Performance Analysis of Asynchronous Parallel Jacobi

Hook, James and Dingle, Nick

2013

MIMS EPrint: **2013.52**

ISSN 1749-9097

Performance Analysis of Asynchronous Parallel Jacobi

James Hook [*]& Nick Dingle[†]

October 29, 2013

Abstract

The directed acyclic graph (DAG) associated with a parallel algorithm captures the order in which separate local computations are completed and how their outputs are subsequently used in further computations. Unlike in a synchronous parallel algorithm the DAG associated with an asynchronous parallel algorithm is not predetermined. Instead it is a product of the asynchronous timing dynamics of the machine and, as such, it is best thought of as a pseudorandom variable. In this paper we present a new tighter bound on the rate of convergence of asynchronous parallel Jacobi (APJ), which is based on statistical properties of the DAG and is valid for systems which satisfy a standard sufficient condition for convergence.

We also describe an experiment in which we make a precise log of the calculations taking place during an implementation of APJ on a distributed memory multicore machine, which enables us to reconstruct and study the DAG. We demonstrate that our bound provides a good approximation of the true rate of convergence in these examples and show how problems in the algorithm's implementation can affect the asynchronous timing dynamics and in turn the rate of convergence of the algorithm.

Introduction

Jacobi's method for solving a linear system of equations iterates an affine map $x \mapsto Mx + b$. In a distributed memory parallel environment x, M and

[*]This author was supported by Engineering and Physical Sciences Research Council (EPSRC) grant EP/I005293 "Nonlinear Eigenvalue Problems: Theory and Numerics".

[†]This author was supported by EPSRC grant EP/I006702/1 "Novel Asynchronous Algorithms and Software for Large Sparse Systems".

b can be broken up into N blocks of rows, so that processor i stores x_i, b_i and $M_{i:}$. Processor i is then responsible for updating its own block of rows and iterates the map by

$$x_i \mapsto M_{i,i}x_i + b_i + \sum_{j \neq i} M_{i,j}x_j,$$

where the x_j are the input from other processors. In order to update its state processor i must have appropriate input from its neighbouring processors.

In the synchronous implementation, a processor cannot finish computing its update until it receives updates from its neighbours previous computations:

```
for k=1,2,...
  x(k)=M*x(k-1)+b
  wait to receive all y(n-1) values
  x(k)=x(k)+y(k-1)
  if converged stop
  broadcast new state
end
```

Delays in interprocessor communication give rise to delays and idle time in the whole computation.

Introduced by Chazan and Miranker [1], chaotic or asynchronous iterations are parallel asynchronous processes in which idle time is removed by allowing individual processors to begin a new iteration as soon as their previous one is completed using whatever input data they have stored in their local memory. This means that processors may use iterate components that are out of date and that some processors will be able to update many more times than others.

In the asynchronous implementation, processors successively update without waiting for communication from their neighbours:

```
for k=1,2,...
  x(k)=M*x(k-1)+b
  x(k)=x(k)+y
  if converged stop
  broadcast new state
end
```

The idea is that by minimizing idle time the update rate of the individual processors is increased and that this can improve the rate of convergence.

However the possible use of older input data can slow convergence and even lead to instability and divergence.

In [2, 3] the authors study the performance of asynchronous parallel Jacobi (APJ) in terms of time to converge. In this paper we will make a similar study except that we record a complete log of the relevant calculations that take place in the implementation. We record all data of the form "at time t processor i updates for the nth time using the mth value of processor j as an input".

The composite of all this data is a *directed acyclic graph* (DAG), whose vertices are the successively computed values associated with each processor and whose edges represent use in computation. The DAG contains an edge from processor j's mth update to processor i's nth update if processor i updates for the nth time using the mth value computed by processor j for its input. The DAG therefore gives us a complete account of the asynchronous dynamics of the algorithm's execution and enables us to explain and make accurate predictions about performance.

Although we will only study an implementation of APJ, there are many other asynchronous algorithms that could benefit from this sort of detailed study.

Avron and Gupta [4] propose a shared-memory asynchronous method for general symmetric positive definite matrices based on the randomised variant of the Gauss-Seidel iteration. Here the rate of convergence is bounded below using the condition number of the matrix and the quantity τ, which is defined as the maximum difference in 'iteration number' between the input and output of a single computation. The authors suggest this should be of the same order as the number of processors. We will see in our experiments that this can be quite a lot larger. In the experiments described in this paper on a 24 processor machine we found $\tau = 10203$ in the first example and $\tau = 308$ in the second.

In fact any variant of the Gauss-Seidel iteration will admit the formulation we develop in section 2. The classical Gauss-Seidel iteration or over relaxation methods give rise to an iteration with a predetermined DAG, whilst randomised Gauss-Seidel iteration gives rise to a random DAG whose randomness derives from a pseudorandom number generator rather than the chaotic dynamics of the execution itself (as is the case in APJ).

Richtarick and Takac [5] propose a family of distributed parallel asynchronous algorithms for certain nonlinear convex optimization problems. These methods can be thought of as hybrids between parallel stochastic gradient descent and randomized parallel block coordinate descent. As in [4], the performance analysis is based on the quantity τ.

Lu and Tang [6] propose an algorithm for solving systems with symmetric positive definite matrices over a network of asynchronous agents, whose interactions are controlled by some (random) external network dynamics. For example the agents could be thought of as devices in a wireless network which are physically moving around, so that their network of connectivity is constantly changing. Here the rate of convergence is bounded below using spectral properties of the problem matrix and a connectedness measure of the dynamic interaction network.

This paper is organized as follows. In section 1 we review APJ. In section 2 we explain our data logging technique and examine the basic statistics of two different implementations of APJ. In section 3 we prove a sandwich of bounds on the rate of convergence. In section 4 we apply these bounds to our examples and discuss the weakest link scenario, in which the rate of convergence is determined completely by the update rate of the slowest processor.

1 Asynchronous Parallel Jacobi

We choose Jacobi's method as the algorithm on which to base our investigations because there exist sufficient conditions for it to converge both synchronously and asynchronously.[1] Jacobi's method for the system of linear equations $Ax = b$, where the $n \times n$ matrix A is assumed to be nonsingular and to have nonzero diagonal, computes the sequence of vectors $x(k)$, where

$$x_i(k) = \frac{1}{a_{ii}}\Big(b_i - \sum_{j \neq i} a_{ij} x_j(k-1)\Big), \qquad i = 1\colon n.$$

The $x_i(k)$, $i = 1\colon n$, 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 $\rho(|M|) < 1$ [1], where $\rho(|M|)$ denotes the spectral radius (maximum absolute value of the eigenvalues) of the matrix $|M| = (|m_{ij}|)$ and $M = -D^{-1}(L + U)$, which we call the *iteration matrix*. Here, D, L, U are, respectively, the diagonal and strictly lower and upper triangular parts

[1]In practice, synchronous parallel versions of other iterative schemes (e.g. conjugate gradient methods) are often preferred, but asynchronous parallel versions of these approaches are not yet well developed.

of A. The synchronous version of Jacobi's method converges if the weaker condition $\rho(M) < 1$ holds.

2 DAG logging

We instrument our implementation as follows. When a processor finishes computing a vector update it records the current time and local iteration number in an array in memory. The local iteration number is then transmitted with the vector update when the update is communicated to the processor's neighbours. When this vector is used to update a remote processor's local vector, the iteration number associated with the vector, the current time and the local iteration number are also recorded in an array. As all the logging information is stored in memory and not written to disk until the iterations have converged, the effect on the run-time of the overall calculation is minimized.

