Parallel Assignment 2: Distributed Computation

Alex Birch – akb29, 19/12/13

Disclaimer: I re-use text from my own Parallel Assignment 1 where the content still applies.

# Problem Description

The program provided will take a grid of specified size, and ‘relax’ it iteratively (each value being set to the average of its cardinal neighbours) until such time that no value changes more than the specified precision. Edge values do not get averaged. The whole grid consists of 0s, excepting the first row and column, which will be populated with 1s.

In general, value flows from the edges toward the middle. If visualised as heat, the edges of 1s could be considered ‘hot’ and the edges of 0s considered ‘cold’. Since edge values never change, the 1s and 0s could be considered to represent an infinite amount of heat transfer in and out of the system, respectively.

The initial 0ing of the middle of the grid is insignificant in the long run; ‘heat’ and ‘cold’ are provided in infinite amounts from the edges, overwhelming any transient contribution the starting values have. The system converges on equilibrium, where cold meets hot in various middles. There is a slight bias overall toward heat, since there are more 1s on the edge than 0s.

The higher the precision (that is, a lower number), the closer we get to seeing the true equilibrium. For low precisions, the problem can complete well before the flow of heat has a chance to meet the flow of cold. That is, if heat is carried one cell per iteration, with each new value being a fraction of the previous one, heat values lower than the precision of the problem can be in-flight yet disregarded as progress, given no chance to be reinforced by their chain of hotter neighbours.

Relaxation can be used to numerically solve systems of equations (such as linear inequalities and elliptic partial differential equations). Correctness of results could be verified via comparison to other methods of solving the same mathematical problems.

The goal of this assignment is to parallelise the solution of this problem, using message passing in MPI. The computation will take place across multiple computers (in this case cluster nodes), and/or multiple times on each computer recruited (one process instance per processor core).

# Compilation

There is only one file; weekend.c. The filename is non-random, and suggests a context in which many agents can relax together. It is another relaxation joke.

There are a couple of macros you can define or undefine to change behaviour…

Program can be compiled to use a matrix of floats, or a matrix of doubles:

// whether to use floats (0) or doubles (1)

#define VALCHOICE 0

Verbose logging can be turned on (this is slow and makes the logs large). It prints the matrix at the end of each iteration.

// Toggle verbose logging  
#define verbose

Even printing the matrix just once can be expensive.

// omit the print if you are doing performance tests or want small logs  
#define printfinalresult

Compile like so:

mpicc -I/usr/local/include -O0 -g3 -Wall -c -fmessage-length=0 ./weekend.c

Then link, like so:

mpicc -o "weekend" ./weekend.o -lmpi –lm

Then, if necessary, add execute privileges to the output executable

chmod +x weekend

# Usage

Invoke (non-distributed) as follows:

./weekend [int gridSize] [float prec] ([int cacheSize])

Allowed boundaries:

./weekend [int 3~65535] [float 0.000001~0.1] ([int 2~procs])

Example usages:

./weekend 100 0.005

./weekend 100 0.005 2

To invoke as a distributed computation, wrap in an mpiexec call:

mpiexec -np 2 ./weekend 100 0.005 2

For following program output, the most sane visualization is provided by separate xterms per process:

mpiexec -np 4 xterm -e "./weekend 100 0.005 2; read"

Aquila runs should specify the appropriate machinefile:

mpiexec -np 2 -machinefile $PBS\_NODEFILE ./weekend 100 0.005 2

PBS Job scripts should recruit the desired number of nodes, and processor cores per node (4\*8 example):

#PBS -l nodes=4:ppn=8

Grid magnitude minimum is based on the minimum number of writable rows that can be allocated. Maximum is informed by the integer size used for indirecting into the matrix. Precision limits were chosen arbitrarily.

An algorithm-specific (optional) parameter that was introduced is ‘cacheSize’. Each process computes some number of iterations before sharing their results (these iterations remain in memory). This is memory-intensive, but theoretically can reduce the amount of message passing required to find the first relaxed iteration.

If any relaxed iteration would suffice as an answer, then a similar algorithm could be made that still shares results on nth iterations, but only ever stores the latest iteration. No support is provided in my program for that, though; a minimum cache size results in sharing results every iteration.

By default, cacheSize is set to maximum (number of processes), but it can be set to minimum (2) or anything in-between. The reason 'number of processes' happens to be the maximum, is that this equals the number of different iterations that can be being worked on at once. A process can compute a newer iteration if its neighbours have completed their own iteration (and those neighbours' neighbours have done the same, and so on). Thus the number of different iterations that can be being worked on, is equal to the number of processes that can depend on another process, plus the remaining process that depends on nobody. This equals, maximally, the number of processes.

# Algorithm - Division of problem

## Initial division of matrix territory

Upon starting computation, each process claims a fair share of the matrix, doled out by process rank, from top to bottom. Segments are not sent out by message; rather, each process generates their own portion, using knowledge of the problem. It is agreed that the matrix will be filled with 0s, excepting the first row and column, which will be 1s. Thus, with the exception of the first process (responsible for the top edge of 1s), all processes generate a similar initial matrix portion. These differ only by the number of rows used.

Every process uses the same formula to work out which matrix rows a process of their rank should start and end on. Should the rows not divide evenly, then remainder rows are allocated to the first few ranks until all are accounted for.

Each process is responsible for writing some unique, fair portion of the matrix. However, since all values are calculated based on neighbouring values, the processes have a responsibility to read the edges they share with their neighbours (message passing is used here; the mechanism will be explored later).

The top and bottom rows are not claimed for writing by any process, since they are read-only in the context of the problem. However the closest process to them is responsible for sending these rows to be printed at the end of the problem.

It occurs to me now that communicating the top and bottom rows is an unnecessary complication; the rows could instead be generated trivially. But since the latency of message passing must be incurred for final print anyway, sending this extra data isn't much more expensive.

## Row allocation effects

Rows are allocated contiguously. A few effects are conceivable from this decision:

Processes in more stable zones of the matrix will reach relaxation sooner than other processes (whereas interleaving the rows across the matrix would mean all processes are equally affected by localized chaos).

Ideally process territory would be allocated in a manner that makes all of them notice as early as possible that they are not relaxed (ie, each should be given an equal number of rows that are in areas expected to be more chaotic).

Since processes need to sync up to check each other for completion criteria, speeding up one process does little good; all processes wait for the slowest process. So equal speedup is more desirable than cumulative speedup. This is also why it is important to give each process a similar amount of work (in fact, there might be small advantages to giving processes fractions of rows, despite the complexity, if it means that we divide the cells out more fairly).

Additionally, if each process is allocated to a different processor core, they each have access to their own memory cache. Since data is fetched blockwise into cache, it is faster to fetch contiguous memory than sparse[[1]](#footnote-1). This is also why the matrix is structured as row-major rather than column-major. Values are calloc()ated as a 1D array, but for legibility and programming simplicity we access rows by indirecting through pointers, which is admittedly slower than using just arithmetic.

## Concerning in-place edit of matrix

### Fairness

In-place edit introduces some concerns. For one thing, the direction in which values are relaxed changes the manner in which results are reached; should values be relaxed top-to-bottom (for example), heat from the top row is carried to the next row, and also from that row to its next row and so on. In this way, heat can travel the whole length of the matrix in one iteration (whereas this wouldn't happen so early in a bottom-to-top configuration).

Moreover, the values of rows that have been relaxed this iteration (and thus effectively belong to a subsequent iteration) influence the values of proceeding rows, effectively meaning that each row is calculated in a different way. The validity of this method is questionable, but it would still ultimately converge on equilibrium. In any case, this is unrelated to parallelism so far.

Regardless of approach, in general all cells will still tend toward their required values, and due to convergence the value difference between two iterations eventually becomes insignificant anyway. It is unclear whether this rampancy would have a significant effect on the number of iterations required.

If one wanted to reduce the effect of neighbour operation order, cells that do not share neighbours could be evaluated first (for example evaluation in non-contiguous diagonal stripes), but eventually those cells’ neighbours will have to be evaluated also.

### Use of neighbour data

Race conditions concerning the reading of values that a neighbour is currently writing to (ie, shared edges), are less of a concern for distributed-memory; explicit negotiation is required to fetch these values, so neighbour processes can send (in their entirety) required rows when they are ready, at an agreed point in program execution (for example, the start of the iteration).

Thus at least this can be made deterministic; a process can agree to wait for neighbour information before relaxing the row that touches it.

## Editing via separate matrices

Separate ‘read’ and ‘write’ matrices can be used. This ensures that there exists an authoritative view of all values for some iteration (that is, the ‘read’ matrix, which won’t change mid-iteration). This removes the fairness and determinism issues described with the in-place edit; irrespective of how the matrix is traversed or how many threads work on the problem, the same values will be read from the ‘read’ matrix, and the same values will be written to the ‘write’ matrix.

