Heat equation, redux

Dr. Christopher Marcotte — *Durham University*

Previously, we have looked at parallelising the solution of steady states of the heat equation using OpenMP in one- and two-dimensional settings. Now we will consider how to approach the problem using MPI. To make things less repetitive, we'll only consider a two-dimensional setting, but you could do this in 1D as well.

> ***Hint***
>
> This exercise requires very little from you to implement, so it's important that you read through this document and ***understand*** what is being told to you, before you look at the code. After reading through, look through `serial.c`, then `jacobi.c`, and then `jacobi_transpose.c` and understand what each is doing, what the differences are, and how you transform one into the next.

Heat equation

As before, we will have a two dimensional plate, with the sides held at a fixed temperature, while the interior of the plate is subject to,

$$\partial_t u = \nabla^2 u,$$

with an initial condition (that we don't care too much about) $u(t = 0, x, y) = u_0(x, y)$.

We are interested, as before, in the steady-state solution: $u : \partial_t u = 0$, and return to Jacobi iteration to compute it, now with multiple processes and disjoint memory spaces in MPI.

Discretisation

For the serial implementation, the plate $u(t, x, y)$ at time t will be represented by a two-index array, `double u[M][N]` where M and N are the number of points in the x and y directions, respectively. Recall our steady-state update previously, aka, Jacobi iteration:

```
unew[i][j] = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1])/4.0;
```

Like last time, this requires distinguishing between interior and boundary elements of the domain.

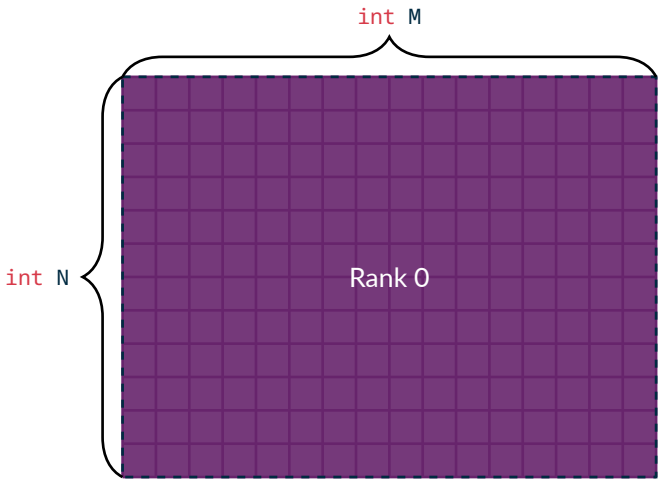

Figure 1: Original discretization.

We will set the top to $T_{\text{hot}} = 100$, bottom to $T_{\text{cold}} = 0$, right to 80, and left to 20. In the corners (e.g., where it is both 'top' and 'left') we will not specify explicitly how to combine it, and leave it to the value which arises out of the setting of the initial conditions. That is, the top-left element of `u[0][0]` may be either 100 or 20, depending on how we establish the initial condition.

Warning

This is bad practice in actual computational science, but for our purposes it's not worth the effort to fix.

Challenge

How might you reasonably combine the information from the boundaries to set the values on the corners? How would you set the initial values so that it always holds, regardless of how you traverse the array? Implement a more robust method of setting the boundary conditions which holds no matter whether we traverse `double u[M][N]` according to the first or second index in the outer loop.

Domain Splitting

As we saw in the MPI stencil exercise, there are a number of ways to split a two-dimensional domain amongst p MPI processes. Let's assume $p = 4$ since this is the smallest number of processes which gives us options for distribution. Additionally, let's assume $M = N$, so we have a square grid and we can make more sophisticated arguments than "split along the longer axis". In this case, we can share the array equally across processes in three different ways: a 2×2 grid, a 1×4 grid, or a 4×1 grid. Let's deal with these cases in turn.

Splitting into a 2×2 grid

This distribution to processes is nice because it is perfectly even (provided `M%2 == 0` and `N%2 == 0`), each rank has an array which matches the same proportions as the original, and we reflect the symmetry of the computation in our distribution. In this case, we map the indices to sequential ranks, as below:

Rank	Region
0	`u[0:(M/2)][0:(N/2)]`
1	`u[(M/2):M][0:(N/2)]`
2	`u[0:(M/2)][(N/2):N]`
3	`u[(M/2):M][(N/2):N]`

