CRESTA

SEVENTH FRAMEWORK PROGRAMME

D2.5.1 –Fault agnostic and asynchronous algorithms at exascale

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M21
Submission date:	30/06/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	UEDIN
Version:	1.0
Status	Final
Author(s):	Mark Bull (UEDIN), Jeremy Nowell (UEDIN)
Reviewer(s)	Jens Doleschal (TUD), Frédéric Magoulès (CRSA)

Dissemination level	
PU	*PU - Public*

© CRESTA Consortium Partners 2011

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	31/05/2013	Initial draft for review	Mark Bull (UEDIN)
0.2	20/06/2013	Revised draft after internal review	Mark Bull (UEDIN)
1.0	26/06/2013	Final document for submission to EC	Mark Bull (UEDIN)

© CRESTA Consortium Partners 2011

Table of Contents

1 EXECUTIVE SUMMARY 4
2 INTRODUCTION 5
2.1 PURPOSE 5
2.2 ASYNCHRONOUS ALGORITHMS FOR SPARSE LINEAR ALGEBRA 5
2.3 PERFORMANCE FAULTS 5
2.4 RELEVANCE TO CRESTA APPLICATIONS 6
2.5 GLOSSARY OF ACRONYMS 6
3 ASYNCHRONOUS JACOBI AND BLOCK JACOBI ALGORITHMS 7
3.1 JACOBI ALGORITHM 7
3.2 BLOCK JACOBI 9
4 SIMULATING PERFORMANCE FAULTS 11
4.1 SLOW CORES 11
4.2 SLOW LINKS 11
5 RESULTS 12
5.1 SLOW CORES 12
5.1.1 Jacobi algorithm 12
5.1.2 Block Jacobi 14
5.2 SLOW LINKS 16
6 CONCLUSIONS AND FUTURE WORK 19
7 REFERENCES 20

Index of Figures

Figure 1 Pseudocode for parallel Jacobi 8
Figure 2 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6 12
Figure 3 Execution time for Jacobi algorithm on 128 nodes of a Cray XE6 13
Figure 4 Residue vs. time for Jacobi algorithm on 1024 nodes (32768 cores) of a Cray XE6 14
Figure 5 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 32 blocks 14
Figure 6 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 32 blocks and conservative residual estimation 15
Figure 7 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 8 blocks and conservative residual estimation 16
Figure 8 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 2 blocks and conservative residual estimation 16
Figure 6 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6 with one slow link 17
Figure 7 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6 with all slow links 18

© CRESTA Consortium Partners 2011

1 Executive Summary

The number of parts in HPC systems is set to increase significantly as their performance approaches the Exascale. This means that fault tolerance in an increasingly important aspect of the design of these systems. However it is also possible to consider software-hardware co-design as a solution to these problems. On the software side, this includes the development of fault tolerant algorithms. In general, this is a difficult problem, especially if faults are considered where part of the current state of a computation is lost. Other types of fault, however, do not involve such state loss: these include performance faults where a component (e.g. a processor or network link) does not fail, but performs at a slower rate than intended. Such faults are less catastrophic, but may be harder to detect.

Performance faults may not cause the computation to fail, but, for many algorithms, the synchronisation patterns mean that the whole computation can run at the speed of the slowest component. Asynchronous algorithms, which are often derived from synchronous counterparts by relaxing some or all of the synchronisation requirements, have the possibility of being much more tolerant to performance faults, though likely at the expense of poorer convergence rates.

In this deliverable, we select two asynchronous algorithms for the solution of large sparse linear systems (Jacobi and block Jacobi), and, using simulated slow cores and slow links on a real HPC system, quantify their ability to maintain performance in the presence of such faults by comparing them to their synchronous counterparts.

Our findings do indeed show that the algorithms have strong resilience to such faults, even when the loss of component performance reaches an order of magnitude. However, in some cases we observe that the asynchronous algorithms can exhibit undesirable convergence behaviour, and that care needs to be taken to avoid this. Finally, we discuss how such algorithms may be of interest in the contexts of alternative uses of fat nodes and power management.

© CRESTA Consortium Partners 2011

2 Introduction

2.1 Purpose

The purpose of this deliverable is to explore the ability of asynchronous algorithms for solving large sparse linear systems to tolerate performance faults such as slow running cores and slow running network links. The rest of the document is organised as follows: The remainder of this section introduces asynchronous algorithms and performance faults, and discusses the relevance of this work to the CRESTA applications. Section 3 gives details of the two asynchronous algorithms we use for this investigation, and Section 4 describes the techniques used to simulate performance faults. Section 5 presents and discusses the experimental results, and finally Section 6 draws some conclusions and points to future work.

2.2 Asynchronous algorithms for sparse linear algebra

Modern high performance computing systems are typically composed of many thousands of cores linked together by high bandwidth and low latency interconnects. Over the coming decade core counts will continue to grow as efforts are made to reach Exaflop performance. In order to continue to exploit these resources efficiently, new software algorithms and implementations will be required that avoid tightly-coupled synchronisation between participating cores and that are resilient in the event of failure.

In this deliverable we investigate one such class of algorithms. The solution of sets of linear equations $A\boldsymbol{x} = \boldsymbol{b}$, where A is a large, sparse $n \times n$ matrix and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors, lies at the heart of a large number of scientific computing kernels, and so efficient solution implementations are crucial. Existing iterative techniques for solving such systems in parallel are typically highly synchronous, in that all processors must exchange updated vector information at the end of every iteration, and the algorithm may require scalar reductions. This creates barriers beyond which computation cannot proceed until all participating processors have reached that point, i.e. the computation is globally synchronised at each iteration. Such approaches are unlikely to scale to millions of cores, and are highly sensitive to performance faults or glitches: a delay in one processor or in one communication link may delay the entire computation.

Asynchronous algorithms avoid this blocking behaviour by permitting processors to operate on whatever data they have, even if new data has not yet arrived from other processors. Note that the term asynchronous is used here in a strong sense: the convergence behaviour of the algorithms can differ from their synchronous counterparts: it is not just a question of relaxing the order of computation and communication and still respecting data dependencies. To date there has been work on both the theoretical [1], [2], [3] and the practical [4], [5], [6] aspects of such algorithms. The use of asynchronous techniques in large, tightly coupled parallel systems of relevance to Exascale computing has been the subject of recent work in [7] [8] and [9]. Asynchronous methods have been shown to work well in distributed heterogeneous environments [10]: in this study we are considering the question of how they behave in a homogeneous system with performance faults, which can be considered a special case of a heterogeneous system.

2.3 Performance faults

Recent work on fault tolerant algorithms for scientific computing focuses on the relevant, but difficult, case of total component failure with accompanying data loss. While some useful progress has been made, for example in the field of dense matrix computations [11], applicability is not widespread and the solutions may still rely in part on some form of checkpointing. It is also not clear how asynchronous algorithms might be exploited in this scenario, since the notion of the last known good state may not be well defined.

In this study, we consider the case of performance faults. In this case, no data is lost, but components may fail to deliver their intended performance. For example, a CPU

© CRESTA Consortium Partners 2011

core, or an entire node, may still be working, but compute at slower rate than all the others in the system. This can occur not only due to hardware problems, but also to software issues such as rogue threads or processes. Alternatively, processes from different applications may be deliberately co-scheduled on the same node to maximise the use of resources, but the impact of this on the timing behaviour of applications may vary from one node to another.