Separate edit does require more memory, since an extra matrix is required. But it removes bias in the relaxation direction of the matrix (which may be a desirable property, or have an impact on the number of iterations required).

Memory is not expended on creating new matrices to write to each iteration, as previous matrices can be re-used; the role of ‘read' matrix and ‘write’ matrix swaps on each iteration, with whichever one is newest becoming the one that values are read from (until it becomes superseded).

Separate edit looks to be simpler to implement, more predictable (and by extension more testable). So this was the solution chosen.

## On ‘work pools’

Allocating rows fairly does mean that, should any process finish their workload significantly faster (for example, if they find out quickly that their territory is not relaxed, and can fast-fail the rest of the checks), that process has no opportunity to take on further work to reduce its wait time.

Instead of doling out all rows at the start, a ‘pool’ of rows could be fetched as and when processes have leeway to take on more work. Ideally small grains of task should be available for those processes that have already done some work, since the process they would otherwise be waiting on, might end up waiting on *them* instead, as they take on their second task.

Deciding on the grain size for work to ensure that taking extra tasks don’t make you become the process that others wait on is a difficult task, and requires knowing which zones of the matrix are likely to take longer to work on (this is influenced by the ‘fast-fail’ of calculating whether the area is relaxed).

There should of course be more grains than processes, since if it is assumed that there will always be a slowest process that others wait on, then having extra work available is always desirable. Likely it is a good idea to have the ‘bonus work’ be in small grains (so long as overhead of fetching them is low) so that waiting on a thread who takes on extra work, wouldn’t take long.

Crucially, though, fetching from the pool would require some choreography between the processes. Though this could have been acceptable in shared-memory, the communication time would likely not be worth it in distributed computation; so much time is wasted waiting for work that it could be better to just wait for a slow process to finish its fair load.

Work pooling also makes it harder to tell when the iteration has been finished; it is no longer a case of “everyone waits after doing the planned amount of work”, and instead becomes “are all the jobs taken, and are all the jobs now finished”. Again, in distributed computation, this could introduce an unacceptable reliance on communication.

I did not implement work pooling, since it is difficult to do right, highly communication-dependent, and would make the code more complex than it needs to be.

# Algorithm - Working in Parallel

## Distributed-memory computation concerns

Communication between computers is (generally) more time-expensive than communication within a computer. The expense is more one of latency than of bandwidth. That is, if one is sending a message, then one may as well send as much useful data as possible, since the crucial latency cost has already been paid.

### Compensating for communication costs by reducing data dependency

Data dependency becomes problematic, since the more data you depend on, the more messages need to be sent. In general it is a good idea to reduce dependency on data. For example, if you can compute something yourself faster than the result could be sent to you, then computation becomes a better option than communication.

In this particular problem, we do not need to know the rest of the matrix to start computing the next iteration correctly; the minimum amount of data we need are those values that neighbour ours (so instead of communicating with all processes for matrix updates, we need only ask our immediate neighbours).

Not only is it the case that we are only interested in neighbour processes - we are only interested in the row of theirs that shares an edge with ours. As a result, less data could be sent (for all the difference it makes). Although, if more rows were sent, we could compute more iterations from the data (by computing also the values of rows that would eventually spread to us). But if extended to infinity, you realise you are relaxing the whole matrix on your own, and the exercise becomes meaningless. It also increases memory usage and program complexity, so I did not implement this particular optimization.

One notable omission we can make from our data dependency, is that we do not need to check for problem completion criteria every iteration - this requires all nodes to wait and synchronise at the same point of program execution, and pool data from all nodes. Instead, we compute a few iterations individually, then check completion criteria of all those iterations at once. Though this does mean that in the final stage of the problem we can overshoot the answer, and do several iterations of un-needed computation, it is hoped that the problem will last long enough that overshooting at the end isn't expensive. As mentioned earlier in 'Usage', we could reduce the amount of data sent by analyzing only the latest iteration, but there is little to gain from this, as the latency is a sunken cost by this point.

If meaningful progress can still be gained without receipt of required data, the computation could be worth pursuing. For example, if waiting for receipt of a neighbour's row would be a burden, perhaps an extra iteration could be calculated based on the out-of-date row, which could serve as a heuristic for the real answer, even if it is otherwise erroneous.

Since the problem is all about convergence to equilibrium, so long as the majority of values get closer to their equilibrium value, progress is made. Error would have to be smoothed out in subsequent computation, but this could require less computation (or time) than avoiding the error. I did not use any error-embracing methods, though, as it is hard to predict the influence they have on speedup (too much error can take you further from the answer), and it would become harder to tell during testing whether the matrix was progressing correctly.

### Compensating for communication costs by doing simultaneous communications

Since there is a considerable latency associated with data transfer, we don't want to be wasting time waiting for the message to arrive. Thus, we should utilise non-blocking message receipt and sending, to allow us to do computation in the meantime.

One aspect of this is enabling multiple messages to be in flight at once, rather than waiting for each one to come back before starting the next. In my implementation, a process depends each iteration on row receipt from its immediate neighbour processes. It invokes non-blocking receipt for both the rows it needs to receive, as well as non-blocking sends for both the rows it needs to send. It is reliant on at least the receives before it can begin the next iteration, so it waits for these. But already this is a huge improvement over blocking message passing; as many as four calls are initiated, so all messages could be in flight at once, before blocking occurs. In fact, it is a very long time before we are blocked on our own sends - we compute an entire iteration before we need to re-use the buffers (and if this block was still a problem, we could make buffers on a per-iteration basis).

### Compensating for communication costs by doing computation during message transit

To benefit further from non-blocking ('immediate') message passing, data requirements need to be known sufficiently far in advance that we don't run out of work during the transfer window. Blocking can be avoided entirely, if the time can be filled with computation. Admittedly I could make use of this in my algorithm.

For example, we could send our starting row to our neighbours almost immediately, as it is one of the first things we compute. Arguably this is futile, as we would always be waiting on the end row no matter how early we receive the start row. But a subtlety exists: in cases where we share a processor with our proceeding process, but not with our preceding one, the row that is computed first could easily take longer to arrive than the one at the end. Thus the order of message receipt is not as simple as the order of computation, and thus there is still merit in moving the one call up.

We could even put in requests to receive neighbour rows as soon as we start the iteration, instead of at the end.

The reason I did not go all-out shuffling my program order to send/receive all messages at the earliest time, is that it makes the output extremely hard to follow, and the code hard to read. Since suitable performance had already been achieved, I decided to keep what simplicity I could.

### Scalability

In stark contrast to the shared-memory parallelism, where resources could only be scaled vertically (improving a single computer - for which a financial/technological limit is quickly met), distributed parallelism allows horizontal scaling (simply adding extra, commodity computers).

Horizontal scaling is limited on the ability of the network to scale with the number of compute nodes. There are topology considerations here: for each node to be connected directly to each other, the amount of links needed multiplies rapidly (n(n-1)/2 for n nodes). The physical topology becomes impractical quickly, but the logical topology is kept simple; no switching or broadcasting is required to contact nodes.

A line topology (where each node is connected to one before and after it, in a double-linked list), could be suitable for this algorithm, because communication is largely between immediate neighbours. They could enjoy a direct connection to just the nodes they need to talk to, and the network is a simple one which scales linearly. However, this is less suitable for detection of problem completion, where all nodes need to share their results with all other nodes via a reduce; the farthest two nodes would need to transmit their data over many hops to reach each other. For reduce operations, a tree topology would work well, as reduce operations map very well to this structure. Trees scale very well (logarithmically), and nodes could still enjoy a direct or near-direct connection to their neighbours.

Horizontal scaling is also limited by the scaling of the algorithm's communication needs. If all nodes need to contact all other nodes, n(n-1)/2 messages need to be sent (excepting cleverless like combining messages, for example in a reduce). A large number of messages can fill the network, slowing communications. And when everyone is waiting on the slowest node to catch up before they can continue, it is worse to have more nodes to wait on. Thus the number of messages being sent, or the number of recipients, can potentially become high enough to dwarf the benefit of adding compute nodes.

Once again, communication is expensive compared to computation. So unless the algorithm and network scale well, there comes a point where the extra computation doesn't help.

### Repeating work

The class of 'constant overheads' that shared-memory parallelism incurred (for example, creation of pthreads at the start of the problem) doesn't map so well to distributed parallelism; any work that has to be done by one process has to be done by all. For example, parameter validation needs to be done on a per-process basis.

Memory allocation has to be done by all processes (which is of course the point), but scaling the problem out does mean we get an equal number of resources to resource demands. There exists a complication in the fact that processes can co-exist on a node, which would mean they share a pool of memory. Within a node, the computation can look very similar to a shared-memory problem, except with the formality of passing messages (very quickly) instead of using globally-shared values.

