BLACS⁴ and MASS libraries. The MASS library provides highly-optimised “mathematical intrinsics”, and can be used by linking `-lmass` before the standard maths library `-lm`.

ESSL and PESSL contain linear algebra, eigensystem, and fourier transform functions, as well as a number of other computation subroutines.

²Engineering and Scientific Subroutine Library

³Parallel Engineering and Scientific Subroutine Library

⁴Basic Linear Algebra Communications Subroutines

In general, the IBM libraries should provide better performance than the equivalent public domain libraries, although not all the public domain functions have been reimplemented by IBM. However, it is possible to use both libraries, picking up the optimised routines from the IBM libraries, and using the public domain libraries where routines haven't been implemented by IBM (see the HPCx User Guide on how to link with LAPACK, ScaLAPACK, and their IBM equivalents ESSL, and PESSL).

There are also some public domain libraries available on HPCx that give good performance, including FFTW[14] which contains highly optimised Fourier transform routines for both serial and parallel transforms. These are discussed in Technical Report HPCxTR0303 [15]. Similarly, there are a number of libraries that provide Eigensolving function for both the General and Standard problems. These include the PLAPACK and PARPACK libraries, as well as the functionality provided by LAPACK and ScaLAPACK. Eigensolvers, and their performance on HPCx, are discussed in technical reports HPCxTR0312 [16], and HPCxTR0406 [17].

It may be that, as well as ensuring that a user is using the most efficient library implementation of a function for their code, the user may have to alter their code to use a different function to gain improved performance and scaling. This maybe moving from serial FFTs or Eigensolvers to parallel versions of those functions, or changing their data decomposition to be able to use a particular Eigensolver. Such changes may need to be made if the performance and scaling of an application is greatly influenced by a single function, or small section of serial code.

References

- [1] "User Guide for the HPCx Service"
<http://www.hpcx.ac.uk/support/introduction/index.html>.
- [2] HPCxTR0307 "Using the Hardware Performance Monitor Toolkit on HPCx", Joachim Hein
- [3] HPCxTR0410 "Improved parallel performance of SIESTA for the HPCX Phase2 system", Joachim Hein
- [4] HPCxTR0413 "Profiling H2MOL on an IBM p690+ Cluster", Lorna Smith, Mark Bull, Andrew Sunderland
- [5] HPCxTR0414 "Optimisation of VASP on HPCx", Gavin J. Pringle
- [6] HPCxTR0415 "Optimization of the POLCOMS Hydrodynamic Code for Terascale High-Performance Computers", Mike Ashworth, Jason T. Holt and Roger Proctor.
- [7] HPCxTR0403 "Mixed Mode Applications on HPCx", Jake Duthie, Mark Bull, Lorna Smith

- [8] HPCxTR0308 “Exchanging multiple messages via MPI”, Joachim Hein, Stephen Booth, Mark Bull
- [9] HPCxTR0401 “An LPAR-customized MPI_AlltoAllV for the Materials Science code CASTEP”, Martin Plummer and Keith Refson
- [10] HPCxTR0409 “Planned AlltoAllv: A Cluster Approach”, Adrian Jackson, Stephen Booth
- [11] IBM Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL <http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html>
- [12] Lapack <http://www.netlib.org/lapack/>
- [13] ScaLapack <http://www.netlib.org/scalapack/>
- [14] FFTW <http://www.fftw.org/>
- [15] HPCxTR0303 “3D FFTs on HPCx (IBM vs FFTW)”, Adrian Jackson, Gavin J. Pringle.
- [16] HPCxTR0312 “An overview of Eigensolvers for HPCx”, Elena Breitmoser, Andy Sunderland
- [17] HPCxTR0406 “A Performance Study of the PLAPACK and ScaLAPACK Eigensolvers on HPCx for the Standard Problem”, Elena Breitmoser, Andy Sunderland