Increasingly, modern processors have power-saving features that alter the clock-rate, for example if the device overheats, and mis-operation of these can be source of such faults. If such power-saving features are deliberately enabled, it may be difficult to synchronise their behaviour across multiple nodes. Performance faults can also occur in networks, where a link may deliver higher latency and/or lower bandwidth than normal.

As is the case with component failure, performance faults are likely to become more common as the size of systems increases. They are, in general, difficult to detect and therefore studying algorithmic approaches that can tolerate them is a useful area of research.

2.4 Relevance to CRESTA applications

Sparse linear solvers are widely used in a variety of HPC applications. A recent PRACE report highlighted its importance to the European HPC community reports the usage of computational paradigms across application areas. Sparse linear algebra was found to be used across the whole range of HPC applications areas, but is particularly heavily used in the Astronomy and Cosmology community, Computational Chemistry and Computational Fluid Dynamics.

Within CRESTA, sparse linear algebra is important in two of the co-design applications. The first, ELMFIRE, is a full-f gyrokinetic plasma simulation code, which at exascale is looking to simulate plasmas in the next generation fusion reactors such as ITER. The second, OpenFOAM, is an open-source computational fluid dynamics application, widely used in both academia and industry to study a large range of problems. Exascale problems under consideration include Large Eddy Simulations of turbulence in moving turbomachinery. It is clear therefore that any improvements to the solvers within these codes will have a direct impact on helping their goal of running at the exascale.

2.5 Glossary of Acronyms

AIAC	Asynchronous Iterations Asynchronous Communications
D	Deliverable
EC	European Commission
HPC	High Performance Computing
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
SISC	Synchronous Iterations Synchronous Communications

3 Asynchronous Jacobi and block Jacobi algorithms

3.1 Jacobi algorithm

Jacobi's method for the system of linear equations $Ax = \boldsymbol{b}$, where A is assumed to have nonzero diagonal, computes the sequence of vectors x^k, where

$$x_i^k = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}\, x_j^{k-1}\right) \quad i = 1, \dots, n$$

The $x_i^k, i = 1, \dots, n$ are independent, which means that vector element updates can be performed in parallel. Jacobi's method is also amenable to an asynchronous parallel implementation in which newly-computed vector updates are exchanged when they become available rather than by all processors at the end of each iteration. This asynchronous scheme is known to converge if the spectral radius $\rho(|M|) < 1$ with $M = -D^{-1}(L + U)$ where D, L and U are the diagonal and strictly lower and upper triangular parts of A. In contrast, the synchronous version of Jacobi's method converges if $\rho(M) < 1$ [3].

To investigate the behaviour of Jacobi's method in the presence of performance faults, we have implemented both synchronous and asynchronous variants of the algorithm. The synchronous version falls into the SISC (Synchronous Iterations Synchronous Communications) classification proposed by Bahi et al. [12], i.e. all processes carry out the same number of iterations in lock-step, and communication does not overlap computation, but takes place in a block at the start of each iteration. The asynchronous version is AIAC (Asynchronous Iterations Asynchronous Communications), since processes proceed through the iterative algorithm without synchronisation, and so may iterate at different rates depending on a variety of factors. Communication may take place at each iteration, but is overlapped with computation, and crucially, the receiving process will continue iterating with the data it has, incorporating new data as it is received. We note that these schemes are more general than simply overlapping communication and computation within a single iteration (e.g. using MPI non-blocking communication) as messages may be (and in general, will be) received at a different iteration to which they are sent, removing all synchronisation between processes. In the asynchronous version we ensure that a process only reads the most recently received complete set of data from another process, i.e., for each element x_i^k received from a particular process we ensure that all such x_i read during a single iteration were generated at the same iteration on the sender. This restriction could be relaxed, allowing processes to read elements x_i potentially from multiple different iterations. As is shown in [7], this reduces communication overhead, but relies on the atomic delivery of data elements to the receiving process, so that every element we read existed at some point in the past on a remote process.

Instead of a general Jacobi solver with explicit A matrix, we have chosen to solve the 3D diffusion problem $\nabla^2 \mathrm{u} = 0$ using a 6-point stencil over a 3D grid. This greatly simplifies the implementation, since there is no load-imbalance, nor are complex communication patterns needed, and it allowed us to easily develop multiple versions of our test code. In all cases, we have fixed the grid size for each process at 50^3, and so as we increase the number of participating processes the global problem size is weak-scaled to match. The boundary conditions for the problem are set to zero, with the exception of a circular region on the bottom of the global grid defined by $e^{-((0.5-x)^2+(0.5-y)^2)}$, where the global domain is $0 \leq x, y, z \leq 1$. Physically, this can be thought of as a region of concentrated pollutant entering a volume of liquid or gas, and we solve for the steady state solution as the pollution diffuses over the region. The interior of the grid is initialised to zero at the start of the iteration, and convergence is declared when the $\mathcal{L}_2$-norm of the residual (normalised by the source) is less than 10^{-3}. In practice a smaller error tolerance might be chosen to stop the calculation, but

this allows us to clearly see trends in performance without the calculation taking excessively long. For our system, the iteration matrix $M \geq 0$ so $\rho(|M|) = \rho(M)$. The spectral radius is strictly less than one, so both the synchronous and asynchronous versions of Jacobi's algorithm are guaranteed to converge.

In common with many grid-based applications, when implemented using a distributed memory model a `halo swap' operation is required, since the update of a local grid point requires the data from each of the 6 neighbouring points in 3D. If a point lies on the boundary of a process' local grid, then data is required from a neighbouring process. To achieve this, each process stores a single-element `halo' surrounding its own local grid, and this is updated with new data from the neighbouring processes' boundary regions at each iteration (in the synchronous case), and vice versa, hence `swap'.

The overall structure of the program is shown in Figure 1, which is common between both versions of the algorithm.

```
do
   swap a one-element-thick halo with each neighbouring process
   every 100 steps
      calculate local residual
      sum global residual
      if global residual < 10^-3 then stop
      for all local points
         u_new(i,j,k)=1/6*(u(i+1,j,k)+u(i-1,j,k)
           +u(i,j+1,k)+u(i,j-1,k)+u(i,j,k+1)+u(i,j,k-1))
      for all local points
         u(i,j,k) = u_new(i,j,k)
end do
```

Figure 1 Pseudocode for parallel Jacobi

However, the implementation of the halo swap and the global residual calculation vary as follows:

In the synchronous version, halo swaps are performed using **MPI_Issend** and **MPI_Irecv** followed by a single **MPI_Waitall** for all the sends and receives. Once all halo swap communication has completed, a process may proceed. Global summation of the residual is done every 100 iterations via **MPI_Allreduce,** which is a blocking collective operation. In this implementation, all processes are synchronised by communication, and therefore proceed in lockstep.