Example 2.1 (Processor 23 problem) We solve $Ax = b$ for $A \in R^{n\times n}$ with $n = 20\,000\,000$. A is tridiagonal with 2.001 on the diagonal and -1 on the off diagonals, and $b_i = i/n \;\; i = 1, \dots, n$. The iteration matrix M is therefore non-negative with zeros everywhere except on the sub and super-diagonal where the entries are all 1/2.001. There exist explicit formulae for the eigenvalues of matrices of this form [7]. We have implemented a parallel Jacobi solver using C++ and MPI, and executed it on the Computational Shared Facility (CSF) at the University of Manchester. Each CSF node contains two 6-core Xeon 2.50GHz CPUs and 48GB of memory, and multiple nodes are connected via Infiniband. The MPI library is OpenMPI version 1.4.3.

Table 1 summarizes the number of iterations and total solution time observed when executing our solver on 24 cores (2 nodes) of the CSF for synchronous and asynchronous Jacobi. We observe that the solution time is much higher for the asynchronous execution than the synchronous, and that the average number of iterations in the synchronous case is approximately the same as the lowest number of iterations in the asynchronous case.

To quantify the performance of an execution we define the *effective update rate* to be the update rate that would be required of a synchronous execution to achieve the same convergence time

$$\text{effective update rate} = \frac{\text{numer of iterations required for synchronous execution to converge}}{\text{time for execution to converge in seconds}}.$$

	Iterations			Time	Update Rate (s^{-1})	
	Min.	Average	Max.	(s)	Mass	Effective
Synchronous	–	7 866	–	569.8	13.8	13.8
Asynchronous	7 748	14 239	17 951	1038.72	13.7	7.6

Table 1: Processor 23 problem results

Processor ID	Update number	Time (ms)
23	3	200
21	6	358
23	4	361
24	8	417
22	4	417
23	5	524

Table 2: Subset of update array for example 2.1

Since this is only equal to the actual update rate in the synchronous case we also define the *mass update rate* to be the total update rate averaged over the number of processors in either the synchronous of asynchronous case

$$\text{mass update rate} = \frac{\text{total number of processor iterations performed during execution}}{24 \times \text{time for execution to converge in seconds}}.$$

Therefore the mass update rate measures roughly how quickly an implementation is performing updates and the effective updates measures how useful these updates actually are.

Surprisingly, in this example the mass update rate of the synchronous execution is greater than that of the asynchronous iteration, which is not what we would have expected as the asynchronous execution ought to be faster. Since the time to converge is much greater in the asynchronous case the effective update rate is much smaller.

A small sample of calculation data that we recorded from the asynchronous execution is displayed in Table 2 and Table 3. Table 2 contains update data which tells us that, for instance, processor 23 updated for the 5th time at $t = 524ms$. Table 3 contains calculation the corresponding data that tells us that when processor 23 made this update it used processor 22's 7th value and processor 24's 8th value.

We bring together all of this data to produce the DAG of the execution. Figure 1 shows a plot of the reconstructed DAG. Vertices represent

Processor ID	Update number	Neighbour ID	Input update number
21	7	22	4
21	7	20	6
24	8	23	3
23	5	24	8
23	5	22	7

Table 3: Subset of calculation array for example 2.1

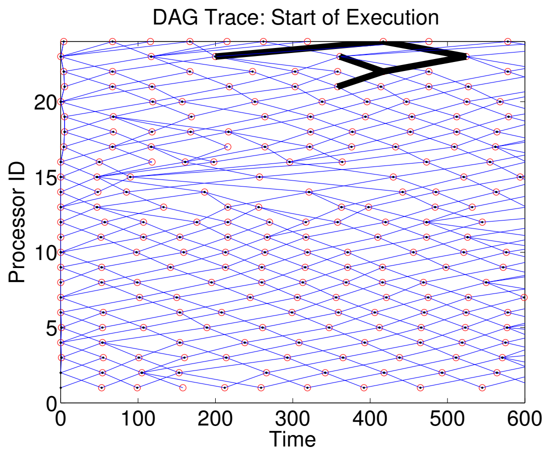
successive updates and edges represent use in computation. Note that every processor's update uses its own previous state in the computation so strictly speaking there should be an edge between each processor's successive updates. However since this is always the case and since it would make the figure hard to read we omit these edges.

We can now analyse the properties of the DAG. We define the *inter-update time* to be the duration of time between successive updates of a processor. We also define the *input freshness* of a piece of input data created by processor j and used by processor i to be the duration of time between processor j's update that produces the data and processor i's update that uses the data. In terms of the DAG plots the input freshness is the horizontal span of an edge in the graph. Figure 2 shows the average inter-update times of each processor as well as the average input freshnesses for communication between neighbouring processors.

Processor 23 is a clear outlier in terms of interupdate times. Its mean interupdate time is roughly double that the other processors.

Since the iteration matrix has a tridiagonal structure there is only communication between processors with ID differing by one. We say that communication from processor $i+1$ to i is downwards and that communication from processor $i-1$ to i is upwards. Notice that the upwards communication is consistently slower than the downwards communication. This is because of the order in which processors broadcast their newly updated values and process inputs from their neighbours. If we reverse this communication ordering we also reverse the relationship between the upwards and downwards input freshnesses. Reversing the communication ordering also causes processor 2 to be the slowest (rather than 23), which leads us to theorise that the poor performance of processor 23 has something to do with the internal workings of the MPI library.

Notice also that communication from processor 23 to processor 24 is considerably slower than all of the other communication channels. This can



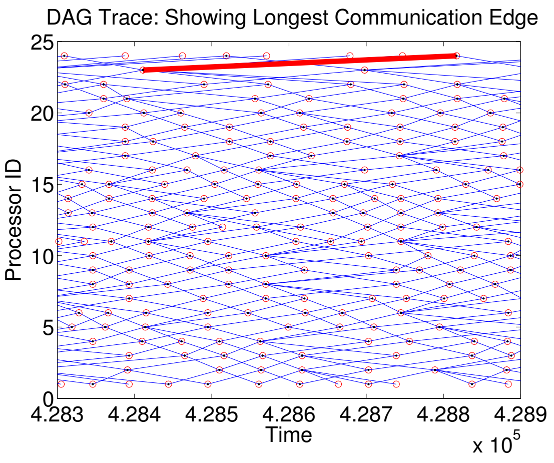


Figure 1: Example 2.1. Reconstructed DAG. Red circles represent updates, blue lines represent use in computation. Bold black lines highlight computations also displayed in the above tables. The bold red line indicates the most out of date input used in the execution.

be attributed to the slower update rate of processor 23. Even when 24 uses the most recent value from 23 it will often be quite old in terms of time delay.

We can break down these results further by studying the sampled distributions of interupdate times and input freshnesses. Figure 3 shows the sampled distributions of interupdate times for each processor, where 23 can clearly be seen to be an outlier. Notice that the processors other than 23 have a mode at around 50ms and that processor 23 appears to have modes at multiples of 50ms.

Likewise we plot the input freshness distributions in Figure 4. As in Figure 2 we see that upwards communication is generally slower than downwards communication. When we average over the non-outlying communication delay distributions we see that the while both distributions have modes at around 50ms and 100ms the upwards distribution has a heavier tail, which accounts for its greater average. Communication from processor 23 to its neighbours provides two outliers. Communication from 23 to 24 is on average the slowest and has a very sporadic distribution. While communication from 23 to 22 is not as slow on average it does have a markedly different distribution to the other communication channels, with a strong mode at around 100ms.