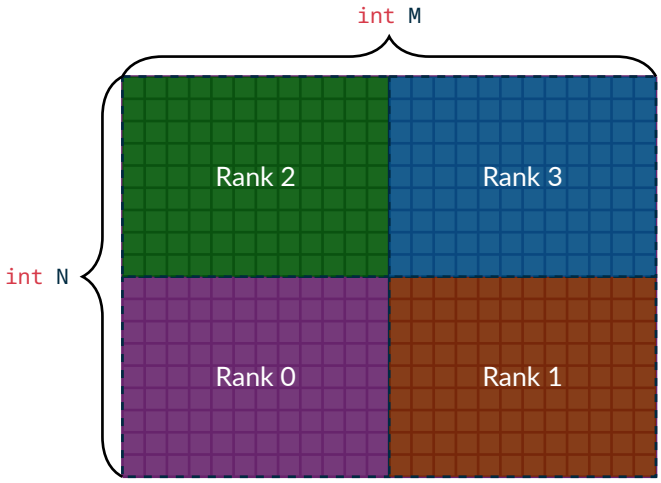

Figure 2: A 2×2 distribution of the original array across 4 processes.

Now, this seems sensible for the initial distribution, but it's something we need to do *on each update* of u. This means rank 0 needs to exchange messages with rank 1 and with rank 2, rank 1 with rank 0 and rank 3, rank 2 with rank 0 and rank 3, and rank 3 with rank 1 and rank 2 – *on every step*. This means that on each iteration, we send at least 4 messages of

size `M/2` and 4 messages of size `N/2`. If we included corner elements in our Laplacian stencil, i.e. if `unew[i][j]` depended on `u[i+1][j+1]`, then rank 0 would *also* need to exchange some (smaller) message with rank 3.

On each process, we would have a local array of size `ulocal[(M/2)+2][(N/2)+2]`, to account for the interior `[(M/2)][(N/2)]` array and `2*(M + N)` border elements for the transfer of edge data due to the stencil width. We would need to transfer row data (easy) and column data (less easy) on each iteration. We will see some ways to deal with the transfer of column data next.

Splitting into a 1×4 grid

This distribution seems more sensible — we need only exhange messages with two neighbors, regardless of stencil pattern (assuming the domain split four ways is much wider than the stencil width). However, we are splitting along the last axis, which is not contiguous in memory (recall the timing exercise from earlier in the term). This is one of the more clear discussions of this topic, and suggests using a custom strided datatype to work around this limitation, or copy your data into and out of intermediate buffers.

Rank	Region
0	`u[0:M][(0*N/4):(1*N/4)]`
1	`u[0:M][(1*N/4):(2*N/4)]`
2	`u[0:M][(2*N/4):(3*N/4)]`
3	`u[0:M][(3*N/4):(4*N/4)]`

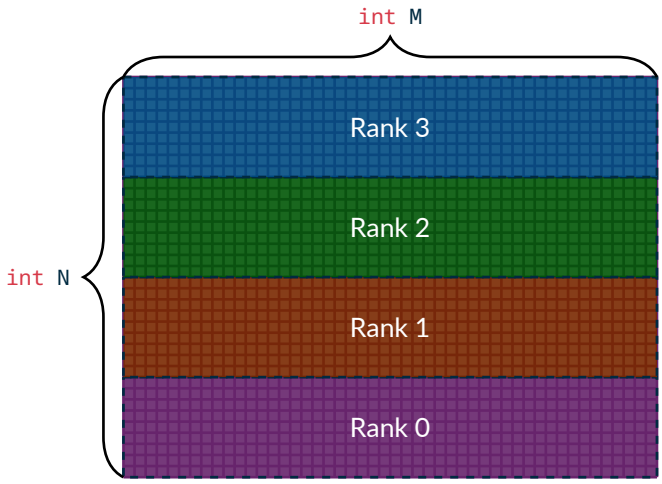

Figure 3: A 1×4 distribution of the original array across 4 processes.

In this scenario we send 1+2+2+1 = 6 messages of size `M` per iteration. Since neither `M` nor `N` is very large, this works out slightly better than the previous distribution, as we send fewer messages, even though the amount of data has slightly grown. It is also a simpler communication pattern, as it can be done in two sweeps (right, then left) rather than four (up, then down; left, then right).

On each process we have an array `ulocal[M][(N/4)+2]`, with the `+2` to account for the boundary exchanges. This is very slightly less data than the 2×2 grid distribution, but the relative savings is not enough to make it preferable. Rather, since we retain the long first dimension, which is the slower to traverse, this distribution works out slightly slower than its transpose.

Splitting into a 4×1 grid

This distribution splits along the slower axis, leaving the faster iteration on each MPI process, which means each process can finish its loop sooner. It exchanges the same amount of data in as many messages as the 1×4 grid (due to the symmetry of the problem), and likewise the local array has the same amount of data.