The asynchronous implementation allows multiple halo swaps to be `in flight' at any one time (up to *R* between each pair of processes). This is done by means of a circular buffer storing **MPI_Request**s. When a process wishes to send halo data it uses up one of the *R* MPI requests and sends the halo data to a corresponding receive buffer on the neighbouring process. If all *R* MPI requests are active (i.e., messages have been sent but not yet received) it will simply skip the halo send for that iteration and carry on with the computation, until one or more of the outstanding sends has completed. We chose *R*=100 for our experiments. On the receiving side, a process will check for arrival of messages and, if new data has arrived, copy the newest data from the receive buffer into the halo cells of its `u` array (discarding any older data which may also have arrived). If no new data was received during that iteration, the calculation continues using whatever data was already in the halos. By using multiple

receive buffers (one for each message in-flight) we ensure that the data in the $\mathbf{u}$ array halos on each process is a consistent image of halo data that was sent at some iteration in the past by the neighbouring process.

In addition, since non-blocking collectives do not exist in the widely-implemented MPI 2.1 standard (although they exist in MPI 3.0) we also replace the blocking reduction with an asynchronous binary-tree based scheme, where each process calculates its local residual and inputs this value into the reduction tree. These local contributions are summed and sent on up the tree until reaching the root, at which point the global residual is broadcast (asynchronously) down the same reduction tree. Since the reduction takes place over a number of iterations (the minimum number being $2\log_2 P$), as soon as a process receives the global residual it immediately starts another reduction. In fact, even on 32768 cores, this asynchronous reduction takes only around 50 iterations to complete. Compared with the synchronous reduction (every 100 iterations), this gives the asynchronous implementations a slight advantage in potentially being able to terminate sooner after convergence is reached. This could of course also be achieved in the synchronous case, but at a higher communication cost.

One side-effect of the asynchronous reduction is that by the time processes receive a value for the global residual indicating that convergence is reached, they will have performed some number of further iterations. Since convergence in the asynchronous case is not necessarily monotonic, it is possible that the calculation may stop in an unconverged state. In addition, since the residual is calculated piecewise locally, with respect to current halo data, rather than the data instantaneously on a neighbouring process, the converged solution may have discontinuities along process' grid boundaries. To overcome this we propose that on reaching asychronous convergence a small number of synchronous iterations could then be performed to guarantee true global convergence, but we have not implemented this extension to the algorithm.

3.2 Block Jacobi

In [7] it was noted that, as the core count increased, the asynchronous communication involved in the point Jacobi algorithm became more beneficial. However, pointwise Jacobi is a very basic and slowly convergent algorithm and far better ones, such as those based upon Krylov subspace methods are available. The benefits gained from using asynchronous communication are far outweighed by the slowness of the convergence of the algorithm. It is possible to rewrite the Jacobi algorithm, not in terms of points but instead of blocks, where each block $\boldsymbol{x}_i$ is made up of a number of individual elements, and the matrix A is also split into blocks. The block Jacobi iterative algorithm can then be written as:

$$\boldsymbol{x}_i^k = A_{ii}^{-1}\left(\boldsymbol{b}_i - \sum_{j\neq i} A_{ij}\,\boldsymbol{x}_j^{k-1}\right) \quad i = 1, \dots, n_B$$

where n_B is the number of blocks. Note that computing the $\boldsymbol{x}_i^k$ requires the solution of a smaller linear system involving the diagonal block of A. This can be done using a conventional Krylov subspace solver (such as CG or GMRES), though convergence theory for this hybrid method is not well developed. The block size does not have to be chosen such than one block consists of the data on a single processor: we can choose much larger block sizes and use a parallel version of a Krylov subspace method to solve the inner linear systems. As with point Jacobi, the communication between blocks after each block Jacobi iteration can be done asynchronously. This interblock communication is implemented on a processor to processor basis rather than directing all communications via a master and has the same pattern as in the point Jacobi case. If new data is available after an asynchronous halo swap then this will be used in the next inner block solve. If no new data is available then we continue using existing data from a previous iteration. Global residual checking can be again be done using an asynchronous reduction where processes have an estimate of the global residual, which might be some number of iterations old.

Multisplitting can be thought of as a generalisation of block Jacobi, where instead of splitting the solution vector x into disjoint blocks, we allow the blocks to overlap. For a 3D stencil problem, the natural choice is to use domains that overlap by a fixed number of gridpoints in each dimension, and to communicate all the points in the overlap with the neighbouring processors (as a deep halo). For points in the overlaps, where more than one processor calculates the new elements of $\boldsymbol{x}$, a weighted average of the values is used to form the solution at the next step. The expectation is that multisplitting enables faster convergence, at the expense of additional computation and communication. For the 3D Laplace problem, it has been found that overlapping by between one and five elements in each dimension can improve convergence, but in the experiments in Section 5 we have used the simpler block Jacobi algorithm with no overlapping.

Another important issue in block Jacobi (or multisplitting) is deciding when the inner solves should be terminated, which is an active topic of our ongoing research. In the experiments reported here we use a fixed number of inner iterations for the entire run, where the value is chosen to minimise the overall execution time.

We have built our block multisplitting implementation using a GMRES solver from the PETSc suite as the inner solver. PETSc [13] is a suite of data structures and routines for the scalable, parallel solution of scientific applications modeled by partial differential equations. Using this existing suite we can easily select between different inner solution methods.

© CRESTA Consortium Partners 2011

4 Simulating performance faults

4.1 Slow cores

To simulate slow cores, we simply add some code to the computational part of the Jacobi algorithm (the two loops over local points in Figure 1) that does nothing useful, but consumes CPU time. We make the delays as fine-grained as possible, taking into account the resolution and overheads of calling the timer routine: for each iteration of the outermost of the triple nested loops over the local domain we measure the time taken t and then add a delay of $s \times t$, where s is the *slowdown factor*. We consider the three possible scenarios which we consider to be the most likely in practice: a single core running slowly, all the cores attached to one block of main memory (i.e. a NUMA domain) running slow, and all cores in a node running slow.

In the case of the block Jacobi implementation, we add the delay after each iteration of the GMRES solver: the PETSc library allows a user-defined routine to be called between iterations. As above, we measure the time taken t for each iteration, and then add a delay of $s \times t$, where s is the slowdown factor.

4.2 Slow links

To simulate a slow link in the synchronous version of the Jacobi algorithm, we replace the **MPI_Waitall** call with a loop over six calls to **MPI_Wait** (one for each neighbour). Immediately before each **MPI_Wait** call we add a delay of a fixed length which occurs with a fixed probability between 0 and 1.

For the asynchronous version, simulating delays is a little more difficult. Whenever a new message is received from a neighbouring process, with fixed probability between 0 and 1 we ignore it (by not copying the contents of the receive buffer in the `u` array halos), and also ignore any subsequent messages that arrive from the same neighbour within a fixed time interval. This has the same effect on the progress of the algorithm as if the link were blocked for a fixed period of time. Note that the cost of actually receiving the messages in the MPI library is not avoided, but this is unlikely to make a significant difference if the delay interval is sufficiently long.

We have not attempted here to simulate slow links in the block Jacobi code, since the MPI communication within the inner solver is spread across a number of different MPI routines, including collectives, called inside the PETSc library.

5 Results

5.1 Slow cores

In this section we report the results of our experiments that simulate slow running cores.