# Summary of approach

## Description

With all the above reasoning in mind, I will describe the algorithm I came to (though much has already been revealed).

On startup, each process works out which rows of the matrix they own. They allocate memory for that number of rows, plus the rows that border those. They allocate memory for at least one more copy of the matrix to write results into. If caching is utilised (as is the default), memory is allocated for as many matrices as the cacheSize calls for. A matrixArray enables sequential pointing to any of these matrices.

Each process loops through all the values they are responsible for, reading the cardinal neighbours of that value, averaging them and writing the result into the corresponding slot in the 'next' matrix in the matrixArray. The index is circular (that is, it is calculated to the modulus of the number of matrices), so there will always be a 'next' matrix, even if it is one we have used before.

Upon calculating the value of any matrix element, the process compares its new value to the one it started the iteration with. If the absolute difference between these is greater than the precision of the problem, then the territory is considered 'not relaxed' (and no further precision comparisons are done that iteration; this is a 'fast-fail'). It follows that, should all value deltas be within the allowed precision, the territory is considered 'relaxed' at the end of the iteration. We record in a 'progressArray' whether our territory was relaxed on this iteration. The progressArray has slots for as many iterations as we cache.

Once we have operated on all the values we own, we send our top and bottom rows to our immediate neighbours. We then prepare to receive from our immediate neighbours those rows that border us, but aren't owned with us. Since we cannot continue to the next iteration without this information, we block on the receipt of both these rows (one call is simply a blocking receive, the other is happening simultaneously, and we wait for it with an MPI\_Wait). We do not block on our sends, as we can start the next iteration without these finishing. But before any subsequent send, we do block on previous sends' finishing, to allow us to re-use that buffer. It must be noted also that, first and last processes have less neighbours (the formation is a non-circular double-linked list), and thus send and receive less data.

Having finished all our writes, and received our neighbouring rows, we have enough information to start the next iteration. We cycle both our matrix pointers forward one (in modular arithmetic); this gives us a new destination matrix to write to, and points our source matrix to the destination we wrote to last iteration. At minimal cache size (2), the result is equivalent to a pointer swap. With this done, we continue the iteration as previously.

Iterations are pursued until we hit the final slot in our cache (and could store no more iterations of work). At this point, we do a blocking reduce on our progressArray; all processes must reach that same iteration to participate in the reduce operation. The reduce performs a MPI\_MIN on the array, telling us the minimum flag value each process set for a given iteration. 'Relaxed' is the highest, with 'unrelaxed', and 'incomplete' being each respectively lower. Theoretically 'Incomplete' should never be seen at this point in the code, as those iterations are all necessarily finished. If the minimum value for an iteration is 'relaxed', then all processes had relaxed territory on that iteration. We loop through the results to find the first iteration this happened on (if any).

If there exists an iteration where all processes were relaxed, then we conclude problem completion, and remember which matrix that answer was written to, as well as working out which iteration number that corresponds to. If there is no winning iteration, then all processes reset their progress array to 'incomplete', and begin the next iteration (where pointers will cycle round to write to the start of their matrix cache).

Upon identification of the winning iteration, the matrix needs to be printed from top to bottom. The first process prints all the rows he is responsible for (which will include the top row as well as his editable rows), then uses blocking receipt to wait for each subsequent process to send it their own portion of the matrix. These messages overwrite his own memory of the matrix (it is not needed any more). There is necessarily enough room in memory (despite the fact that other processes can have different numbers of rows), because extra rows are allocated from first to last process; rank 0 necessarily has memory for the largest number of rows that can be allocated. On receipt, the process prints each matrix portion until the end is reached. The final process is responsible for the bottom row as well as the rows he writes, so this is sent at that point (though it could instead be generated if message size were a concern).

Admittedly it is not completely necessary to block on each matrix receipt; where caching is enabled to maximum, we have enough matrix memory allocated (one per process) to receive the entire matrix at once. But as this is a transaction that happens just once per program run, its performance is less important. Again I have opted for program simplicity over performance here; if the cache size isn't large enough, using the matrix cache for message receipt would becomes quite complex, as individual re-use would need to be negotiated.

## Main advantages

* Expensive completion detection only occurs once every cacheSize iterations.
* Data dependency is kept low; maximum of two processes need to be contacted each iteration.
* Slowest process does not stop all others doing computation until they have gone many iterations ahead. In fact, if that process speeds up again for some reason, it can avoid ever blocking progress (where otherwise it could be guaranteed).
* Caching iterations allows utilisation of the generous memory resources of distributed computation.
* Provided cacheSize is lowered to allow each portion to fit, arbitrarily large matrices can be tackled!
* Continues computation whilst rows are sent to neighbours.
* Receives rows from neighbours simultaneously, rather than blocking on both.
* Very deterministic; introduces no error, and finds first successful iteration.
* Parallelisation applies even on shared-memory systems; distribution therefore utilises multi-core clusters well.

# Correctness Testing

There are two factors in correctness: does the program compute the correct answer, and does the program compute the answer correctly. These are complicated by determinism; if race conditions exist in the program, then a different answer could be reached on subsequent runs. For example, if processes didn't block on receipt of neighbour rows each iteration, they might only sometimes receive the row in time to use it (and error is introduced when they don't, that has to be smoothed out later).

Additionally, since we are looking to converge on an equilibrium within a required precision, there are many answers that fulfil the program criteria.

There exists also the concern of which iteration to end on, when success is evaluated in batches - if it is difficult to find out which was the *first* successful iteration, the alternative is to return the *latest* successful answer. Different batch sizes could then return different answers, with the 'first successful batch' ending on a different iteration.

Fortunately this particular algorithm was designed to be highly deterministic (it looks for the first successful answer, and aims not to introduce error), and avoids race situations by using appropriately-placed waits.

## Is the answer computed correctly?

Here we have to confirm that all the intentions we have as the programmer are carried out.

### Is the work distributed correctly?

We expect the matrix to be divided at least evenly in terms of integer divison, with the first few processes given remainder rows on top of that.

We split a matrix with a prime number of operable rows (7x7 gives 5 operable rows), among a smaller prime number (3) of processes:

|  |
| --- |
| PROCS = 3  Operable Rows: 5 Min Rows/Proc: 1 Remainder Rows: 2  [RANK0 ] Start Row: 1 End Row: 2  [RANK0 ] (Rel) Writable Start Row: 1 End Row: 2  [RANK0 ] (Rel) Readable Start Row: 0 End Row: 3 |
| [RANK1 ] Start Row: 3 End Row: 4  [RANK1 ] (Rel) Writable Start Row: 1 End Row: 2  [RANK1 ] (Rel) Readable Start Row: 0 End Row: 3 |
| [RANK2 ] Start Row: 5 End Row: 5  [RANK2 ] (Rel) Writable Start Row: 1 End Row: 1  [RANK2 ] (Rel) Readable Start Row: 0 End Row: 2 |

The rows allocated for writing are: 1-2, 3-4, 5-5. So, writable rows 1-5 are distributed, as required.

Dividing the 5 fairly requires a minimum of 1 row per process. True enough, each process is responsible for at least one row.

There are 2 remainder rows to allocate. And two processes receive an extra row, as required.

For all processes, their read rows start one row before and end one row after their write rows, so all values have neighbours to read from.

### Are messages passed correctly each iteration?

We need to receive start and end rows from our neighbours before we can start the next iteration.

Let's confirm that this happens. A 5x5 matrix is used, on two processes.

|  |  |
| --- | --- |
| PROCS = 2  [RANK0 ] Start Row: 1 End Row: 2  [RANK0 ] (Rel) Writable Start Row: 1 End Row: 2  [RANK0 ] (Rel) Readable Start Row: 0 End Row: 3  [RANK0 ] beginning iteration 0.  [RANK0 ] For iteration 0 (0 modx), relaxed: 1  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.500000 0.250000 0.250000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  Sending our end row to next rank:  1.000000 0.250000 0.000000 0.000000 0.000000  Received data will overwrite these last row contents:  1.000000 0.000000 0.000000 0.000000 0.000000  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.500000 0.250000 0.250000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000 | [RANK1 ] Start Row: 3 End Row: 3  [RANK1 ] (Rel) Writable Start Row: 1 End Row: 1  [RANK1 ] (Rel) Readable Start Row: 0 End Row: 2  [RANK1 ] beginning iteration 0.  [RANK1 ] For iteration 0 (0 modx), relaxed: 1  1.000000 0.000000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  Sending our start row to previous rank:  1.000000 0.250000 0.000000 0.000000 0.000000  Received data will overwrite these first row contents:  1.000000 0.000000 0.000000 0.000000 0.000000  Final matrix for this iteration:  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 |

Rank 0 writes to rows 1 and 2. It does not change row 3, as this belongs to Rank 1.