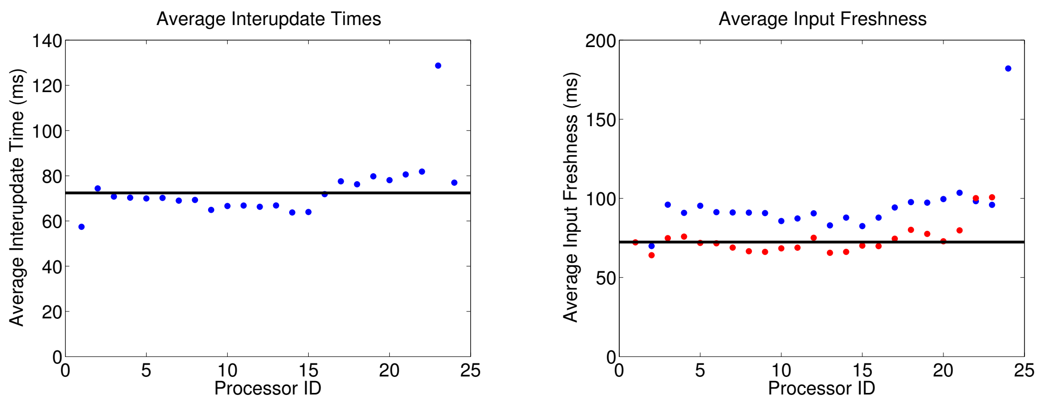


Figure 2: Example 2.1. Left, average processor interupdate times. Right, Average communication delay between processors, upwards in blue, downwards in red. Black lines show average values for synchronous execution.

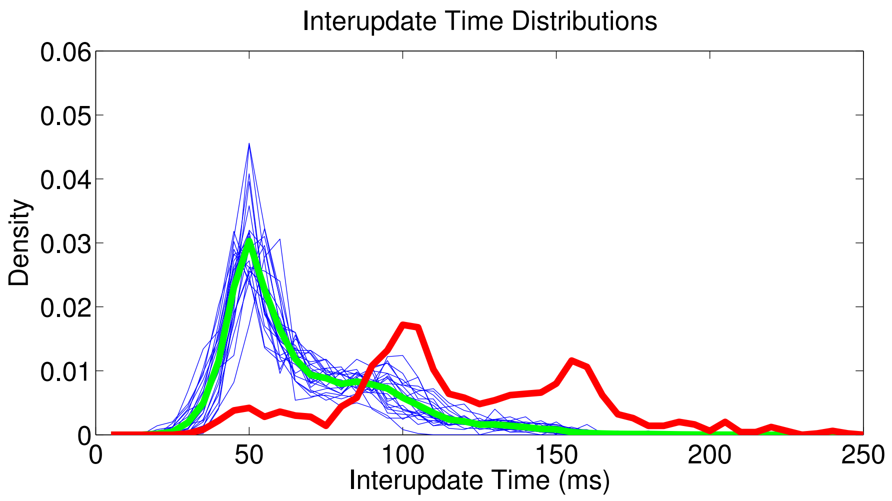


Figure 3: Example 2.1. Sampled distribution of interupdate times for each of the 24 processors. Processor 23 in red. Group average of all other processors in green.

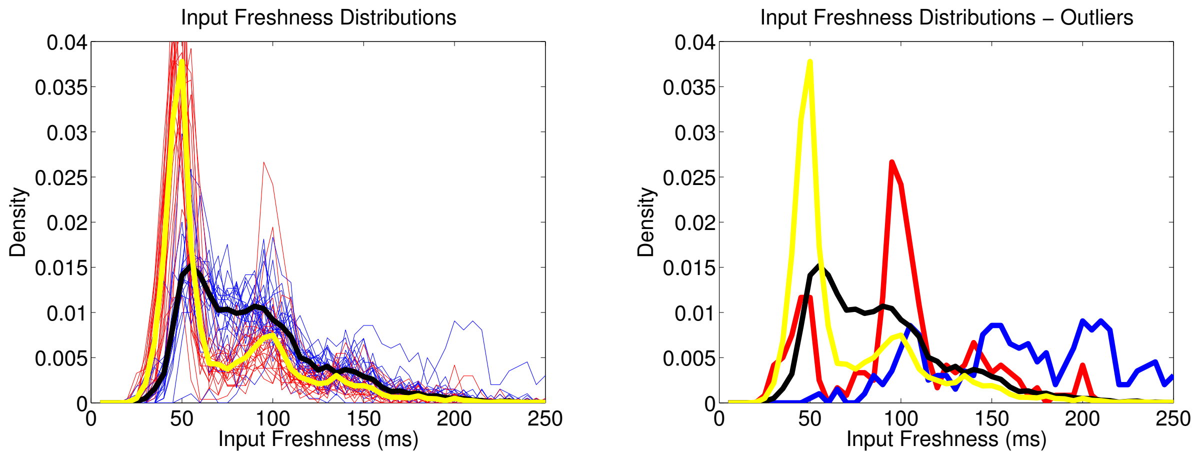


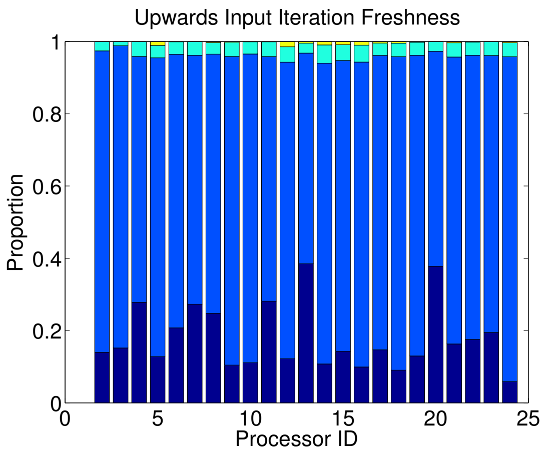
Figure 4: Example 2.1. Sampled distributions of input freshness. Upwards in blue, downwards in red. Bold yellow is overall distribution of downwards communication, bold black upwards. Bold red is communication from 23 to 22, bold blue is communication from 23 to 24.

The longest single communication delay is 466ms which is attained by processor 23 broadcasting its 3340th value to processor 24, which is used in its 5558th computation.

We can also measure input freshness in the discrete timeframe induced by the updates sequence. If processor j updates at time t_1 and processor i uses this value to update at time t_2 then the *iteration freshness* of this communication is equal to the number of times that j updates in the interval (t_1, t_2). Figure 5 is a stacked chart of the iteration freshness of upwards and downwards communication. We found iterations delays of up to 6 but they where so infrequent that they are not visible in the chart. Upwards communication mostly has iteration freshness of 1, while downwards communication has mostly iteration freshness of 0.

A possible explanation of the pathological behaviour observed in this execution is that the communication causes a weak synchronisation between neighbouring nodes. This would explain why the input freshness distributions all have modes at multiples of 50ms, which is the mode of the interupdate times. If the updates are roughly in time with each other then whatever the input freshness is in terms of iteration delay it will always be a multiple of 50ms in time delay.

This could also explain the longer interupdate times of processor 23.