5.1.1 Jacobi algorithm

All our experiments are run on a Cray XE6 system. Each node of the system contains two 16-core AMD Interlagos processors, running at 2.3 GHz. Each 16-core processor consists of two 8-core NUMA domains, each with a shared L3 cache and 8GB of main memory. We solve the 3D Laplace problem on 1024 MPI processes using a local grid size of 50^3, which a global tolerance of 10^{-3}. We consider three cases:

- A single core running slowly (nslow=1)
- All eight cores in a NUMA domain running slowly (nslow =8)
- All 32 cores in a node running slowly (nslow=32)

In each case, we slow down the computational part of the algorithm by a factor which is varied from zero to 10. The slow core, or set of cores, is selected at random, and the results shown are an average of 10 runs.

Figure 2 shows the results of running these experiments using 32 nodes (1024 cores).

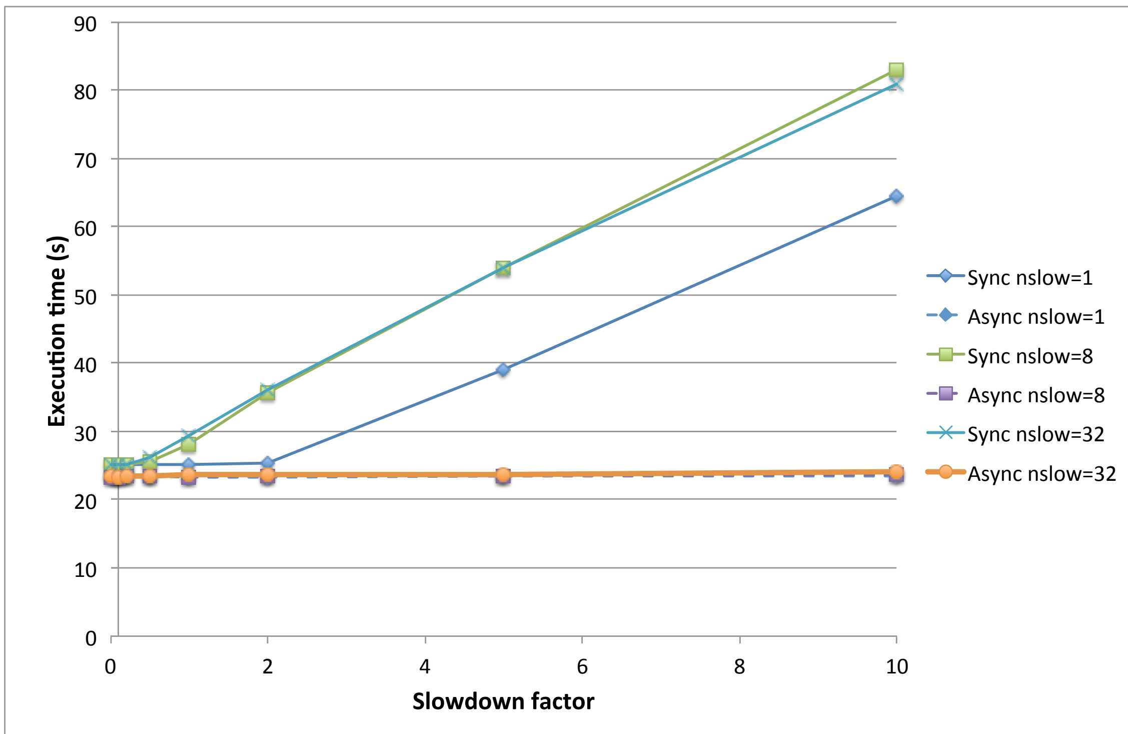

Figure 2 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6

We observe that the performance of the synchronous version of the algorithm is strongly affected by slow cores, whereas that of the asynchronous version is almost entirely unaffected. One surprising feature here is that the synchronous version with just one slow core is unaffected by a slowdown factor of up to 2. The reason for this is that the algorithm is very memory bandwidth dominated, with little re-use of data. With no slowdown, the computational phase of each timestep takes place at more-or-less the same time on all cores, followed by the communication phase, and a high degree of contention for memory bandwidth takes place. If one process suffers a modest delay, the processes with which it communicates are forced to start the next computation phase late, while others are free to begin theirs earlier. The computational phases in processes on the same NUMA domain no longer take place at more-or-less the same

© CRESTA Consortium Partners 2011

time: this reduces bandwidth contention, and all processes (including the delayed ones) execute their computational phases faster (by almost a factor of two), which compensates for the slow core. When more than one core is slowed down, the proportion processes delayed increases, and the effect is less marked.

Since this effect is somewhat peculiar to this code, and is unlikely to occur in real applications, we also ran the same experiment using 1024 MPI processes, but on 128 nodes, with just two MPI processes per NUMA domain (i.e. 8 MPI processes per node). This reduces the bandwidth contention significantly. In this case the number of slow cores simulated is one, two (on a NUMA domain) or eight (on a node). The results of this experiment are shown in Figure 3.

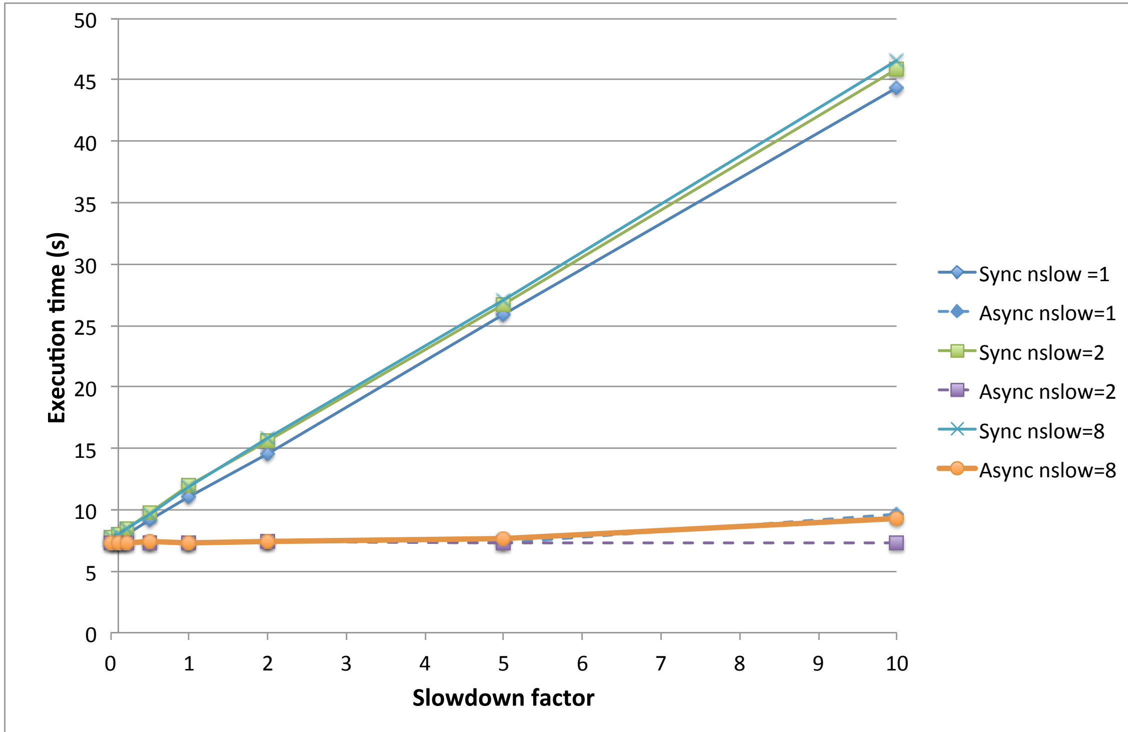

Figure 3 Execution time for Jacobi algorithm on 128 nodes of a Cray XE6

Now, as expected, the execution time of the synchronous version increases linearly with the slowdown factor, but again the execution time of the asynchronous version is barely affected by slow cores.