Rank 0 sends its end row to the next process. There is no previous process to send its start row to.

Rank 0 prepares for receipt of a new read-only last row, declaring which values will be overwritten.

Rank 0 now prints the final matrix portion, with the received read-only bottom row from Rank 1.

The top edge of the matrix, which Rank 0 owns, remains unchanged.

In the meantime, we see that:

Rank 1 changes its only write row, row 3.

Rank 1 sends its starting write row (row 3) to Rank 1.

Rank 1 prepares for receipt of a new read-only first row from Rank 0.

Rank 1 now prints its final matrix portion, with the received read-only top row from Rank 0.

The bottom edge of the matrix, which Rank 1 owns, remains unchanged.

All of this is as required.

Investigating behaviour of further ranks (such as those that have two neighbours) followed a similar process.

### Is progress made (by each process) every iteration?

We look at the final matrix portion for a few contiguous iteration (5x5 matrix):

|  |  |
| --- | --- |
| [RANK0 ] beginning iteration 0.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.500000 0.250000 0.250000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  [RANK0 ] beginning iteration 1.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.625000 0.437500 0.312500 0.000000  1.000000 0.437500 0.125000 0.062500 0.000000  1.000000 0.312500 0.062500 0.000000 0.000000  [RANK0 ] beginning iteration 2.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.718750 0.515625 0.375000 0.000000  1.000000 0.515625 0.250000 0.109375 0.000000  1.000000 0.375000 0.109375 0.031250 0.000000 | [RANK1 ] beginning iteration 0.  Final matrix for this iteration:  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  [RANK1 ] beginning iteration 1.  Final matrix for this iteration:  1.000000 0.437500 0.125000 0.062500 0.000000  1.000000 0.312500 0.062500 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  [RANK1 ] beginning iteration 2.  Final matrix for this iteration:  1.000000 0.515625 0.250000 0.109375 0.000000  1.000000 0.375000 0.109375 0.031250 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 |

The most obvious sign that progress is flowing across the grid, is that more and more 0s get filled in each iteration.

Value is also top-left-heavy each time, which is indicative of the fact that value flows from that direction.

### Does the matrix cache cycle correctly?

This can be demonstrated by watching one process, using a small cache size. A 4x4 matrix is shared between two processes.

For iterations 0 and 1, we show just the final matrix achieved in the iteration, but for iterations 2 and 3 we show the matrix achieved before message passing fills in the edges of the final matrix.

|  |  |
| --- | --- |
| **[RANK0 ] beginning iteration 0.**  [RANK0 ] For iteration 0 (0 modx), relaxed: 1  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000  1.000000 0.500000 0.250000 0.000000  *1.000000 0.250000 0.000000 0.000000*  **[RANK0 ] beginning iteration 1.**  [RANK0 ] For iteration 1 (1 modx), relaxed: 1  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000  1.000000 0.625000 0.375000 0.000000  *1.000000 0.375000 0.125000 0.000000*  **[RANK0 ] beginning iteration 2.**  [RANK0 ] For iteration 2 (0 modx), relaxed: 2  1.000000 1.000000 1.000000 1.000000  1.000000 0.687500 0.437500 0.000000  *1.000000 0.250000 0.000000 0.000000*  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000  1.000000 0.687500 0.437500 0.000000  1.000000 0.437500 0.187500 0.000000  **[RANK0 ] beginning iteration 3.**  [RANK0 ] For iteration 3 (1 modx), relaxed: 2  1.000000 1.000000 1.000000 1.000000  1.000000 0.718750 0.468750 0.000000  *1.000000 0.375000 0.125000 0.000000* | ==\  |  |  |  |  |  | Matrix 1  |  ==|==\  | |  | |  | |  | |  | |  ==/ |  |  |  | Matrix 0  |  |  |  |  |  |  |  |  |  =====/ |

Cache size is 2; matrices 0 and 1 exist.

Iteration 0 reads from matrix 0 (and writes to matrix 1, which we print). The next time we print matrix 1, it will still have the original values we wrote in it.

We print matrix 1 again at the start of iteration 2; we have already written all our write rows into it. But on the bottom edge, which will be filled in by message passing, we still see the old values until that happens.

Both the matrices in our cache get reused.

The same technique demonstrates effectively that this holds for higher cache sizes (this is just the minimum example). Matrices in the cache continuously take turns being written to.

### Is relaxation detected correctly?

We can confirm this with a small problem (5x5, 0.1 precision, 2 processes), watching the first process.

|  |
| --- |
| [RANK0 ] beginning iteration 1.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.625000 0.437500 0.312500 0.000000  1.000000 0.437500 0.125000 0.062500 0.000000  1.000000 0.312500 0.062500 0.000000 0.000000  [RANK0 ] beginning iteration 2.  [RANK0 ] For iteration 2 (0 modx), relaxed: 1  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.718750 0.515625 0.375000 0.000000  1.000000 0.515625 0.250000 0.109375 0.000000  1.000000 0.375000 0.109375 0.031250 0.000000  [RANK0 ] beginning iteration 3.  [RANK0 ] For iteration 3 (1 modx), relaxed: 2  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.757812 0.585938 0.406250 0.000000  1.000000 0.585938 0.312500 0.164062 0.000000  1.000000 0.406250 0.164062 0.054688 0.000000 |

For reference, the 'relaxed' enumerations are as follows:

* 2 = Finished, Relaxed
* 1 = Finished, Non-relaxed
* 0 = Unfinished

Though 'unfinished' should never be seen at print-time, which prints only after finishing an iteration.

From iteration 1 to 2, 'relaxed' is enumerated as 'finished, not relaxed'.

Sure enough, there exist deltas between those iterations that exceed 0.1. On row 2, we see the value 0.125 change to 0.25, a difference of 0.125.

From iteration 2 to 3, no value changes by as much as 0.1. The element which prevented relaxation last iteration, changes this time from 0.25 to 0.3125, with a difference of 0.0625.

This looks to be generating the correct result.

### Is the correct end iteration recognised (by all processes)?

We demonstrate a problem where the oldest iteration in the cache is the winner (5x5, 0.07), and is recognised as such:

|  |  |
| --- | --- |
| [RANK0 ] For iteration 2 (0 modx), relaxed: 1  [RANK0 ] For iteration 3 (1 modx), relaxed: 1  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 1  min is: 1  Checking iteration (mod): 1  my relax on that iteration was: 1  min is: 1  [RANK0 ] For iteration 4 (0 modx), relaxed: 2  [RANK0 ] For iteration 5 (1 modx), relaxed: 2  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 2  min is: 2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK0] Wrapping up..  First winning iteration was: 4  This is at matrix cache: 1 | [RANK1 ] For iteration 2 (0 modx), relaxed: 2  [RANK1 ] For iteration 3 (1 modx), relaxed: 2  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 2  min is: 1  Checking iteration (mod): 1  my relax on that iteration was: 2  min is: 1  [RANK1 ] For iteration 4 (0 modx), relaxed: 2  [RANK1 ] For iteration 5 (1 modx), relaxed: 2  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 2  min is: 2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK1] Wrapping up..  First winning iteration was: 4  This is matrix: 1 |

After iteration 3, the processes have filled their cache and share results ('Gather!' marks the reduce action). They check the first array progress slot (iteration 2 = iteration 0 mod cacheSize), then the second (iteration 3 = iteration 1 mod cacheSize), and find that minimum value was '1' for both of these; total relaxation was not achieved on either of these iterations.

Both processes then computer two further iterations, and do the reduce again.

This time they discover that they were both relaxed on iteration 4 (iteration 0 mod cacheSize).

Though the source matrix for that iteration was 0, the destination matrix is the next matrix in cache, so the matrix to print is recognised correctly as 1.

The algorithm also copes with identifying the winner when it is the newest iteration in cache (5x5, 0.1):

|  |  |
| --- | --- |
| [RANK0 ] For iteration 2 (0 modx), relaxed: 1  [RANK0 ] For iteration 3 (1 modx), relaxed: 2  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 1  min is: 1  Checking iteration (mod): 1  my relax on that iteration was: 2  min is: 2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK0] Wrapping up..  First winning iteration was: 3  This is at matrix cache: 0 | [RANK1 ] For iteration 2 (0 modx), relaxed: 2  [RANK1 ] For iteration 3 (1 modx), relaxed: 2  Gather!  Checking iteration (mod): 0  my relax on that iteration was: 2  min is: 1  Checking iteration (mod): 1  my relax on that iteration was: 2  min is: 2  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK1] Wrapping up..  First winning iteration was: 3  This is matrix: 0 |

The notable difference this time is that the first minimum is seen to be low, but it continues to check the next minimum and finds success.