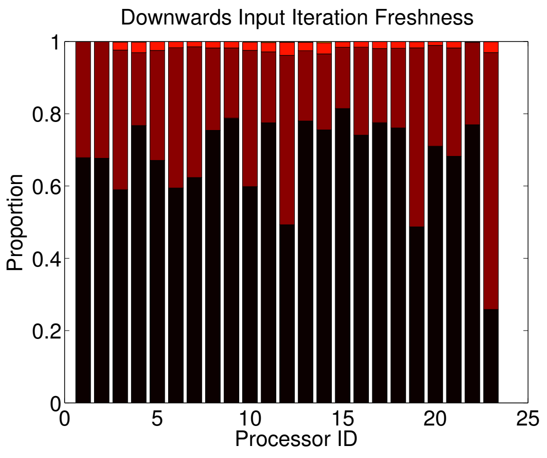


Figure 5: Example 2.1. Stack column charts showing iteration freshness. Darkest regions represent zero delay with delay increasing by one with each new region. Upwards communication in blue on left, downwards communication in red on right.

Perhaps it must wait for some other processor to communicate with it before it can update itself (as it is being given some extra work to oversee all of the communication), and this is why it is slower and why its interupdate times tend to be multiples of 50ms.

Example 2.2 (Processor 23 problem resolved) It was not immediately obvious what was causing processor 23's poor performance. We discounted a hardware fault as the same behaviour was consistently observed across multiple executions on different CSF nodes. A's regular structure meant that computational load was balanced across the MPI ranks, as was the amount of data communicated at each iteration (save for the edge ranks, 1 and 24, which only communicated with one neighbour rather than two). The asynchronous solver was implemented using MPI-2's single-sided communication functionality, and we theorised that this behaviour was a manifestation of the inner-workings of the MPI-2 implementation on our machine.

To test this we used a variant of our solver which employed MPI-1's asynchronous communication methods and explicit buffer management to achieve asynchronous execution. We also implemented a synchronous version using standard MPI-1 functions as the synchronous solver in the previous section also used MPI-2 single-sided communication.

	Iterations			Time	Update Rate (s^{-1})	
	Min.	Average	Max.	(s)	Mass	Effective
Synchronous	–	7 865	–	495.5	15.8	15.8
Asynchronous	8 120	8 317	8 428	490.5	17.0	16.0

Table 4: Processor 23 fixed results

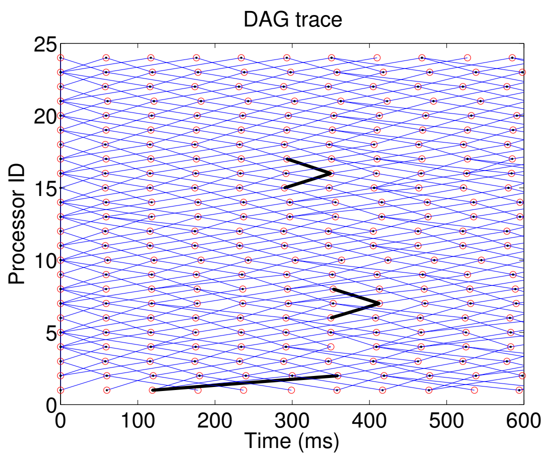
Table 4 summarises the number of iterations and total solution time observed when executing our new solvers on 24 cores (2 nodes) of the CSF for synchronous and asynchronous Jacobi. Compared with Table 1, it can be seen that removing MPI-2's single-sided communication improved the performance of both the synchronous and asynchronous solvers, and that the asynchronous solver now outperforms the synchronous. Furthermore, the distribution of iteration counts is much less widely spread than for the MPI-2 solver, and we no longer find that rank 23 performs significantly fewer iterations than the other ranks.

We reconstruct the DAG from the calculation data as in example 2.1. Figure 6 shows a sample of the reconstructed DAG. With the exception of the communication from processor 1 to processor 2 the structure of this graph is far more regular than before. Although there are some random fluctuations the interupdate times are all roughly the same and communication tends to have iteration delay equal to 1 on all channels. Communication from processor 1 to processor 2 tends to be slower, although processors 1 and 2 complete their updates no slower than the other processors.

Figure 7 shows the average interupdate times and average input freshness for this execution. Figure 8 shows the sampled distributions of interupdate times and input freshness. The interupdate times have a bell shaped distribution and there is little difference between the different processors. The 'spikiness' of the distribution is due to the coarse time measurement. The input freshness distributions look like a (stochastic) linear combination of uniform distributions with end points at multiples of 60ms (the mode of the interupdate times).

The longest single delay is 382ms which is attained by processor 1 broadcasting its 1130th value to processor 2, which is used in its 1129th computation.

Figure 9 shows the input iteration freshness. The vast majority of communication has freshness 1. Communication from processor 1 to processor 2 is often quite a lot slower. We found iteration delays of up to 7 but they where so infrequent that they are not visible in the stacked chart.



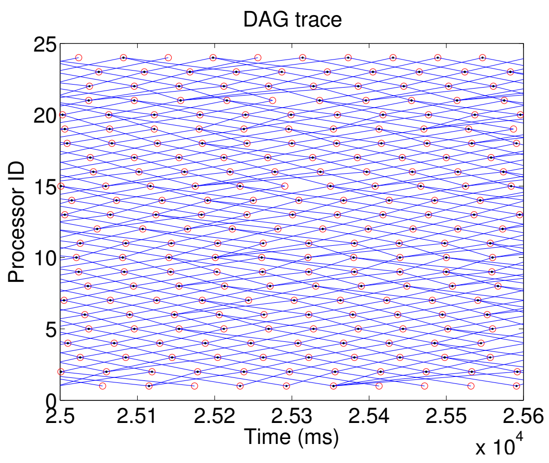
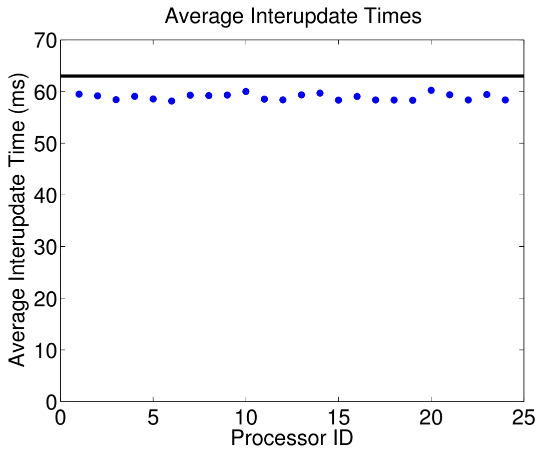


Figure 6: Example 2.2. Reconstructed DAG. Red circles represent updates, blue lines represent use in computation. Bold black lines highlight regular structure.



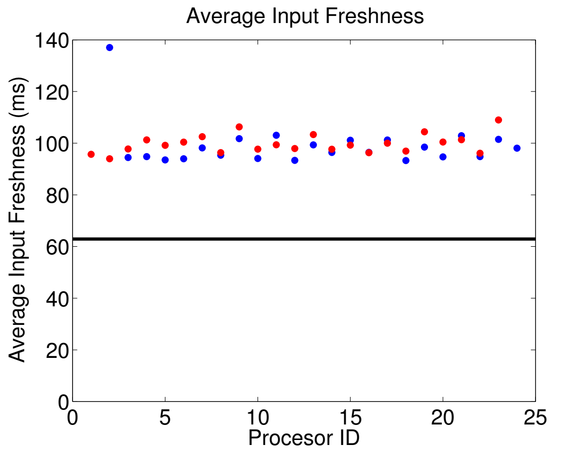


Figure 7: Example 2.2. Left, average processor interupdate times. Right, averaged input freshness, upwards in blue, downwards in red. Black lines show average values for synchronous execution.