Rank	Region
0	`u[(0*M/4):(1*M/4)][0:N]`
1	`u[(1*M/4):(2*M/4)][0:N]`
2	`u[(2*M/4):(3*M/4)][0:N]`
3	`u[(3*M/4):(4*M/4)][0:N]`

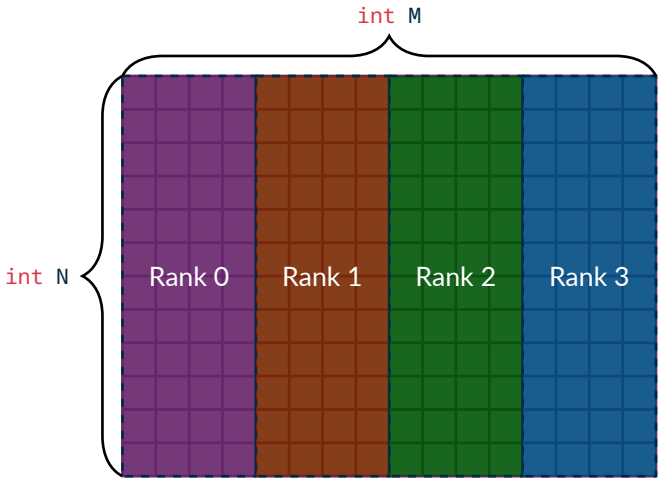

Figure 4: A 4×1 distribution of the original array across 4 processes.

This distribution also has the benefit of easy sharing of rows of data — no intermediate buffers nor bespoke strided datatypes necessary.

Exercise

Given a problem of size `double u[M][N][O]`, with $O = N > M$, how should we split this domain across four MPI processes? Estimate the number of and size of memory transfers for a single update using this decomposition. Justify your decomposition by taking consideration of the speed of on-process computation and message size into account.

Solution

With an array `double u[M][N][O]` where $O = N > M$, then we can simplify and just write `double u[M][N][N]`. The last index will be the fastest to iterate through in C (as we saw in the array timing exercise) and so we would want to keep that dimension intact as much as possible, so long as N/M is not too large. If we split along the first dimension, we will have $3 + 3$ transfers of size $k \times N \times 0$ in each step, where k is the overlap of the stencil. If we split along the second and third dimension analogously to Figure 2, then we would have $4 + 4$ transfers of size $k \times M \times 0/4$ or $k \times M \times N/4$. In the latter case, we have more transfers (because each process has more neighbors, as a baseline) sending $M/(4 \times N)$ times less elements in each message. Thus, in each *round* of messaging (two rounds per step) the second option will send $(4/3) \times M/(4 \times N)$ – or $(M/N)/3$ – as much data. Thus if $M \ll N$, then this might be preferable – it will represent a not insignficant reduction in network congestion, overall – but if $M \approx N$ the increase in the number of messages might present an unnecessary burden on the communication which we would need to work hard to hide. In either case, we would be splitting our fastest array index across processes, which would most likely slow the overall program if we are compute-bound.

Initialization

As with our previous look at the heat equation, we need to initialize the array `double u[M][N]` with values. With MPI, we now have the added wrinkle that `u` is spread across several processes without shared memory access – this is the point of MPI. This can be even more complicated due to splitting – we are setting each boundary line of the array to a distinct value initially, with the interior set to 0. The figure below shows initialization of the combined array using the ***`jacobi.c`*** and ***`jacobi_transpose.c`*** codes.

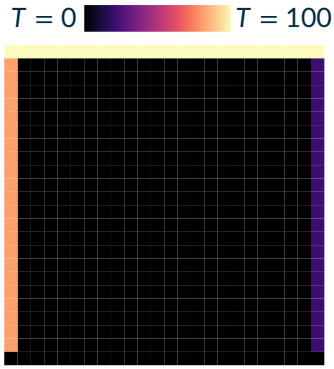

Figure 5: Initial condition of the array as a pseudocolor plot of the temperature using **`jacobi.c`**.

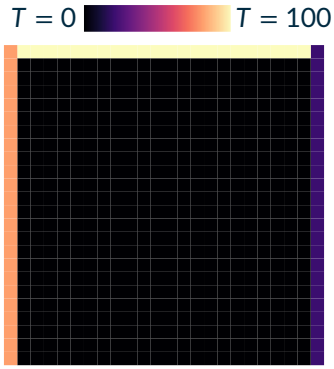

Figure 6: Initial condition of the array as a pseudocolor plot of the temperature using **`jacobi_transpose.c`**.