In the course of earlier work, we observed a real case of a slow running core performance fault on the Cray XE6. This fault triggered no hardware diagnostics, and could only be observed by its effect on application performance. Figure 4 shows the convergence behaviour of the synchronous and asynchronous versions of the code running on 32768 cores (1024 nodes). For the synchronous version, we also show the behaviour after the slow core was replaced. Unfortunately we did not re-run the asynchronous version under the same conditions with the replaced core, but other tests show that its convergence behaviour was unaffected. We can see that the slow core caused the synchronous version of the algorithm to converge at about half the rate, whereas the asynchronous version converges at a similar rate with the slow core as does the synchronous version after it was replaced.

© CRESTA Consortium Partners 2011

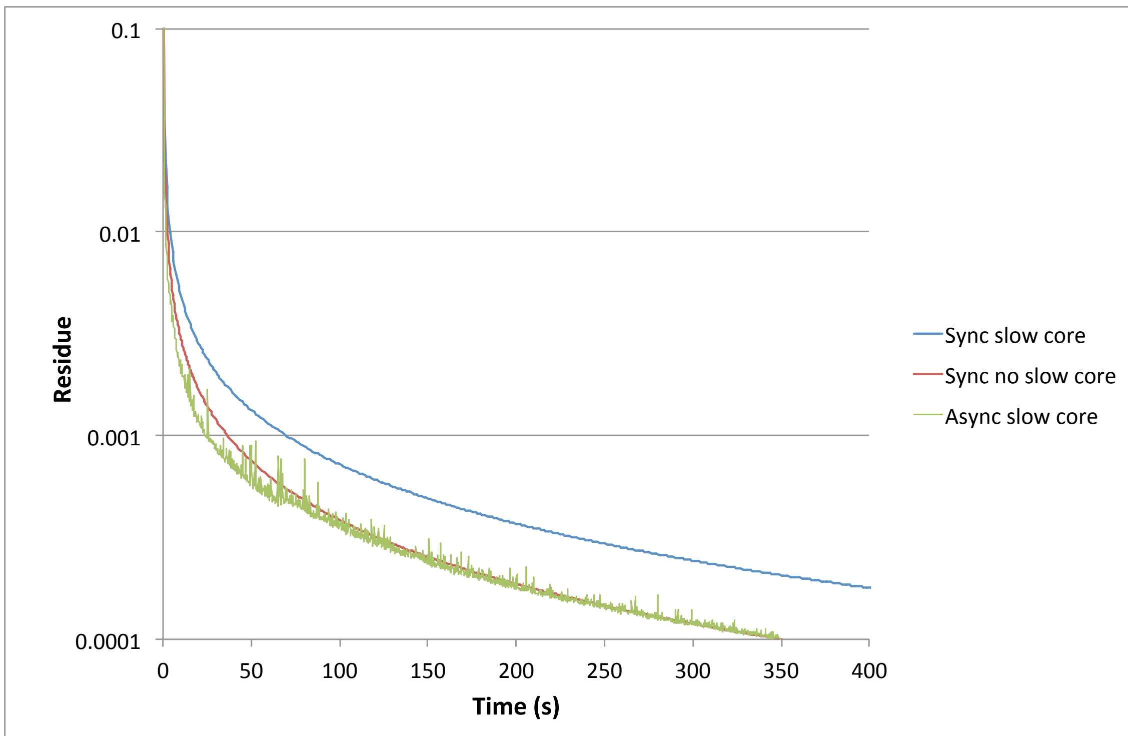

Figure 4 Residue vs. time for Jacobi algorithm on 1024 nodes (32768 cores) of a Cray XE6

5.1.2 Block Jacobi

In this section we report the effect of a slow core on the block Jacobi algorithm described in Section 3.2. Figure 5 shows the performance of the synchronous and asynchronous versions against the slowdown factor when running with on 1024 MPI processes (on 32 nodes) using a local grid size of 50^3, with a global tolerance of 10^{-4}. There are 32 blocks, four each in the x- and y-dimensions and two in the z-dimension. We fixed the number of inner iterations in the GMRES solver to 10.

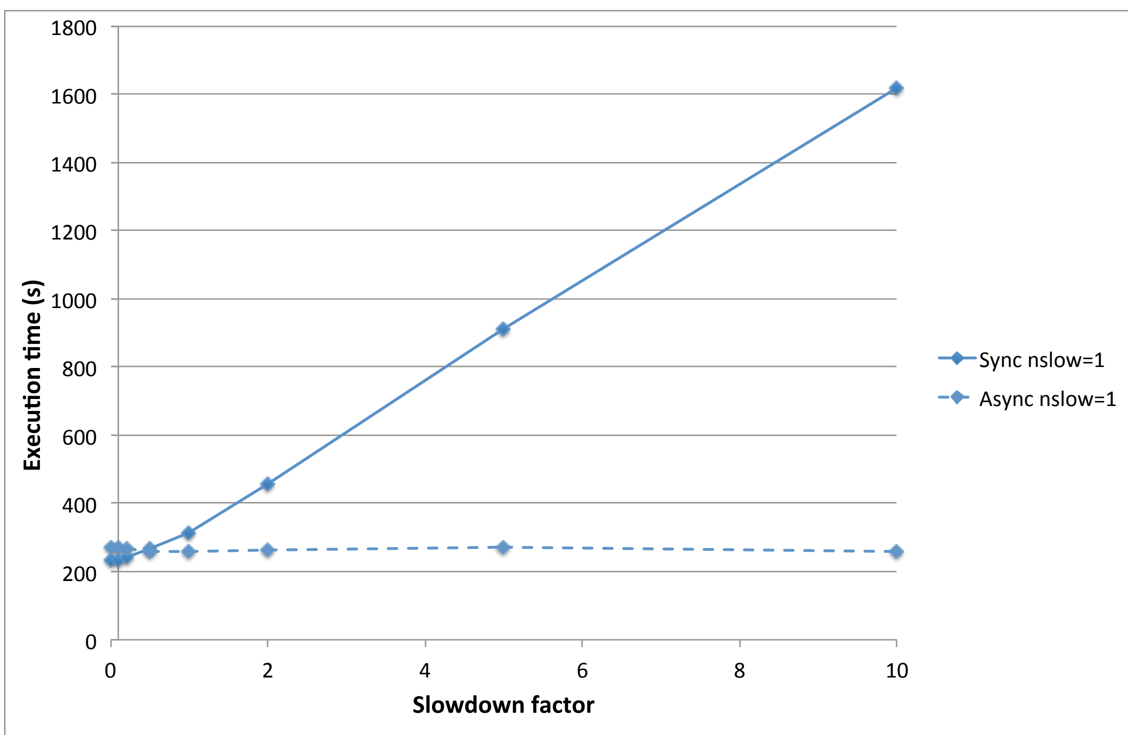

Figure 5 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 32 blocks

© CRESTA Consortium Partners 2011

We observe similar behaviour as with the simple Jacobi algorithm when bandwidth effects were eliminated: the execution time for the synchronous version grows almost linearly with the slowdown factor, but that for the asynchronous version is unaffected.