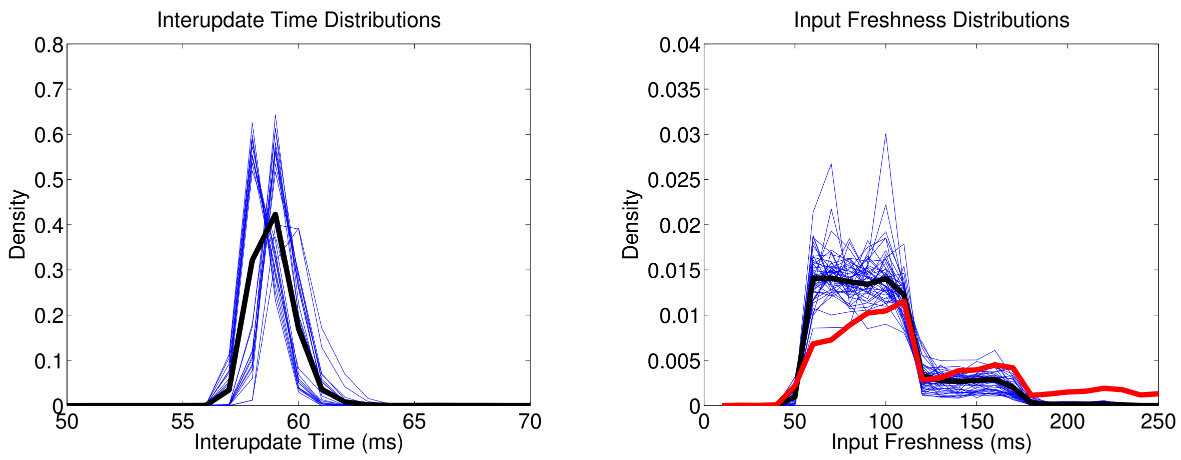


Figure 8: Example 2.2. Left, sampled distribution of interupdate times for each of the 24 processors. Group average of 1-12 bold green, 13-24 bold red. Right, sampled distributions of communication times. Upwards in blue, downwards in red. Bold yellow is overall distribution of downwards communication, bold black upwards.

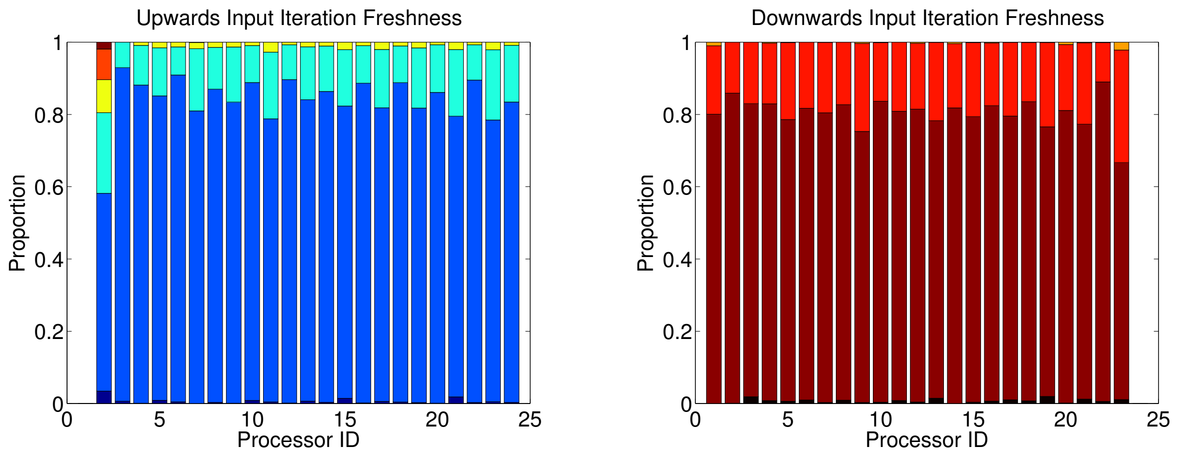


Figure 9: Example 2.2. Stack column charts showing iteration freshness. Darkest regions represent zero delay with delay increasing by one with each new region. Upwards communication in blue on left, downwards communication in red on right.

In spite of the slow communication between processors 1 and 2, these results are far more like we would expect. Since the processors update independently they should appear to be randomly out of phase with each other so that whenever there is a communication with iteration freshness 1 the time taken for the communication should be uniformly distributed on (60ms,120ms), likewise if it has iteration freshness 2 then it will be uniformly distributed on (120ms,180ms) and so on.

We conclude that in this execution the individual processors are updating independently and not having to wait for any sort of communication from each other. Then as we would expect the mass update rate is higher. The poor performance of the communication from processor 1 to processor 2 remains a mystery for now. We shall see that for this example problem the performance of the communication channels is far less important than the individual processor update rates, which is why the asynchronous execution has a greater effective update rate than the synchronous one in this case.

3 Theory for non-negative matrices

Before we can state and prove our theoretical results we need to introduce a few definitions.

Suppose that the problem $Ax = b$ has Jacobi iteration matrix M, which we iterate asynchronously. Let x^* denote the solution to the problem.

Since a processor may record different values for another processor's state at a particular time we must be a little careful in defining the state of the system at time t. The state of the system at time t, denoted $x[t]$, is given by the direct product of each of the processor's most recently computed values. We label individual coordinates of x with a subscript and blocks of coordinates (as stored by the individual processors) with a superscript. In order to talk about the state of a processor or coordinate after a certain number of updates we use parentheses so that $x_1(2)$ is value of coordinate 1 after its second update and $x_1[4.214]$ is the value of coordinate 1 after 4.214 seconds.

As well as working directly with the logged DAG graphs that we analysed previously we will also use an expansion that we call the *calculation graph*. The calculation graph is much like the DAG graph but rather than associating vertices with processors we associated vertices with individual coordinates. Single processor updates are replaced by many coordinate updates, likewise a single edge in the DAG will correspond to many edges in the calculation graph.

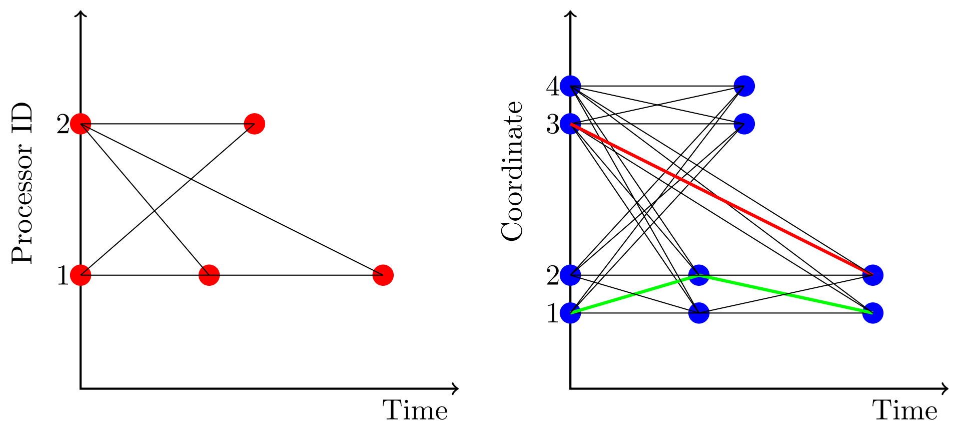


Figure 10: Example 3.1. Left, processor level DAG. Right, expanded calculation graph. Coloured edges correspond to coloured terms in the matrix below.