Exercise

Look at the initialization procedures for the two codes. Can you explain why they produce different initial arrays (when recombined into our original `double u[M][N]`? Can you think of a way to change the initialization in ***`jacobi_transpose.c`*** so that the initial arrays match? Can you think of a reason why it may not be worthwhile doing so for this problem?

Solution

The two codes produce different initializations because they correspond to different sweeps of the arrays on each process. E.g., a global mapping $(i, j) \to T_{i,j}$ will always produce the same initialization so long as (i, j) accounts for the correct offsets for the sub-arrays on each MPI process.

It is likely not worth fixing this issue for the present problem because we are interested in the mechanics of halo exchanges in MPI and thus whether we are getting the physically correct answer is mostly a secondary concern, for now. The other reason I didn't fix this is because it helps illustrate an indexing problem people often run into with MPI – your *local* array coordinates are no longer equivalent to the original *global* coordinates, and this needs to be taken into account.

Update

The update is, locally, the same as when we solved for the steady-state of the heat equation with Jacobi iterations using OpenMP. On each rank, we iterate over the interior points and compute `unew[i][j]` as above, taking care to sample boundary values but not update them. The *new* bit is the communication of the local boundaries between processes; these are rows or columns of the local arrays which are necessary for the other ranks to complete their updates. Additionally, we need to communicate the convergence of the state – how much it's updating on each step – to determine when to end. Both of these require inter-process communication, and we'll discuss them in turn.

Halo exchanges

MPI is not a shared memory parallelisation approach — to update the interior of the state `double u[M][N]`, we need copies of the boundary data on each process. Since our (global) array `double u[M][N]` is of size $M \times N$ (M columns and N rows), we need to figure out how the split onto distinct processes will affect the computation and memory usage. The most

significant change is that this will incur extra memory for each local array, where we store the boundary values from the other ranks.

Hint

How many boundary values we need will depend on the shape of our stencil operation. Since we use a simple 3×3, the furthest extent beyond the boundary that we access is one, in both directions. E.g., if we split the domain such that rank 0 has only 10×10 elements `double u[10][10]`, then we need to access `double u[11][10]` and `double u[10][11]` to update the value `u[10][10]`. Larger stencils thus incur more storage overhead.

This exchange is implemented using `MPI_Send(...)` and `MPI_Recv(...)`, so in some sense, this is not a particularly advanced MPI technique. However, many a simulation has been undone by poorly considered halo exchanges – writing them correctly can be tricky.* It can be made more verbose by the consideration of different process decompositions – see the discussion above for references to bespoke exchange types and temporary variables.

Exercise

Look at the exchanges in the ***`jacobi.c`*** code. Make sure you understand how the transfers are done between processes. Try using `MPI_Ssend(...)` instead of `MPI_Send(...)` – is this a good idea? How many `MPI_Send(...)`/`MPI_Recv(...)` pairs could you swap with calls to `MPI_Sendrecv(...)` – is this a good idea?

Solution

Swapping `MPI_Send(...)` for `MPI_Ssend(...)` is a good idea in this instance if the size of the message is small enough (for the values used in this code, I expect they are). However, if your domain was larger and the messages needed to be exchanged were much larger, then you might see severe degradation of the performance, or worse, complete deadlock.

Similarly, you can replace all the `MPI_Send(...)` and `MPI_Recv(...)` calls with a coordinated `MPI_Sendrecv(...)`, with a very slight restructuring of the code (e.g., ensuring the boundary ranks communicate their halo regions correctly).

Reduction

Our reduction to control the convergence is now more complicated because we lack a shared memory copy of the state. Sending and assembling the interior arrays from each rank to a particular one, forming the original `u[M][N]` array from the pieces, and then doing a serial reduction by iteration over all the indices just as in the serial code (or, indeed, the OpenMP code!) is one option – but it requires a lot of communication (a gather of $N \times M/p$ elements from $p - 1$ processes) and completely serializes the thing we wish to track.

The way it's done in the code is using `MPI_Allreduce(...)` – each rank takes care of it's local `diffnorm`, and then reduces to `gdiffnorm` ("global `diffnorm`") across all ranks. This is much more performant, as we compute the reduction locally serially, but in parallel globally across all the ranks.

Exercise

*In a fortran code I have inherited, there was an incorrect MPI exchange for a 3D PDE system – this required substantial fixes associated with manually checking the correct size of the halo to rewrite all the MPI calls. My advice: do it right the first time.

