Parallel Assignment 2: Distributed Computation

Alex Birch – akb29, 19/12/13

Disclaimer: I refer to my own Parallel Assignment 1 for some content that is still applicable.

# 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

## Initial division of problem

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\_. 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.

# Parallelism

## 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 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.

## Problem-specific concerns

BTW! With 32 processes, we can overshoot the answer by 32 iterations. For our standard 72-iteration job, that would constitute almost 50% of the necessary computation…