We have also experimented with using fewer blocks, in which case the synchronous algorithm converges in fewer outer iterations. However, for small block counts, the convergence behaviour of the asynchronous version became erratic, with a tendency for the convergence test to be passed too early. Further investigation showed that the reason for this is that our asynchronous reduction scheme for tracking the residual can underestimate the true residual. If we have solved the local systems within each block quite accurately, then the majority of the contribution to the global residual comes from points on the boundaries between the blocks. If the processors handling the boundaries have not received messages from their neighbours in other blocks since the start of the current outer iteration, the contribution to the residual from this in-flight data will be lost. This problem becomes especially severe when there are only a small number of blocks. We therefore modified our asynchronous residual tracking scheme to use the previous value of the residual on processors that have not recently a halo update since the start of their most recent inner solve.

The results of using this more conservative estimate are shown in Figure 6 to Figure 8 for 32, 8 and 2 blocks respectively.

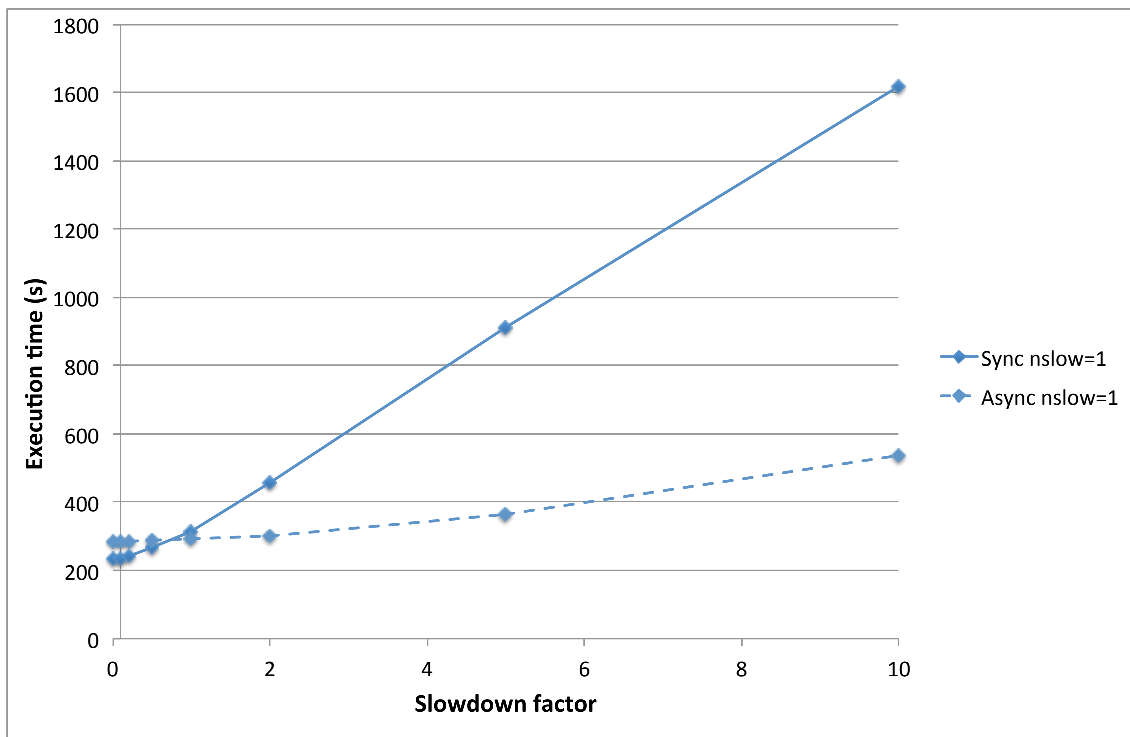

Figure 6 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 32 blocks and conservative residual estimation

We see that for 32 blocks, some of the ability of the asynchronous version to tolerate large slowdown factors has been lost: the execution time starts to grow for slowdown factors above 2.

With 8 blocks (Figure 7), early termination is avoided, and the asynchronous still shows better performance than the synchronous version at high slowdown factors. However, with only 2 blocks (Figure 8), the asynchronous version is always slower than the synchronous version.

© CRESTA Consortium Partners 2011

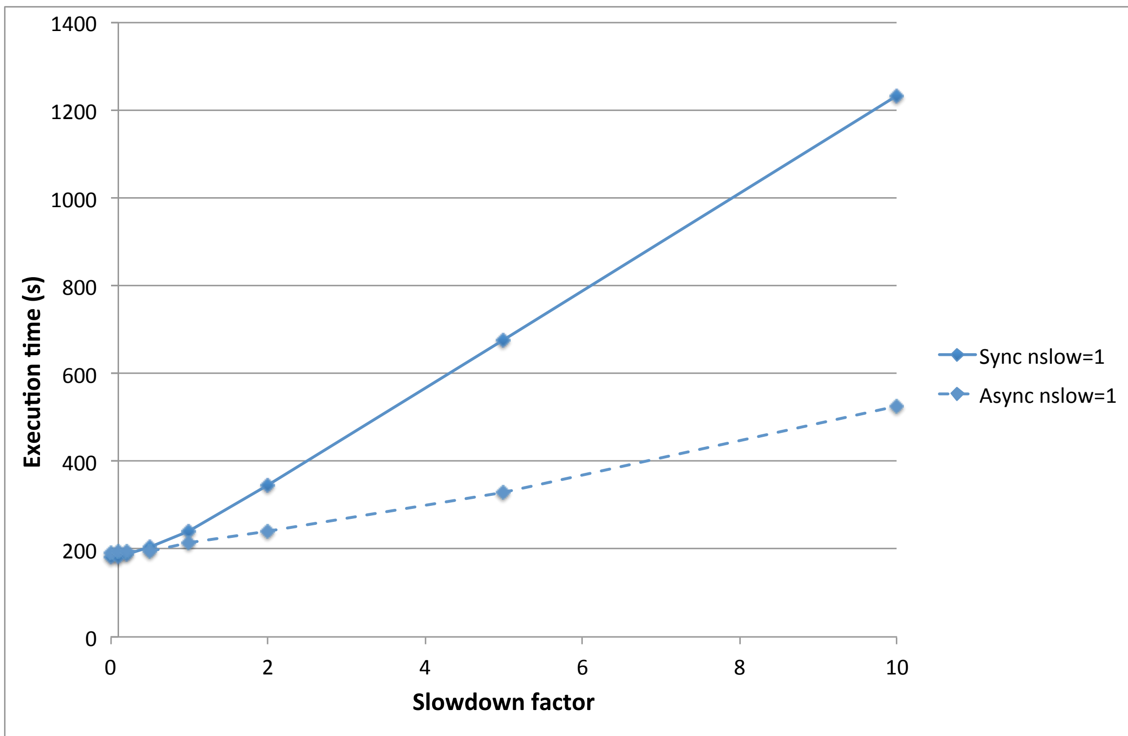

Figure 7 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 8 blocks and conservative residual estimation