Why have we used an `MPI_Allreduce(...)` here instead of `MPI_reduce(...)` to a particular rank (say, rank 0)? Do we save anything doing it this way (complexity, bandwidth, number of MPI calls)? Write out how you would solve the same issue using `MPI_Reduce(...)` to rank 0. Is it as clear what is happening?

Solution

We have used `MPI_Allreduce(...)` instead of `MPI_Reduce(...)` because the communication is more performant (significantly fewer values are sent), and we need each rank to have access to that global value. We also only send a single value from each rank, and `MPI_Allreduce(...)` more efficiently constructs the global value from these pieces than manually sending things to a single node (i.e. with `MPI_Reduce(...)`), at least for large p. Furthermore, if we construct the global difference norm on a single rank, we would then need to send it to all other ranks using a `MPI_Bcast(...)` – this is quite expensive, in comparison.

Scaling

Weak scaling is frequently the domain of MPI, as distribution across processes only makes sense if your problem is already pretty large. However, we've coded this example to use 4 ranks, explicitly, by the size of the `double xloc` and `double xnew` variables – and the explicit check! Making a PDE decomposition using MPI robust to arbitrary numbers of processes is not a trivial task, and is frequently only worthwhile if we have a very large problem (certainly larger than the `int maxn` used here).

Exercise

Make a copy of the ***`jacobi.c`*** code and modify it to use only *two* processes. What needs to be changed? Test your code, is it faster or slower than the 4 process one? Based on the two and four process codes, what parts of the code would need to be expanded for a theoretical 8 process version?

In theory, the most processes you could have is `maxn*maxn`, or only `maxn` if sticking with the 1-by-q decomposition pattern. Estimate the number of exchanges and extra data you would need in these limiting cases. Do they seem worthwhile?

What, other than the number of ranks, would need to change to do a weak scaling analysis of this code? Can you estimate what size array you should have on each process from your 2 and 4 rank codes?

Solution

For the ***`jacobi.c`*** code with 2 processes, we need to ensure we are:

- creating the correct array sizes on each process,
- exchanging the correct number of messages, especially since every rank is a boundary rank,
- and send the right size message.

You are very likely to find the 2-process code is faster than the 4-process code, especially on modern hardware. We are very unlikely to be compute-bound in this problem, and thus we benefit from simply having to wait less to get the data. For an 8-process version of the code, the concerns are the same but we are liable to spend substantially longer in MPI calls, and have a slower code overall.

In the extreme cases: we have N^2 processes exchanging their 1 element of their data with 4 neighbors in each iteration, or we have N processes exchanging their N elements with two neighbors in each iteration. In the former case we would need 4 extra elements per local element in our array, and due to the rectangular structure of such, likely 8. In the latter case, we would need 2 extra elements per local element in our array. In niether case does it seem worthwhile to send both substantially more messages, of smaller size, with huge extra memory costs.

The only real concern for weak scaling is the performance, rather than correctness. So we should check that the initialization doesn't produce unreliable starting values for any of the p we investigate, and we should ensure that we *fix* the number of steps we do, rather than exit when convergence is satisfied. That is, the number of iterations depends on how close we *start* to the correct answer, so we want to ensure we aren't producing wildly incorrect starting conditions and we want to ensure the actual amount of work is being held constant, per process. For a weak scaling, I would recommend using a large base size for the array – e.g., the whole array on each process – as we are unlikely to be compute bound.

Jacobi and the transpose

In ***jacobi_transpose.c*** we have an additional declared variable `xedge[maxn+2]` for temporary copies when sharing edge data across ranks. This means explicit copying before `MPI_Send(...)` and after `MPI_Recv(...)` on each transfer, adding latency to our communication.

Exercise

Compare the performance of the ***jacobi_transpose.c*** code to ***jacobi.c*** code with the same number of ranks and (as near as you care to make) initial conditions, or otherwise fix the number of iterations. Is the extra latency incurred by memory copies visible in the transpose version?

In your 2 rank ***jacobi.c*** code, you made some changes from the original 4 rank code; what similar changes need to be made for a 2 rank transpose code?

Solution

In my testing the difference in timing wasn't significant enough to peak above the noise of running the codes with 4 processes. In a larger code, where the memory use is more significant, you would expect the transpose to incur about a microsecond of delay for every transfer of roughly 1000 `double`s.

To make the ***jacobi_transpose.c*** code work with 2 processes, the changes are very similar – you just ensure the sizes of the local arrays, messages – and now also the transposition! – are correct.

Aims

- Consideration of the parallel distribution of a PDE in two spatial dimensions for communication overheads.
- Parallel implementation of PDEs using MPI
- Explicit consideration of serial and distributed memory allocations for high performance computing