Iteration 3 is correctly declared the first winning iteration, and the position it was written into in cache is correctly identified as 0.

### Do answers merge correctly?

To demonstrate a particularly difficult merge, I present a (7x7) matrix with a prime number of operable rows (5) being allocated to a lower prime number (3) of processes. Multiple processes need to co-operate, in order, and they don't all have the same number of rows.

|  |
| --- |
| [RANK0 ] For iteration 3 (0 modx), relaxed: 2  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.765625 0.609375 0.527344 0.488281 0.382812 0.000000  1.000000 0.609375 0.335938 0.218750 0.171875 0.125000 0.000000  1.000000 0.527344 0.218750 0.078125 0.042969 0.027344 0.000000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK0] Wrapping up..  First winning iteration was: 3  This is at matrix cache: 1  1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.765625 0.609375 0.527344 0.488281 0.382812 0.000000  1.000000 0.609375 0.335938 0.218750 0.171875 0.125000 0.000000  Expecting buffer of size: 14  1.000000 0.527344 0.218750 0.078125 0.042969 0.027344 0.000000  1.000000 0.488281 0.171875 0.042969 0.007812 0.003906 0.000000  Expecting buffer of size: 14  1.000000 0.382812 0.125000 0.027344 0.003906 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000  Reached MPI Finalize.  Reached end of main. |
| [RANK1 ] For iteration 3 (0 modx), relaxed: 2  Final matrix for this iteration:  1.000000 0.609375 0.335938 0.218750 0.171875 0.125000 0.000000  1.000000 0.527344 0.218750 0.078125 0.042969 0.027344 0.000000  1.000000 0.488281 0.171875 0.042969 0.007812 0.003906 0.000000  1.000000 0.382812 0.125000 0.027344 0.003906 0.000000 0.000000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK1] Wrapping up..  First winning iteration was: 3  This is matrix: 1  Sending buffer of size: 14  Reached MPI Finalize.  Reached end of main. |
| [RANK2 ] For iteration 3 (0 modx), relaxed: 2  Final matrix for this iteration:  1.000000 0.488281 0.171875 0.042969 0.007812 0.003906 0.000000  1.000000 0.382812 0.125000 0.027344 0.003906 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK2] Wrapping up..  First winning iteration was: 3  This is matrix: 1  Sending buffer of size: 14  Reached MPI Finalize.  Reached end of main. |

It can be seen that rank 0 correctly prints the portions of the matrix owned by each process.

This means also that it overcame the complication of requesting only those rows unique to each process.

The order is correct, and the correct number of rows are printed; a 7x7 matrix is created.

The read-only top and bottom rows are also preserved, as well as the read-only columns on left and right.

### Is the answer the same on subsequent runs?

Admittedly if race conditions exist, just showing a sample of successful runs would not imply a 100% success rate. I can at least demonstrate that the variance isn't wild, by showing several equal runs.

5x5 matrix, 0.0001 precision was used.

|  |
| --- |
| First winning iteration was: 23  This is at matrix cache: 0  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.857051 0.714164 0.499908 0.000000  1.000000 0.714164 0.499817 0.285592 0.000000  1.000000 0.499908 0.285592 0.142766 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 |
| First winning iteration was: 23  This is at matrix cache: 0  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.857051 0.714164 0.499908 0.000000  1.000000 0.714164 0.499817 0.285592 0.000000  1.000000 0.499908 0.285592 0.142766 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 |
| First winning iteration was: 23  This is at matrix cache: 0  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.857051 0.714164 0.499908 0.000000  1.000000 0.714164 0.499817 0.285592 0.000000  1.000000 0.499908 0.285592 0.142766 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000 |

All three runs gave the exact same answer. So we have some small amount of confidence.

The best way to rule out race conditions is to reason through the logic of the code.

Some important orderings are enforced:

* Cannot end iteration without starting send of border rows to neighbours.
* Cannot end iteration until receipt of read-only rows from neighbours.
* Cannot evaluate win criteria until all processes have reached the same iteration.
* Final matrix merge uses a sequence of blocking receives, listening to ascending process ranks.

Thus we cannot send a row update to the wrong iteration; the neighbour can only receive into the latest iteration. But that is necessarily the right iteration, since we can only be ahead of them once they reciprocate our row update.

By enforcing that a process can only advance 'cacheSize' iterations before they wait for others to catch up, we ensure that all processes are on the same iteration when they compare results, and by extension that everyone's result history refers to the same set of iterations.

As for merging results at the end, the rank 0 process that orchestrates this, specifies in order which processes it wants to receive matrix portions from. This is a blocking receive, so only one process is being attended to at a time. Thus matrix portions are necessarily read in the right order, even if they are sent in a chaotic order.

## Is the answer correct?

The most complete way to investigate this would be to compare the result achieved against an alternative, trusted method; since relaxation of matrices is used to solve some mathematics problems, this algorithm’s output could be compared against a different numeric method (or non-numeric method) and see if their results can be used to solve the same problem. However this is not a maths assignment, so I will not worry about how to do this.

Alternatively, understanding of the problem can be applied, to see if the predicted behaviour manifests as expected. I described my interpretation of the problem in the first section of this report (‘Problem’).

### Does result converge in the predicted manner?

Using two parallel xterms, we print each process's portion of the matrix on each iteration, and check that the behaviour converges in the expected pattern, and also that the final result is a plausible equilibrium in light of predictions. A 5x5 matrix is used, to precision 0.001, with two processes.

|  |  |
| --- | --- |
| [RANK0 ] beginning iteration 0.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.500000 0.250000 0.250000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  [RANK0 ] beginning iteration 1.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.625000 0.437500 0.312500 0.000000  1.000000 0.437500 0.125000 0.062500 0.000000  1.000000 0.312500 0.062500 0.000000 0.000000  [RANK0 ] beginning iteration 2.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.718750 0.515625 0.375000 0.000000  1.000000 0.515625 0.250000 0.109375 0.000000  1.000000 0.375000 0.109375 0.031250 0.000000  ...  [RANK0 ] beginning iteration 14.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.855190 0.711356 0.498047 0.000000  1.000000 0.711356 0.496094 0.282785 0.000000  1.000000 0.498047 0.282785 0.140904 0.000000  [RANK0 ] beginning iteration 15.  Final matrix for this iteration:  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.855678 0.712333 0.498535 0.000000  1.000000 0.712333 0.497070 0.283761 0.000000  1.000000 0.498535 0.283761 0.141392 0.000000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK0] Wrapping up..  First winning iteration was: 15  1.000000 1.000000 1.000000 1.000000 1.000000  1.000000 0.855678 0.712333 0.498535 0.000000  1.000000 0.712333 0.497070 0.283761 0.000000  1.000000 0.498535 0.283761 0.141392 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  Reached MPI Finalize.  Reached end of main. | [RANK1 ] beginning iteration 0.  Final matrix for this iteration:  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.250000 0.000000 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  [RANK1 ] beginning iteration 1.  Final matrix for this iteration:  1.000000 0.437500 0.125000 0.062500 0.000000  1.000000 0.312500 0.062500 0.000000 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  [RANK1 ] beginning iteration 2.  Final matrix for this iteration:  1.000000 0.515625 0.250000 0.109375 0.000000  1.000000 0.375000 0.109375 0.031250 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  ...  [RANK1 ] beginning iteration 14.  Final matrix for this iteration:  1.000000 0.711356 0.496094 0.282785 0.000000  1.000000 0.498047 0.282785 0.140904 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  [RANK1 ] beginning iteration 15.  Final matrix for this iteration:  1.000000 0.712333 0.497070 0.283761 0.000000  1.000000 0.498535 0.283761 0.141392 0.000000  1.000000 0.000000 0.000000 0.000000 0.000000  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [RANK1] Wrapping up..  First winning iteration was: 15 |

Progress flows across the grid, filling in more and more 0s each iteration, as required.

Values are highest in the top-left, indicative of the fact that value flows from that direction.

Value change by smaller and smaller amounts each iteration (0.5 -> 0.625, vs 0.85519 -> 0.855678).

The outermost edges remain unchanged.

It is reasonable to conclude that this is the right ending distribution for an equilibrium, and that the relaxation progress shows convergence.

# Speedup

## Theoretical speedup

### Predictors

Two popular laws exist for measuring the speed-up of parallel programs: Amdahl’s[[2]](#footnote-2) and Gustavson’s[[3]](#footnote-3).



Amdahl’s Law: S(N) is the amount of speedup possible on N processors, where P is the parallelizable proportion of the program (and hence 1-P is the sequential portion). If a program were wholly parallelisable (P = 1), then the speedup limit is N, the number of processors recruited.