In order to reason about the rate of convergence of an asynchronous iteration we define the error operator $\Omega(C, M, t)$, which is the unique matrix with

$$\Omega(C, M, t)(x(0) - x^*) = x[t] - x^*,$$

for all possible initial conditions $x(0)$. In the synchronous case this would simply be the iteration matrix M raised to the power of the number of iterations performed by time t but in the asynchronous case it is a little more complicated. Note that the error operator depends on the iteration matrix, the calculation graph and the time.

Example 3.1 Suppose that we are iterating the map $x \mapsto Mx + b$ on a two processor machine, where each processor is responsible for two coordinates. A possible DAG and corresponding calculation graph C are displayed in Figure 10.

If the initial state in processor 1 is $x^1(0) = (x_1, x_2)$ and the initial state in processor 2 is $x^2(0) = (x_3, x_4)$ then the state in processor 1 after its first update is

$$x^1(1) = M_{1,1}x^1(0) + M_{1,2}x^2(0) + b^1,$$

the state in processor 2 after its first update is

$$x^2(1) = M_{2,1}x^1(0) + M_{2,2}x^2(0) + b^2,$$

and the state in processor 1 after its second update is

$$x^1(2) = M_{1,1}x^1(1)+M_{1,2}x^2(0)+b^1 = M_{1,1}(M_{1,1}x^1(0)+M_{1,2}x^2(0)+b^1)+M_{1,2}x^2(0)+b^1.$$

As in the synchronous case we can decouple the solution from the error. Let $y = x - x^*$, where x^* is the solution to the problem. Then the behaviour of the error y is also determined by the DAG. The error in the state at processor 1 after its first update is

$$y^1(1) = M_{1,1}y^1(0) + M_{1,2}y^2(0),$$

the error in the state at processor 2 after its first update is

$$y^2(1) = M_{2,1}y^1(0) + M_{2,2}y^2(0),$$

and the state in processor 1 after its second update is

$$y^1(2) = M_{1,1}M_{1,1}y^1(0) + M_{1,1}M_{1,2}y^2(0) + M_{1,2}y^2(0).$$

Notice that for every path through the DAG graph from processor i's nth update back to processor j's initial condition there is a term in i's nth error expression which is given by the product of the submatrices corresponding to those edges in the path and the initial condition at j. For example from 1's second update there are two paths to 2's initial condition: one of length two which gives the term $M_{1,1}M_{1,2}y^2(0)$ and one of length one which gives the term $M_{1,2}y^2(0)$.

The same is true for the calculation graph C, so we can express the value of the error for a particular coordinate at a particular stage as a sum over the paths through the graph. If we compute this sum for each coordinate we arrive at the error operator, $\Omega(C, M, t) =$

$$\begin{bmatrix}
\begin{matrix} M(1,1)M(1,1) \\ +M(2,1)M(1,2) \end{matrix} & \begin{matrix} M(1,2)M(1,1) \\ +M(2,2)M(1,2) \end{matrix} & \begin{matrix} M(1,3)M(1,1) \\ +M(2,3)M(1,2) \\ +M(1,3) \end{matrix} & \begin{matrix} M(1,4)M(1,1) \\ +M(2,4)M(1,2) \\ +M(1,4) \end{matrix} \\
\begin{matrix} M(1,1)M(2,1) \\ +M(2,1)M(2,2) \end{matrix} & \begin{matrix} M(2,1)M(2,1) \\ +M(2,2)M(2,2) \end{matrix} & \begin{matrix} M(1,3)M(1,2) \\ +M(2,3)M(2,2) \\ +M(2,3) \end{matrix} & \begin{matrix} M(1,4)M(2,1) \\ +M(2,4)M(2,2) \\ +M(2,4) \end{matrix} \\
M(3,1) & M(3,2) & M(3,3) & M(3,4) \\
M(4,1) & M(4,2) & M(4,3) & M(4,4)
\end{bmatrix}$$

where t is any time after processor 1's second update. Each term in the matrix corresponds to a unique path through the calculation graph C. Two terms have been coloured along with their corresponding paths through C in Figure 10.

The following results will be useful in our analysis.

Lemma 3.2 (Properties of operator norm) *Let*

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2},$$

denote the l^2 operator norm.

1. *If A is a non-negative matrix and B is a submatrix of A then $\|A\| \geq \|B\|$.*
2. *If A and B are non-negative $d \times d$ matrices then $\|A+B\| \geq \max\{\|A\|, \|B\|\}$.*
3. *The sequence $\|A^k\|_{k=1}^{\infty}$ is eventually monotonic.*
4. $\lim_{n\to\infty} \|A^k\|^{\frac{1}{k}} = \rho(A)$.

Items 1 and 2 are trivial, and 3 and 4 follow by considering the Jordan form of A [8].

Theorem 3.3 (Weakest link performance barrier) *Suppose that the problem $Ax = b$ has non-negative Jacobi iteration matrix M, which we iterate asynchronously with calculation graph C. Let $M_{i,i}$ be the block submatrix iterated by processor i and let $a_i(t)$ be the the number of times that processor i has updated by time t. We have the bound*

$$\|\Omega(C, M, t)\| \geq \|M_{i,i}^{a_i(t)}\|,$$

so that for generic initial condition $x(0)$

$$\limsup_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \geq \max_i \alpha_i^* \log[\rho(M_{i,i})],$$

where

$$a_i^* = \lim_{t\to\infty} \frac{\alpha_i(t)}{t},$$

is processor i's update rate, if this limit exists.

The submatrix $\Omega(C,M,t)_{i,i}$ can be expressed as a sum of products of matrices taken over all the paths through the DAG (not calculation graph) from i's initial condition to its current state

$$\Omega(C,M,t)_{i,i} = \sum_{\sigma \in \mathrm{DAG}: i \mapsto i} \prod_{k=1}^{|\sigma|-1} M_{\sigma(k+1),\sigma(k)}.$$

This sum contains the term $M_{i,i}^{a_i(t)}$, which corresponds to the path that moves through i's successive updates, so by non-negativity we have

$$\Omega(C,M,t)_{i,i} \geq M_{i,i}^{a_i(t)},$$

where $a_i(t)$ is the the number of times that processor i has updated by time t and $\geq$ means greater than or equal in each component. From items 1-2 from the lemma we have

$$\|\Omega(C,M,t)\| \geq \|\Omega(C,M,t)_{i,i}\| \geq \|\|M_{i,i}^{a_i(t)}\|.$$

For the second part of the theorem we need to take the SVD of the error operator

$$USV^T = \Omega(C,M,t),$$

where U,S,V are parametrized by C,M,t, and the diagonal entries of S are ordered in decreasing modulus. Now

$$x[t]-x^* = \Omega(C,M,t)(x[0]-x^*) = U_1 S(1,1)\langle x[0]-x^*, V_1\rangle + \text{terms perpendicular to } U_1,$$

and using the fact that $S(1,1) = \|\Omega(C,M,t)\|$, we have

$$\|x[t]-x^*\| \geq \|U_1 S(1,1)\langle x(0)-x^*, V_1\rangle\| \geq \|M_{i,i}^{a_i(t)}\|\langle x(0)-x^*, V_1\rangle.$$

Since the SVD of the error operator is a function of t we need to be a bit careful here. To make the time-dependence explicit we now include the parametrization in the notation. Consider the set

$$K = \{x : \lim_{t\to\infty} \langle x, V_1(t)\rangle = 0\},$$

clearly this set forms a subspace of $\mathbb{R}^d$. Suppose that $K = \mathbb{R}^d$, then $\lim_{t\to\infty} V_1(t) = 0$, which is a contradiction since $\|V_1(t)\| = 1$ for all t. Therefore K is a proper subspace and for a generic initial condition $x(0)$, $x(0)-x^*$ will not lie in K. So that there exists non-zero ϵ such that

$$\limsup_{t\to\infty} |\langle x(0)-x^*, V_1(t)\rangle| = \epsilon.$$

Therefore using result 4 from the lemma we have

$$\limsup_{t\to\infty} \frac{\log \|x[t]-x^*\|}{t} \geq \limsup_{t\to\infty} \frac{\log \|M_{i,i}^{a_i(t)}\| + \log \epsilon}{t} = \lim_{t\to\infty} \frac{a_i(t)}{t} \log \rho(M_{i,i}),$$

the full result follows by taking the maximum over all of the processors, i.e. the maximum over i.