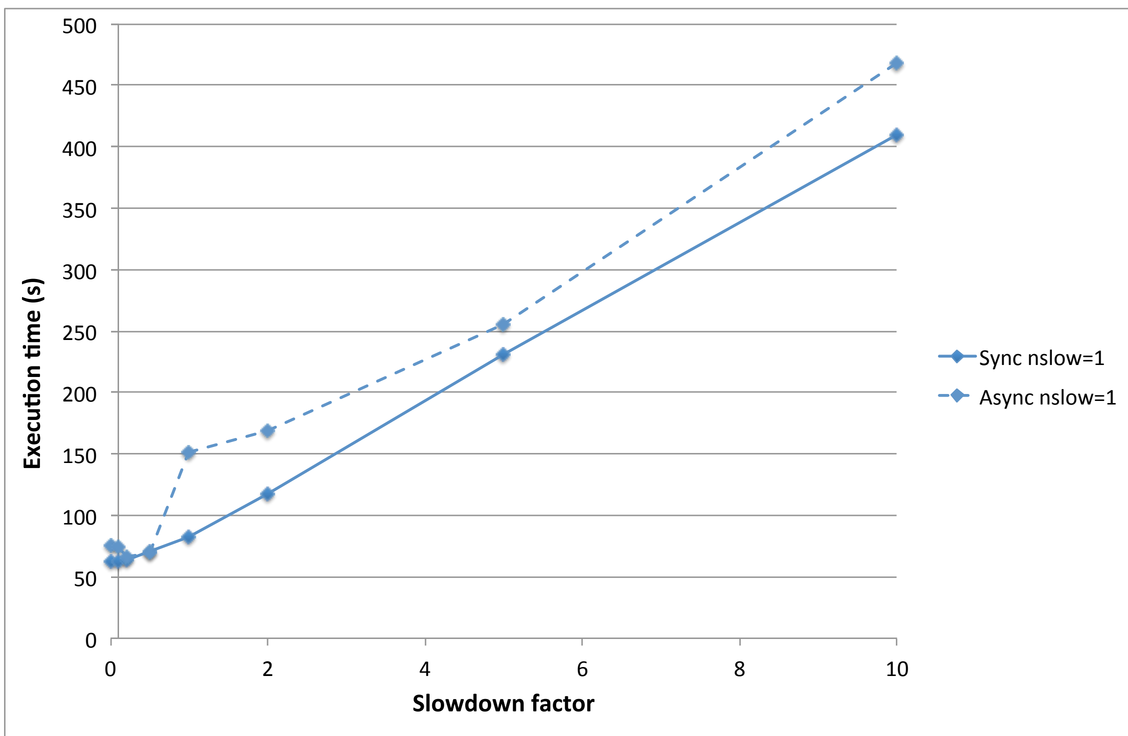

Figure 8 Execution time for block Jacobi algorithm on 32 nodes of a Cray XE6, using 2 blocks and conservative residual estimation

5.2 Slow links

In this section we report the effects of slow links on the performance of the synchronous and asynchronous versions of the Jacobi algorithm. Our first experiment shows the results of delaying a single, randomly chosen link. At each iteration of the algorithm, a delay of a fixed length is introduced with a given probability, as described in Section 4.2. Once again, we solve the 3D Laplace problem on 1024 MPI processes using a local grid size of 50^3, with a global tolerance of 10^{-3}, and the results are shown

in Figure 9. Note that the time taken for a single iteration in the synchronous version is approximately 3.6×10^{-3} seconds. Once the delay time becomes comparable with the length of an iteration, the performance of synchronous version of the algorithm is degraded, and the degradation increases as the probability of the delay occurring increases. In contrast, the asynchronous algorithm is unaffected by the simulated link delays.

In the final experiment, all links are subject to delays with the given probability, rather than just one. The synchronous version is again affected by the link delays once they are on order the length of an iteration, but the asynchronous version tolerates the delays almost perfectly. However, when we ran this experiment with the delay probability equal to 1 (results not shown), and with sufficiently long delays, the asynchronous version failed to give the correct results, because the convergence test succeeds prematurely. Additional tests would need to be put in place to safeguard against this occurring.

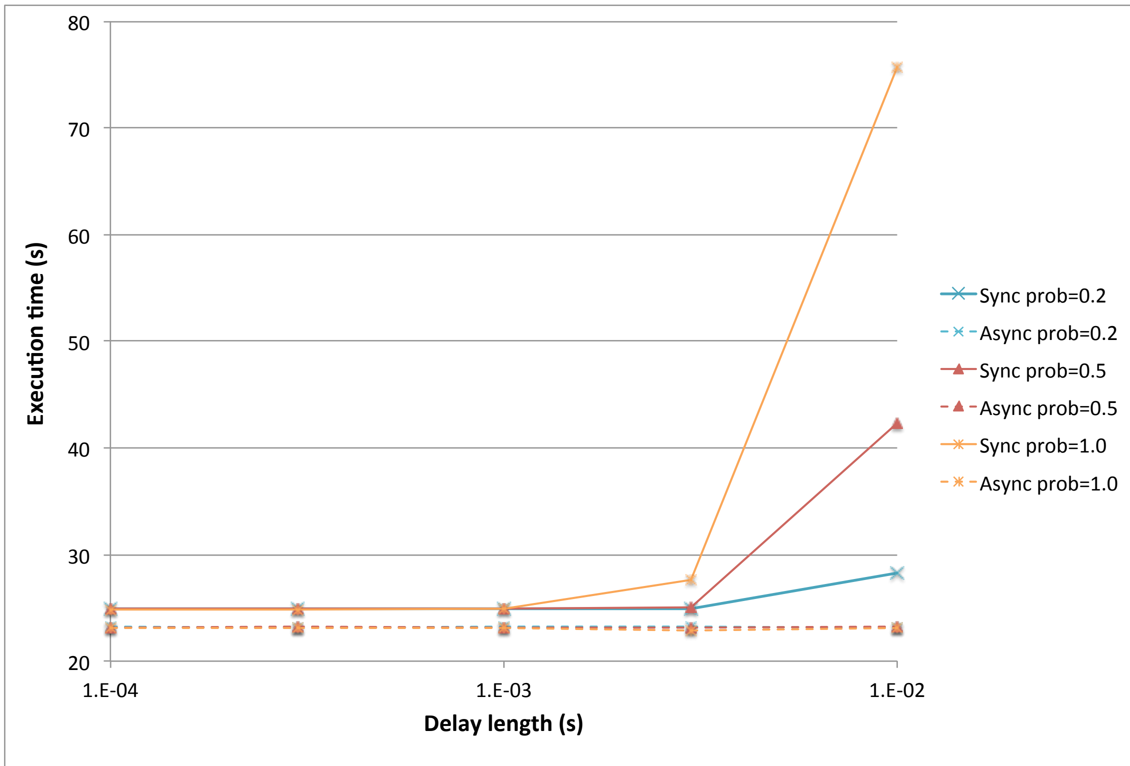

Figure 9 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6 with one slow link