Amdahl’s Law seeks to find the amount of speed-up a problem of fixed-size can attain in parallel. The main take-away is that, for a problem that is not completely parallelisable, each processor added makes a smaller contribution than the last.



Gustafson’s Law: S(P) is the ‘scaled speed-up’ for P processors, with *α* being the non-parallelizable fraction of any parallel process.

Gustafson’s Law finds ‘scaled speed-up’, the ratio between single-process run-time and per-process parallel run-time. In other words, Gustafson’s law recognises that since the overall workload of a machine increases with problem size, distributing the work in parallel allows arbitrarily larger problems to be solved in equivalent time (rather than solving a fixed problem size in a shorter time).

As with Amdahl's Law, if a program were wholly parallelisable (*α* = 0), then the 'scaled speed-up' limit equals P, the number of processors recruited. But for a program that is not completely parallelisable, the speedup is P minus the time each processor loses to sequential work. This effectively says, "each processor recruited can provide the same amount of speedup if the problem scales too", whereas Amdahl's Law said "each processor recruited will provide less speedup than the last, assuming the problem size stays the same".

Both of these laws are simplifications that do not take into account the overhead incurred in

parallel problem solution - for example, processes waiting for each other to catch up, or waiting on communication. These factors worsen as the amount of parallelization increases (perhaps at a rate worse than linear - for example, if all processes need to communicate with all processes), and are significant enough in reality that it is often better not to add more processors (or parallelise at all), since overhead is so significant. Ideally a problem should be chosen that is highly suited to parallelisation.

### Application to our problem

A large portion of our problem is parallelisable; the majority of program computation is on averaging the matrix values. This work is highly data-parallel; the same set of steps can be applied to multiple chunks of data independently and simultaneously, and thus can be distributed well. There is relatively little data dependency in our algorithm, so each process can do a lot of computation individually before needing to consult another.

There is, however, a lot of work which has to be done P times with the number of processors used, like working out which rows are allocated to each process, or allocating memory for each process. Work that can be done in parallel, but has to be done P times, is equivalent to sequential work, as real time neither benefits nor suffer from its parallelisation. Unlike with shared-memory, the concept of 'work that only has to be done once' is a bit harder to map, since work like setting up the initial program state cannot be shared to other processes; it has to be done on a per-process basis.

As we can consider ourselves to have at least some amount of 'sequential' work (or work that has to be done P times by P processes), we should not expect anything as generous as linear speedup, especially when considerable communication costs are taken into account.

It is possible to achieve ‘super-linear‘ speed-up with respect to processors, in cases of problems where significant overhead is incurred in sequential processing that is not present in parallel processing. For example, if the sequential run did not have enough memory to work on the entire problem, then it would be slowed down by swapping that the parallel run would not suffer from.

In distributed-memory parallelism, it is easy to imagine a situation where recruiting a large number of computers would yield a large amount of memory to work with. A problem which does not fit in the memory of one computer could be either rejected (with the OS refusing to allocate the memory), or attempted using virtual memory from the hard drive, incurring a huge performance penalty. Thus, using two computers that together have enough memory for the problem, versus one which does not, allows the whole problem to be computed in RAM, completing the problem orders of magnitude faster than could be done on the single computer.

Memory cache is another resource factor that can lead to superlinearity; recruitment of more processors means recruitment of more fast, precious cache, allowing more portions of the problem to be computed quickly than if one single-core processor attempted it.

## Practical speedup

We can see how close we get to the speed-up predicted by these models when scaling up our algorithm. Time is measured using the UNIX ‘time’ function, which provides an approximation of ‘real’ world time elapsed, as well as ‘user’ time (the amount of CPU time cumulatively incurred by all processors). This ‘user’ time could be useful later for measuring efficiency, but for now we used only the ‘real’ time, to compare how long the run took.

'sys' time is measured also, and provides a view on how much time was spent making system calls, particularly memory allocation. Since our algorithm reserves a potentially large matrix cache, this time could be significant.

All values used are the mean average of 3 runs, to ensure validity.

Times are normalised as fractions of the metric used for the first run, so..

If the problem takes 20 seconds with 1 node,

and 15 seconds with 2 nodes,

then we plot ‘20÷15 = 1.33 speedup’.

In cases where multiple processor cores were recruited, then the distinction between per-node speedup and per-process speedup will be important (as 1 node vs 4 could mean 8 cores vs 32, meaning the measured speedup is only relative to the number of nodes).

These runs were done on 4 Aquila nodes with either 1 or 8 processor cores (across two processors).

The program was compiled with 'verbose' and 'printfinalresult' undefined, since this logging significantly influenced recorded times.

The program was compiled in ‘float’ mode rather than ‘double’, out of arbitrary choice.

There are a lot of factors which influence which data type is faster on a processor (such as whether it is natively implemented, or just emulated), but computation time would scale for all processes equally, meaning the speedup would not be affected by that alone. Message sizes would vary, and this could make a difference in communication wait times in the program. But an informative trend in speed-up will be seen either way.

### Speedup on small problem, multi-core, matrix cache enabled

0.005 precision

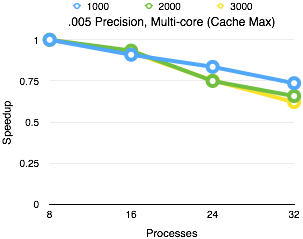
[1000, 2000, 3000] magnitude

Iterations Cached = Maximum (number of processes)

8 Processor cores per node

1-4 nodes used

'real' UNIX time



Each 8 processes recruited represents an extra node being used.

Note that the single-node run is already parallelised across its 8 cores, without having to send messages to other nodes. Adding more nodes will introduce communication costs, so they would need to do a lot of computation to make up for this.

For a small problem, we quickly reach the limit of how far we can speed up the parallel portion of the problem. Beyond this, adding processes just adds communication work, and waits on other processes (due to data dependency).

Thus is it unsurprising to see that speedup goes down with the number of nodes recruited.

It is interesting to see that, counter-intuitively, the benefit of parallelism gets even worse as we increase the size of the matrix.

There are two possible reasons for this, and both are effects of the cache:

* The time saved by caching may be smaller than the time spent allocating the memory.
  + For larger matrices, more time is spent allocating memory for the matrix cache.
* The cache checks for successful iterations in batches. Larger batch sizes can overshoot the answer by more iterations. More iterations are computed than necessary.
  + For low-precision matrices, few iterations are necessary. A large batch size can overshoot the answer very early. More time is spent on wasted computation than is saved on communication.
  + The penalty for overshooting the answer gets worse with the size of the matrix, as each wasted iteration takes more time to compute.

**Conclusions:**

Do not parallelise small problems.

Only use caching when there are many iterations (high-precision problems).

### Speedup on slightly larger problem, single-core, matrix cache enabled

0.001 precision

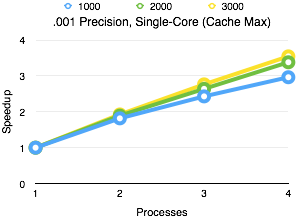
[1000, 2000, 3000] magnitude

Iterations Cached = Maximum (number of processes)

1 Processor cores per node

1-4 nodes used

'real' UNIX time



Here we see more speedup with every process added. Perhaps because the problem size has been increased slightly (so there is enough work to parallelise).

Though we might expect a drop in speed due to only using one core per node (we are introducing communication without so much computation to show for it), this does not seem to be a problem.

The results are still strange, as the problem is only slightly larger (double precision) than the previous one. But perhaps this can be explained by the fact that, with the number of processes lower, the size of the cache (which caused the slowdown previously) is smaller also.

Additionally, the previous problem had processes co-existing on one node battling for an already-large amount of memory. This problem has just one process per node, so the RAM is uncontested; potentially a much larger cache could be used than in the previous problem, simply because it is not being shared. Though without more nodes, the cache size could not be raised to any benefit.

Another consideration of the reduced cache size is that the batch size is reduced; we are less likely to overshoot the answer. For small problems, this is an important consideration.

We see Gustafson's Law in effect here, as the larger problems achieve greater speedup.

Within each fixed problem size, we see also the predictions of Amdahl's Law; recruiting more processes increases speedup (and we could generously say that the smallest problem starts to show less and less benefit from the parallelism with each process added).

**Conclusions:**

A reduced cache size helps for small problems.

More processes solve the problem faster, although they might not do that forever.

Larger problems benefit more from parallelism.

### Speedup on medium problem, matrix cache enabled

0.001 precision

[1000, 2000, 3000, 5000] magnitude

Iterations Cached = Maximum (number of processes)

8 Processor cores per node

1-4 nodes used

'real' UNIX time

### 

With a medium problem size, we see that in general, increasing the size of problem increases the speedup, albeit only slightly.