Theorem 3.3 shows that the performance of APJ is always held back by its slowest sub-system. In our next result we show that there is a limit to how poorly APJ can perform, which is determined by a slowest path through the calculation graph.

Theorem 3.4 (Shortest path performance guarantee) *Suppose that the problem $Ax = b$ has non-negative Jacobi iteration matrix M, which we iterate asynchronously with calculation graph C. Let $s(t)$ be the length of the shortest path (in terms of number of edges) through C from an initial condition (that is an update with local iteration number 1) to a value at t (that is an update for which there is not another for the same coordinate with a greater local iteration number before time t). Likewise let $l(t)$ be the longest such path. We have the bound*

$$\|x[t]-x^*\| \leq \| \sum_{k=s(t)}^{l(t)} M^k \| \|x(0)-x^*\|.$$

and

$$\lim_{t\to\infty} \frac{\log \|x[t]-x^*\|}{t} \leq \lim_{n\to\infty} \frac{s(t)}{t} \log[\rho(M)] = s^* \log[\rho(M)],$$

where

$$s^* = \lim_{n\to\infty} \frac{s(t)}{t},$$

is the shortest path growth rate, if this limit exists.

The components of $\Omega(C, M, t)$ can be expressed as sums over the paths through the calculation graph C,

$$\Omega(C, M, t)(i, j) = \sum_{\sigma \in C: j \to i} \prod_{i=1}^{|\sigma|-1} M[\sigma(i+1), \sigma],$$

where the sum is taken over all paths σ through C from j's initial condition $x_j(0)$ to i's most recent update at time t.

Consider the matrix

$$\sum_{k=s(t)}^{l(t)} M^k.$$

We can expand its coordinates in terms of path weights

$$[\sum_{k=s(t)}^{l(t)} M^k](i,j) = \sum_{\varsigma \in \widehat{C}:j\to i} \prod_{i=1}^{|\varsigma|-1} M[\varsigma(i+1), \varsigma],$$

where the sum is taken over the set of all sequences of length between $s(t)$ and $l(t)$, which start at j and end at i. Therefore since the components of the error operator are sums over subsets of this set, and since all the terms being summed are non-negative, we have

$$\Omega(C, M, t) \leq \sum_{k=s(t)}^{l(t)} M^k,$$

where $\leq$ means less than or equal in every component. From result 2 of the lemma

$$\|\Omega(C, M, t)\| \leq \| \sum_{k=s(t)}^{l(t)} M^k \|,$$

and therefore

$$\|x[t] - x^*\| \leq \| \sum_{k=s(t)}^{l(t)} M^k \| \|x(0) - x^*\|.$$

Now to calculate the convergence rate implied by this bound

$$\lim_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \leq \lim_{n\to\infty} \frac{\log \| \sum_{k=s(t)}^{l(t)} M^k \|}{n} + \frac{\log \|x(0) - x^*\|}{n},$$

the second term converges to zero

$$\lim_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \leq \lim_{n\to\infty} \frac{\log \| \sum_{k=s(t)}^{l(t)} M^k \|}{n}$$

$$\leq \lim_{n\to\infty} \frac{\log[l(t) - s(t) + 1]}{n} + \frac{\log \max_{k=s(t)}^{l(t)} \|M^k\|}{n},$$

the first term converges to zero since s and t grow at most linearly. From result 3 in the lemma the maximum in the second term is, for sufficiently large t, attained by $k = s(t)$ so that

$$\lim_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \leq \lim_{n\to\infty} \frac{\log \|M^{s(t)}\|}{n} = \lim_{n\to\infty} \frac{s(t)}{t} \log[\rho(M)] = s^* \log[\rho(M)],$$

if this limit exists.

Corollary 3.5 *Suppose that the problem $Ax = b$ has (not necessarily non-negative) Jacobi iteration matrix M and take everything else as in the statement of theorem 3.4. We have the bound*

$$\|x[t] - x^*\| \leq \| \sum_{k=s(t)}^{l(t)} |M|^k \| \|x(0) - x^*\|.$$

and

$$\lim_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \leq \lim_{n\to\infty} \frac{s(t)}{t} \log[\rho(|M|)] = s^* \log[\rho(|M|)],$$

where

$$s^* = \lim_{n\to\infty} \frac{s(t)}{t},$$

is the shortest path growth rate, if this limit exists.

This follows by considering the two different error operators $\Omega(C, M, t)$, and $\Omega(C, |M|, t)$. Each is a sum over paths through C, but the sum in $\Omega(C, |M|, t)$ is taken over the absolute value of the terms in the sum of $\Omega(C, M, t)$. So that

$$|\Omega(C, M, t)_{i,j}| \leq \Omega(C, |M|, t)_{i,j},$$

for all i, j. So, from the lemma, the operator norm of the error operator of the system with the matrix of absolute values gives an upper bound on that of the original system and the corollary follows from theorem 3.4.

4 Weakest link scenario

To apply these bounds to our examples we need to compute the processor update rates α_i as well as the shortest path growth rate s^*. The update rates are

$$\alpha_i = \frac{\text{numer of times processor } i \text{ updates during execution}}{\text{execution time in seconds}}.$$

The shortest path growth rate is a little more complicated. Since the iteration matrix M is tridiagonal with zeros on its diagonal, a path through the calculation graph can either move up one coordinate index or down one coordinate index at each step. This is unlike in the DAG where a path can either stay at the same processor or move up one processor or down one processor with each step. Since each processor stores many coordinates this means that the paths through the calculation graph are far more restricted in their movement than those through the DAG. This was our reason for using the calculation graph in theorem 3.4; a similar result could easily be proved for the DAG but the bound would be less sharp. Using the calculation graph enables us to exploit the sparsity pattern in the matrix.

Now, since edges corresponding to inter-processor communication tend to be longer than those corresponding to intra-processor communication, and since it would take thousands of intra-processor steps for a path to move across a processor, it follows that the shortest path will be one that repeatedly crosses back and forth between two adjacent processors.

We call the crossing between two adjacent processors i and $i+1$ the ith communication channel.

To compute the shortest path growth rate associated with the ith channel we construct a path backwards through the DAG starting at processor i's final update, then to its input from processor $i+1$, then to that update's input from processor i and so on until we arrive at either processor i or $i+1$'s initial condition.

The shortest path growth rate associated with channel i is then given by

$$s_i = \frac{\text{number of edges in constructed path}}{\text{execution time in seconds}},$$

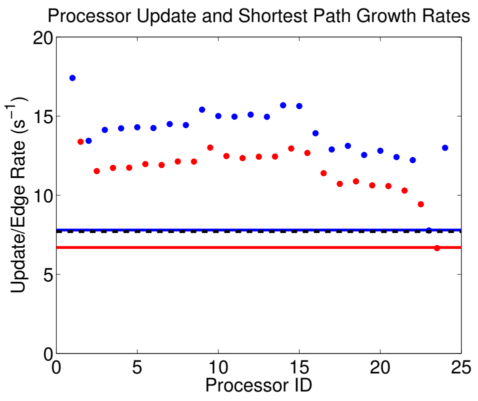
and $s^* = \max_i s_i$. This quick heuristic method yields exactly the same result and identifies the same shortest path as a considerably more expensive method which considers all possible paths through the calculation graph.