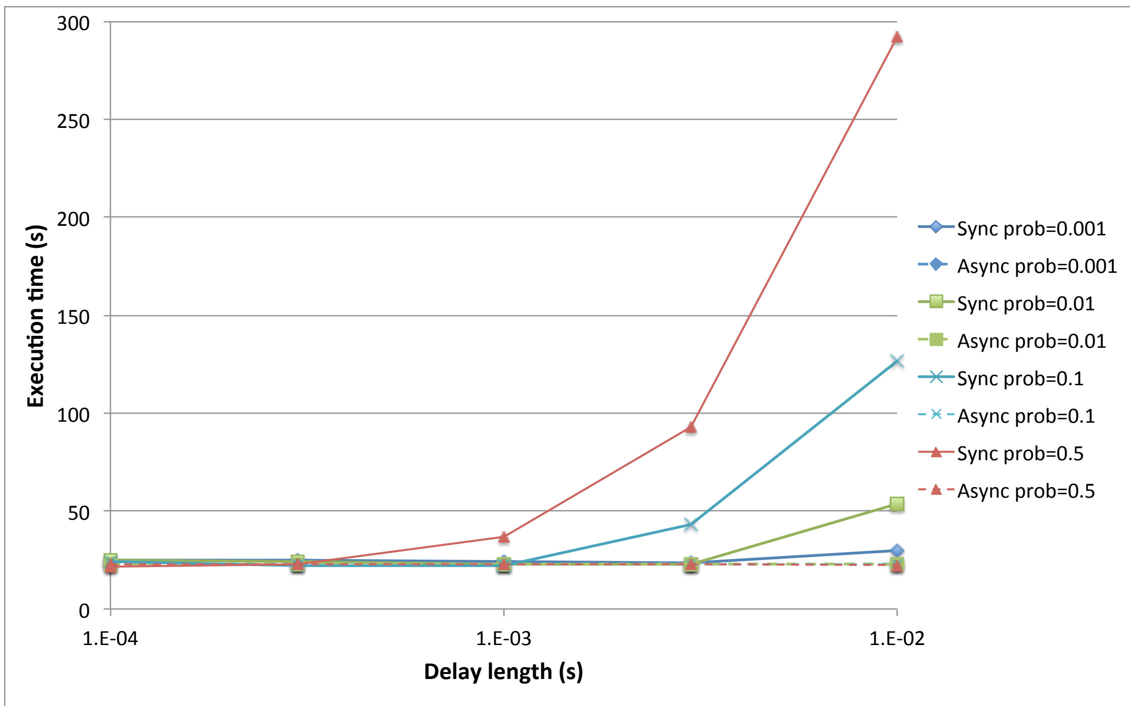

Figure 10 Execution time for Jacobi algorithm on 32 nodes of a Cray XE6 with all slow links

© CRESTA Consortium Partners 2011

6 Conclusions and Future Work

We have undertaken some experiments to assess the ability of two algorithms for solving large sparse linear systems (asynchronous Jacobi and asynchronous block Jacobi) to tolerate performance faults. We have simulated the effects of slow running cores and slow running links, and the results suggest that these algorithms are indeed highly insensitive to such faults compared to their synchronous counterparts, even when the degree of component performance loss is quite large. However, in the block Jacobi case, the performance faults can induce erratic convergence behaviour in the asynchronous algorithm. This can be avoided by using a more conservative way of estimating the global residual, but this comes at the expense of a reduced ability of the asynchronous version to tolerate slow cores.

Future work in this area includes applying these algorithms to more interesting real-world problems, and undertaking further studies to improve convergence detection. It would also be interesting to apply the same performance fault simulation methodology to other asynchronous algorithms, such as asynchronous Schwartz methods for solving PDEs.

Although our asynchronous solvers are not yet mature nor general enough to be used in a full application, there are other areas within the CRESTA project where asynchronous algorithms, and their abilities to tolerate performance faults and heterogeneity are of interest. Task 2.4 is investigating the alternative uses of fat nodes, which includes the possibility of co-locating applications that do not compete for the same hardware resources on the same nodes. One of the difficulties here is that any resource contention which does occur may do so in ways which are not well synchronised between the nodes, and asynchronous methods could tolerate this better than synchronous ones. An important theme in the rest of Task 2.5 is power management. Attempts to reduce power consumption by reducing the clock rate of CPUs when they are idle, for example, is also difficult to synchronise across nodes, and may result in load imbalance in highly synchronous codes. Once again the ability of asynchronous methods to tolerate differences in CPU clock rates can be of interest.

© CRESTA Consortium Partners 2011

7 References

[1] G.M. Baudet, *Asynchronous iterative methods for multiprocessors*, Journal of the Association for Computing Machinery, vol 25, no. 2, pp 226-244, 1978.

[2] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall,1989

[3] D. Chazan and W. Miranker, *Chaotic Relaxation*, Linear Algebra and Its Applications, vol. 2, pp. 199-222, 1960.

[4] J.M. Bull and T.L. Freeman, *Numerical Performance of an Asynchronous Jacobi Iteration*, Proceedings of the Second Joint International Conference on Vector and Parallel Processing (CONPAR}'92), pp. 361-366, 1992.

[5] D.V. de Jager and J.T. Bradley, *Extracting state-based performance metrics using asynchronous iterative techniques*, Performance Evaluation, vol. 67, no. 12, pp. 1353-1372, 2010.

[6] J. M. Bahi, M. Jacques, S. Contassot-Vivier, and R. Couturier, Performance Comparison of Parallel Programming Environments for Implementing AIAC Algorithms, J. Supercomput, vol. 35, no. 3, pp. 227-244, 2006.

[7] I. Bethune, J.M. Bull, N. Dingle and N. Higham, *Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP*, International Journal of High Performance Computing Applications, to appear.

[8] N. Brown, J.M. Bull and I.Bethune, *A hybrid approach for extreme scalability when solving linear systems*, to appear in Proc. of EASC2013: Solving Software Challenges for Exascale, Edinburgh, April 2013.

[9] F. Magoulés and C. Venet, *Asynchronous Parallel Algorithms for Petaflop and Exaflop Computation*, Chapman Hall, to appear.

[10] F. Jézéquel, R. Couturier, and C. Denis, *Solving large sparse linear systems in a grid environment: the GREMLINS code versus the PETSc library*, in Journal of Supercomputing vol. 59, no.3, pp. 1517-1532, 2012

[11] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.* vol. 47, no. 8, pp. 225-234, 2012.

[12] J. Bahi, S. Contassot-Vivier and R. Couturier, *Coupling Dynamic Load Balancing with Asynchronism in Iterative Algorithms on the Computational Grid*, in 17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium, IPDPS 2003, Nice, France, pp. 40-49, IEEE computer Society Press, 2003.

[13] D. O'Leary and R. White, *Multi-splittings of matrices and parallel solution of linear systems*", SIAM J. Alg. Disc. Meth., vol. 6, pp. 630-640, 1985.

[14] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knep- ley, L. Curfman, B. Smith and H. Zhang. *PETSc Users Manual*, ANL-95/11 Revision 2.3.2, Argonne National Laboratory, 2006.

© CRESTA Consortium Partners 2011