Anomalously, the largest problem really suffers from being pushed to 24 processes. This anomaly, like previous ones, can be explained through the matrix cache:

Adding more processes increases the maximum number of iterations that a process can advance ahead of another, and by extension, the maximum number of iterations that can be cached.

With many large matrices allocated, not all can fit in RAM. Matrices not in use (or worse - parts of the current matrix, not yet in use) will be swapped out to disk.

Especially considering that there are 8 processes fighting for the RAM of each node, this swapping has a huge impact on speed.

If there is not enough RAM to keep all matrices in RAM, then swapping necessarily occurs during every batch of iterations. For multiple processes on one computer, the boundary tightens to "if there is not enough RAM *per process*".

This explains why we only see this huge speed drop on the largest matrix. Although it is also possible that this is happening to a lesser extent on the other matrices, holding back their potential speedup.

In all cases, the increase from 8 -> 16 processes yielded speedup. The problem was thus large enough this time to benefit from parallelisation.

It is only the smallest problem that benefits from increasing to 24 processes. Likely, on the larger problems, disk swapping is encountered at this point. This could also explain the further slowdown that is met by all at 32 processes.

**Conclusions:**

Matrix cache should be kept small, to avoid swapping.

### Speedup on medium problem, matrix cache disabled

0.001 precision

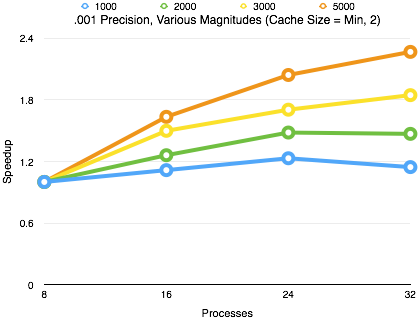
[1000, 2000, 3000, 5000] magnitude

Iterations Cached = Minimum (2)

8 Processor cores per node

1-4 nodes used

'real' UNIX time



Quickly confirming that the slowdown in the previous graph was indeed caused by the effects of the cache.

With caching disabled, the results fit a much more classical Gustafson's Law (the larger problems benefit more from parallelism).

The smaller problems start to lose speedup once too many processes are used; costs of communication or data dependency start to outweigh the benefit of dividing the computation. Excepting this slowdown which Amdahl's Law doesn't model, all lines fit its predictions of "each processor core added gives speedup (albeit less and less) for a fixed problem size".

**Conclusions**

Results do fit Gustafson and Amdahl's predictions of scalability.

The matrix cache was indeed the culprit for slowdown in the previous graph.

### Speedup on larger problem with matrix cache enabled, and variance of algorithm

0.0001 precision

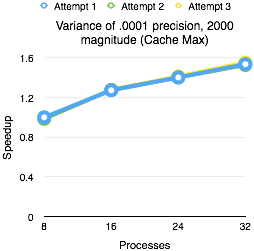
2000 magnitude

Iterations Cached = Maximum (number of processes)

8 Processor cores per node

1-4 nodes used

'real' UNIX time



This problem is made large by high precision, whilst keeping the matrix size average. This smaller matrix reduces the size of the iteration cache in RAM.

Each node added improves the speed, with the maximum speedup achieved being 1.54x faster than the single node.

Amdahl's law suggests a theoretical maximum speedup of 4x for 4 nodes on this fixed problem size (assuming negligible sequential work), but of course it does not model the waits for communication and dependent data.

Without measurement from more nodes, we can't see whether this trend in speedup is tapering off, or continuing arbitrarily. But we do know that unless we restrict the cache size, memory demands will increase as we add processes, until slowdown occurs.

As for the repeated attempts, we see that the behaviour of the algorithm is stable, even though the cache allows many different patterns of program execution to happen; on one run, the first process can find its neighbour has completed an iteration, and can use that information to pursue the next iteration. Whereas on another run, that first process may have to wait on the same neighbour.

**Conclusions:**

Scaling up the precision of the problem is a way to increase work without increasing memory demands.

### Speedup on a larger problem with caching turned off

0.0001 precision

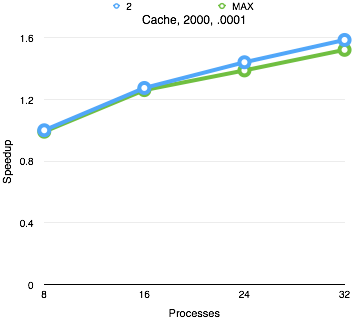
2000 magnitude

Iterations Cached = [Minimum (2), Maximum (number of processes)]

8 Processor cores per node

1-4 nodes used

'real' UNIX time



Since a lot has been said about how the iteration cache (intended to provide speedup) has been the cause of most of our slowdown, an investigation is merited regarding whether the cache is useful for anything.

The reason why the cache is meant to improve speed, is that it enables success criteria to be checked in batches (reducing communication), by recording historical work. Admittedly if we weren't interested in the determinism of finding the 'first' answer, then we could still check for success every n iterations, without this large memory penalty.

Overshooting the answer is a big problem; if the problem takes 33 iterations, and we work in batches of 32, then we would compute 64 iterations before we conclude success. High-precision problems minimize this problem by making the number of iterations high enough that more time is saved by the batching than lost in an overshoot.

This graph compares a high-precision, medium-sized matrix, with caching set to minimum and turned to maximum.

It appears to show that, even in an ideal situation, the speeds are at best equal between the two strategies. Past 16 processes, the cached solution becomes increasingly slower (presumably disk swapping begins here).

It becomes hard to believe that there is any benefit to caching. But the next graph will reveal that there is more than meets the eye:

### Speedup on a larger problem with caching turned off, adjusted for memory allocation time

0.0001 precision

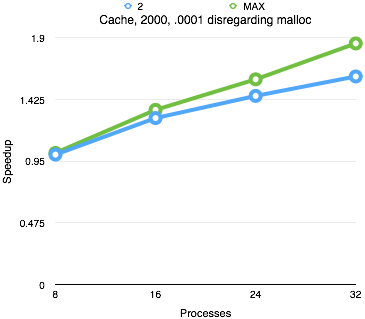
2000 magnitude

Iterations Cached = [Minimum (2), Maximum (number of processes)]

8 Processor cores per node

1-4 nodes used

'real' UNIX time - 'sys' UNIX time



These are the same figures as before, but with 'sys' UNIX time subtracted from our measurements.

This is the time spent by our program making system calls, and in particular memory allocation time.

Of course a large (and increasing) proportion of time is spent on memory allocation by the cache-heavy solution. This takes a significant amount of time.

By disregarding this time penalty, we can see the real speed difference in problem solving.

Finally some support for caching! Maximum cache size provides a significant speed boost over minimum cache size. At the peak, the 'MAX' cache achieves 1.857, where 'MIN (2)' provides only 1.602. This makes the cache-heavy solution 16% faster.

Is it fair to disregard the memory allocation time (as it still happens)? Certainly we have learnt that there is merit in the optimization. If the program were in constant use, it would be possible to allocate the memory once, and solve multiple problems. But for solving just one problem, this time penalty is harmful.

**Conclusions:**

Caching does provide a small benefit.

Memory allocation time dwarfs benefit of caching.

Re-using memory for subsequent computation could claw back this lost time.

### Larger real-time cache comparison

|  |  |
| --- | --- |
|  | 0.0001 precision  2000 magnitude  Iterations Cached = [Minimum (2), Maximum (number of processes)]  8 Processor cores per node  1-4 nodes used  'real' UNIX time  There is not much new information in this complicated graph.  It is an extension of a previous experiment, 'speedup on a larger problem with caching turned off', to measure more problem sizes.  In all cases, it is faster (in real time) to use the minimum cache than the maximum cache.  The outlier result is the 'maximum cache' being applied to a large matrix, and running out of memory as a result.  Thus the trend we have already seen in speedup with the matrix cache, exists for various problem sizes. |

### Full scale test

0.0001 precision

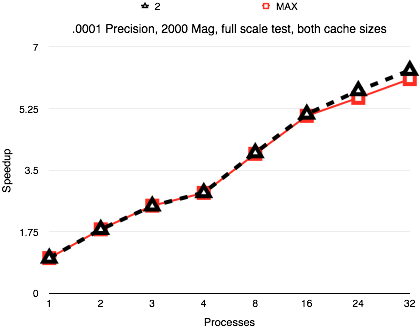
2000 magnitude

Iterations Cached = [Minimum (2), Maximum (number of processes)]

[1, 8] Processor cores per node

1-4 nodes used

'real' UNIX time



Since all previous graphs compared only per-node speedup, I provide a graph of per-core speedup.

This allows comparison on 32 processes with 1 process, rather than with 8.