Figure 11 shows the computed values for the update rates and communication channel shortest path growth rates.

Example 4.1 (Example 2.1) The slowest processor was processor 23 with an update rate of 7.8 so from theorem 3.3 (the weakest link performance barrier) we have

$$\limsup_{t\to\infty} \frac{\log \|x[t] - x^*\|}{t} \geq 7.8 \log[\rho(M_{23,23})],$$

where $\rho(M_{23,23})$ is the Perron root of processor 23's iteration matrix. We can convert this into a bound on the effective update rate by using



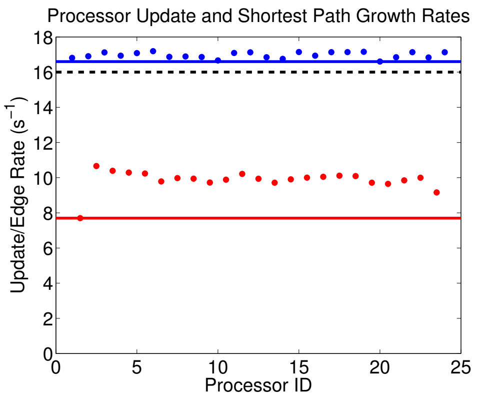


Figure 11: Left, example 2.1. Right, example 2.2. Processor update rates (blue) and communication channel shortest path growth rates (red). Effective update rates of executions in black.

$$\text{effective update rate} = \frac{\text{rate of convergence}}{\log \rho(M)},$$

where $\rho(M)$ is the Perron root of the entire iteration matrix. Note that the rate of convergence is minus the Lyapunov exponent, so that

$$\text{effective update rate} \leq 7.8 \frac{\log \rho(M_{23,23})}{\log \rho(M)},$$

from [7] we have

$$\rho(M) = \tfrac{2}{2.001}\cos(\tfrac{\pi}{20000001}), \quad \rho(M_{23,23}) = \tfrac{2}{2.001}\cos(\tfrac{\pi}{8340001}),$$

so that the ratio of the logs is very close to 1. The slowest communication channel was the one between processor 23 and processor 24 with shortest path growth rate was 6.7, so from theorem 3.4 (shortest path performance guarantee) we have

$$\lim_{t\to\infty} \frac{\log \|x[t]-x^*\|}{t} \leq 6.7 \log \rho(M),$$

so that

$$7.8 \geq \text{effective update rate} \geq 6.7,$$

which provides a fairly tight sandwich of bound for the effective update rate that we originally calculated directly from the execution time to be 7.5.

Example 4.2 (Example 2.2) Applying the same bound in the same way to the execution of example 1.2 we obtain

$$16.6 \geq \text{effective update rate} \geq 7.7.$$

which gives a tight upper bound but poor lower bound on the effective update rate that we calculated from the execution time to be 16.0.

In both examples the weakest link performance barrier provides a very good approximation for the effective update rate. We expect that this will be a fairly common situation. If the typical norm of the communicated data is very small compared to the norm of the processors' states then the rate of convergence ought to be determined by the rate of convergence of the slowest processor sub-system (in our examples the communicated data is a single coordinate whereas the state of a processor is many thousands). What's more in this low communication scenario there will be submatrices $M_{i,i}$ with Perron root very close to that of the whole iteration matrix so that

$$\frac{\log \rho(M_{i,i})}{\log \rho(M)} \approx 1,$$

which means that we can replace the bound with the approximation

$$\text{effective update rate} \approx \min_i \alpha_i.$$

We refer to the situation where this approximations hold for the reasons outlined above as the *weakest link scenario*.

The shortest path performance guarantee gave a tight bound for example 2.1 but a poor bound for example 2.2. In example 2.1 the poor performance of processor 23 also results in the 23-24 communication channel being very slow. If one processor is much slower than its neighbour then we might expect the communication channel growth rate to be roughly equal to the update rate of the slower processor, as is the case here. This explains why we get such a tight bound in this case.

In example 2.2 there is a slow communication channel between processors 1 and 2. However this does not have a dramatic affect on the effective update rate as the size of the communicated data is so small compared with the size of the processors' states.

It is possible for an execution to have effective update rate close to the shortest path performance guarantee without being close to the weakest link performance barrier. These examples require the iteration matrix to be unstructured so that the size of the input from each communication channel is comparable to the size of the processor's state. In this scenario the bound is particularly sharp if the spectral radius of the iteration matrix is small.

5 Discussion

We have presented upper and lower bounds for the effective update rate of APJ. Our experimental results agree with out theory and in our examples, where communication was less important, the weakest link performance barrier provides a good approximation for the effective update rate. We conjecture that this will be a fairly typical scenario even in the general (not just non-negative) case.

Although our shortest path performance guarantee was less useful as an approximation than the weakest link performance barrier, it is arguably the more important result. It is valid in the general case and performance guarantees are more useful than upper bounds on the rate of convergence. We conjecture that shortest path performance guarantees should be possible to prove for a wide variety of asynchronous algorithms.

We also made a detailed study of the DAGs associated with two different executions of APJ. In our first example the asynchronous execution was slow compared to the synchronous one. We resolved this by switching to MPI-1 communication routines and obtained a DAG which better resembled what one would expect from the theory and, crucially, outperformed the corresponding synchronous execution.

We would advise caution when assuming that an asynchronous algorithm will behave in a particular way. One might naïvely assume that the execution of an asynchronous algorithm can be modelled with i.i.d. interupdate times with i.i.d. communication delays, but neither of our traces resembled output from such a simple model.

We certainly did not expect to see the problems with processor 23 in our first execution and still have not found a satisfactory explanation for the poor performance of the 1-2 communication channel in our second example.

One avenue for further research would be taking an even more detailed recording of the execution and log exactly when input data arrives in memory, when it is checked for, how it is transmitted and so forth. Such an experiment would require novel instrumentation to avoid introducing an overhead that interferes with the natural dynamics.

References

[1] D. Chazan, W. Miranker, Chaotic relaxation, Linear Algebra and its Aplications 2 (1969) 199–222.

[2] J. Bull, T. Freeman, Numerical performance of an asynchronous jacobi iteration, in: Parallel Processing, Vol. 634 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1992, pp. 361–366.

[3] I. Bethune, J. Bull, N. Dingle, N. Higham, Performance analysis of asynchronous jacobis method implemented in mpi, shmem and openmp, International Journal of High Performance Computing Applications.

[4] A. D. H. Avron, A. Gupta, A randomized asynchronous linear solver with provable convergence rate, arXiv preprint-arXiv:http://arxiv.org/pdf/1304.6475.pdf.

[5] P. Richtárik, M. Takáč, Lock free randomized first order methods, manuscript.

[6] J. Lu, Y. Tang, Distributed asynchronous algorithms for solving positive definite linear equations over dynamic networks, arXiv preprint-arXiv:http://arxiv.org/pdf/1306.0260.pdf.

[7] R. Gregory, D. Karney, A Collection of Matrices for Testing Computational Algorithms, Wiley, 1969.

[8] R. Horn, C. Johnson, Matrix analysis, Cambridge University Press.