The 1-4 processes were performed as 1 core per node. The 8-32 processes used 8 cores per node.

The problem is high-precision, medium matrix size.

The highest speedup achieved over 1 process, is 6.3x, by MIN cache. This occurs at 32 processes. MAX cache achieves 6.1x speedup here.

The results between MIN (2) cache and MAX cache are similar, up until 24 processes, where memory demands become too high.

Again, if memory allocation time were disregarded, perhaps MAX would have faster speedup than all of MIN's values.

In accordance with the predictions of Amdahls' Law, each processor added makes less and less difference to the speedup; either this is because the parallel portion of the problem is starting to be completed in negligible time (as the model predicts), or more likely because our communication and wait costs do not scale linearly.

Not graphed, the speed of a 4-core node was actually *exactly the same* (within 1sec) as four 1-core nodes. Similarly, two 4-core nodes went the same speed as one 8-core node. So perhaps communication costs are not a problem in this implementation!

**Conclusions:**

Communication time is not a bottleneck for us!

Algorithm scales well, but not linearly with number of cores.

Good speed-up is achieved (with and without distributed computation

### Arbitrary scalability

30000 double problem: fit in memory (cache size 2)

|  |  |
| --- | --- |
| **Nodes** | **Fits in memory** |
| **1** | No |
| **2** | Yes |
| **3** | Yes |
| **4** | Yes |

So long as the cache size is restricted, our work division should theoretically be able to fit any size problem across the memory of multiple nodes (well, until we hit other limits like the size of integer used to indirect into the matrix, or allocate the memory).

A 30000x30000 matrix of (8-byte) doubles would take 6.7GB of memory. Minimally we need at least one more matrix in our cache, as we need a source and destination to swap between.

As each Aquila node has 8GB of RAM, allocating two 6.7GB matrices isn't going to fit. Perhaps a pagefile would allow some of that to be stored on disk, but even the pagefile would need to be quite big.

Distributing the matrix across multiple nodes allows us to tackle large problems like these. When 1 node attempted this problem, malloc() refused to allocate the amount of memory. But two nodes can ask for just half the amount. In this way, larger problems were able to be fit across distributed memory.

The advantage of this is that we can utilise the cheap horizontal scalability of cluster computing. It also allows us to solve problems that we simply couldn't have done before. Where speedup is concerned, this allows us to take advantage of Gustafson's Law, and pursue the larger speedup benefits we see on larger problems.

# Efficiency

Efficiency is measured as speedup per processor:

En = Sn÷n = T1÷(nTn)

That is, speedup Sn experienced using n processors, divided by that many processors. Also readable as ‘time to perform on one processor, compared to time for n processors to each incur Tn time’.

Ordinarily this value will be a bottom-heavy positive fraction, but in cases of super-linear speedup (mentioned previously in the ‘Speedup’ section), efficiency can exceed 100% (that is, time to perform on one processor is greater than time taken for n processors to incur Tn time).

Efficiency is relevant, because it highlights how much additional useful work is output by recruiting more processors. In cases of low efficiency, the costs of providing the extra processors might not justify the amount of extra work output by them. This is a serious concern for datacenters, that pay for the electricity!

We will work out efficiency by dividing our existing ‘speedup’ values by the number of cores used (equally, processes used).

### Full scale test

0.0001 precision

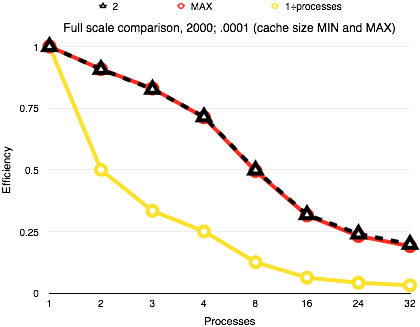
2000 magnitude

Iterations Cached = [Minimum (2), Maximum (number of processes)]

[1, 8] Processor cores per node

1-4 nodes used

'real' UNIX time



This is an efficiency graph based on the speedup results of the previous full scale test, which compared 1-core nodes with 8-core nodes.

There is little difference between cache being maximized or minimized (the speedup graph of these same values showed that the speed was about the same anyway). Hence these lines perfectly overlap.

In accordance with the predictions of Amdahl's Law, each extra core recruited makes less and less of a contribution to useful work output (as evidenced by the efficiency going down, instead of staying the same).

The lowest efficiency is 20%, at 32 processes. This could be worse, really! As long our efficiency is greater than 1÷processes, then we saved time by adding that process. We are still fairly safe from that limit.

Our efficiency draws closer to the 1÷processes line each time we add a process, so we are getting worse and worse efficiency as we scale the problem. But this is thankfully not happening very quickly, and we can always consider bigger problems if we want to use all our compute power.

The possible reason for losing more efficiency as we increase the number of processes, is that the number of dependent processes increases (so we have to wait for more to be ready when we exchange results, and also have to talk to more people when we reduce said results), and this slows down all existing processes.

**Conclusions:**

Good efficiency is achieved

Efficiency scales well with problem size

Efficiency worsens with the number of processes used, but slowly

### Small problem

0.005 precision

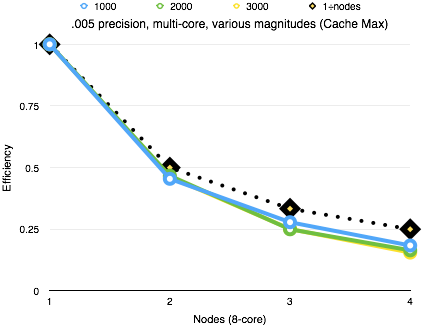
[1000, 2000, 3000] magnitude

Iterations Cached = Maximum (number of processes)

8 Processor cores per node

1-4 nodes used

'real' UNIX time



Here we present the graph from earlier which achieved negative speedup. It was a small problem, which distributed the work across far too many processes (32), with a cache size far too large for the problem.

Sure enough, we see that this is inefficient; all nodes added give efficiency below the crucial 1÷nodes line, meaning that adding them actually subtracted from the amount of useful work output.

Amdahl's Law doesn't model this phenomenon, as it assumes that processes can only contribute progress; waits are not modeled, for example.

**Conclusions:**

Small problems are inefficient or even harmful to parallelise!

### Various problem sizes

0.001 precision

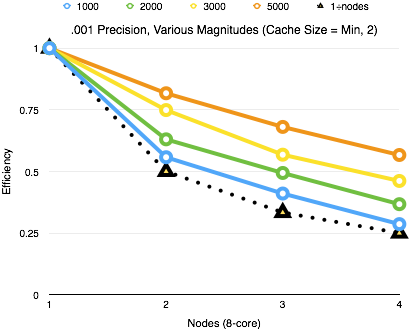
[1000, 2000, 3000, 5000] magnitude

Iterations Cached = Minimum (2)

8 Processor cores per node

1-4 nodes used

'real' UNIX time



Here we see Gustafson's Law in full effect; the smallest problem is at all times almost touching the 1÷nodes lines, so it is just barely providing useful work output for the investment made.

The larger problem maintains a much higher efficiency, still as high as 57% by the end.

**Conclusions:**

Bigger problems parallelise more efficiently, as per Gustafson's Law.

# Conclusion

Despite this’s being a highly parallelisable problem, it is still challenging to squeeze the full potential out of the processor cores available. When considering the costs of parallelising a problem, one must consider not just the cost of answering a problem slowly, but also the costs of answering it inefficiently, and finally the costs of writing parallel software in the first place!

Using a distributed architecture allowed us to fit to larger problem sizes in memory than shared-memory systems could cope with (although demonstrating the performance would be hard, as timing such large computations might hit WALLTIME on Aquila). Crucially, we have demonstrated that this algorithm could scale to almost arbitrarily larger problems.

The matrix cache felt promising on paper, but delivers only a small speedup (when memory allocation can be ignored), and at all other times slows down the computation. Since its added complexity makes the software harder to maintain and debug, and makes the correctness testing process harder, a cacheless version would probably be better in a real engineering context.

Though due care was taken to think through the logic, race conditions could perhaps exist. No problems were encountered in practice, but the '101th run' that ruiins everything could easily be the next one.

This is not the fastest the problem can be solved; a not-insignificant speed sacrifice was made for the sake of determinism. But it appears to be a pretty good way to solve deterministically.

-Alex Birch, akb29

1. *Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice* - Matthias Müller-Hannemann, Stefan Schirra [↑](#footnote-ref-1)
2. Amdahl, Gene (1967). *"Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities"*. AFIPS Conference Proceedings (30): 483–485. [↑](#footnote-ref-2)
3. *Reevaluating Amdahl's Law*, John L. Gustafson, Communications of the ACM 31(5), 1988. pp. 532-533. [↑](#footnote-ref-3)