Message-Passing and MPI Programming
Problem Decomposition

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

10.1 Objective

This lecture does not teach you **what** to do, because that is fundamentally problem-dependent. It describes some important possibilities that you should be aware of. Specific uses of the techniques it mentions will probably be described in other courses, especially ones on mesh generation.

If all of the processes do the same work, there is obviously no speedup, so the first requirement is to divide up the work between the MPI processes.

Many scientific requirements (i.e. algorithms) work on updatable data, such as the matrix in Cholesky decomposition or a FFT, and there is often too much data for any one of the processes. So we need to consider dividing up the data, too.

This has nothing to do with MPI, as such, and it applies equally to all distributed memory parallelism – in fact, the first requirement (and to some extent the second) also apply to to shared memory codes (including on GPUs). It is usually more difficult than using MPI, but it is critical to the resulting efficiency. The same remark applies to shared memory parallelism, too, but the details of the requirements and constraints differ in most details.

The best approach is *KISS – Keep It Simple and Stupid* – and here are the initial guidelines:

- Always start with the simplest partitioning that looks plausible for your problem.
- Design your program to allow for change later; this is far easier to say than to do.
- Balance the workload across the processes, so they all have roughly the same amount of work to do.
- Design to minimise the amount of inter-process communication needed.
- Gathering data is what reductions are for.

Check if the work is reasonably well balanced across processes and the performance is reasonable. Only if it is not, redesign the partitioning for more efficiency. As always, such rules have exceptions, but that is how you should start.

10.2 Embarrassingly Parallel Problems

These are problems that divide up naturally into a large number of separate tasks, such as Monte-Carlo work or parameter space searching. Each task is largely independent of all of the others, so there is little need for communication between tasks. The approach

is for a master process to spawn tasks and collect results, with little or no interaction except during startup and termination. You just divide the tasks between processes, and normally give all of them an equal number of tasks. Very often, that is all you need to do.

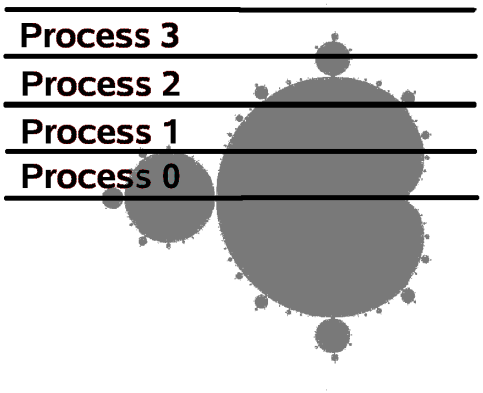

Figure 10.1

Remember the *Mandelbrot set* example? That divided into sections along the Y axis, and it did not work very well, as we saw. The problem was the correlation of the time time with the partitioning. Randomising the points before assigning them to processes worked much better. You can often use a cyclic partitioning instead – anything that breaks up the correlation will do.

The problem is if the task time is very variable. It is easiest to regard it as a statistical distribution. The problem is easy when the standard error is smaller than the mean, and harder when it is much larger than the mean. The reason is that the scalability relies on the *Law of Large Numbers*, and the general rule is to give each process lots of tasks in a a single run of the program, preferably much more than $(\text{S.E.}/\text{mean})^2$. It you cannot, the only problem is inefficiency.

Most non-statisticians believe that the *Law of Large Numbers* is a universal law, but that is not true, and it does not **always** hold. Very nasty distributions are surprisingly common in practice, and randomisation does not help much if the time distribution has no mean. We need to be a bit cleverer in that case, and to write a simple queue manager. The technique is useful anyway, so it is well worth learning, but it will not necessarily deliver high efficiency if the time distribution is truly foul. However, if it does not work well, nothing will.

Such a design is based around a master process, which gives each worker process one task to do, and gives it another when it finishes the previous one. The master process does not do any computation itself; you can get it to do so, but that counts as more advanced use. Writing a queue manager is not difficult, and is a good exercise in using `MPI_Probe` or `MPI_Waitany`. These is still a problem if some tasks never complete, but that also causes statistical and numerical problems, and so needs deeper analysis.

10.3 Partitioning

In general, problems cannot be split into semi-independent tasks, and any division will imply a fair amount of inter-task communication. To tackle those, we need to decide how to partition the problem. The most important guideline is to try to minimise inter-process communication, as will be covered in the next lecture. Looking ahead, the objectives are:

- Minimise the amount of data transferred between processes, because bandwidth is a limited resource.
- Minimise the number of transactions involved in transfer, both because each one has some overhead and because there is always a minimum latency.
- Minimise the circumstances when one process is waiting for another, because it is wasting time while it is waiting.

The following slides describe some possibilities; they are not exhaustive, but cover the main ones used in scientific codes. We start with problems that split up naturally – it is generally a good idea to consider using natural divisions if you have them.

- A problem may have semi-separate components, such as different species in an ecology, or different compounds in a composite material, or the components of an engine.
- A problem may have a graph structure; in mathematics, that is nodes connected by links. This is often a mathematical formulation of a problem of the previous type.

In such cases, the components or nodes are the units of data that need distribution, and their connexions or links are the communication paths; a mapping to MPI is often obvious but may not be optimal. Generally, you should look for a division that minimises either or both of the number of links or the communication traffic across those links. There is, of course, a compromise between this and balancing the workload. But the universal rule is that, if it works well enough, it is right; do not waste time trying to achieve perfection.

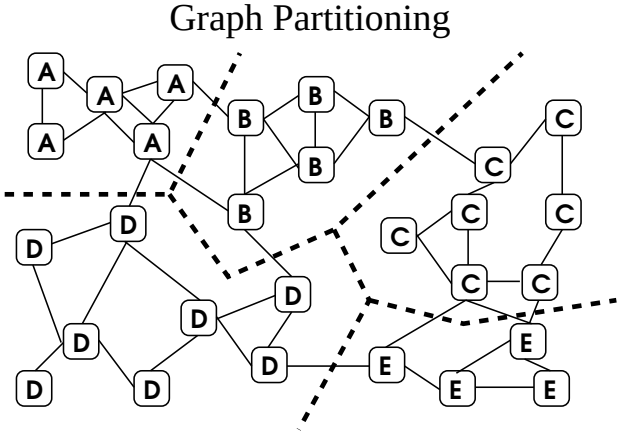

Figure 10.2

10.4 Regular Grids

Many scientific problems are described in terms of rectangular grids, sometimes because it is a natural decomposition, and sometimes just because things have been traditionally done that way. It is so dominant that most books and Web pages consider only this form of layout. The regularity has the advantage that you can often analyse its properties, and can sometimes choose the best design before you start coding. Even if you cannot, they are usually easy to parameterise, and so tune by experimentation on real data. *ScaLAPACK* puts a lot of effort into supporting this, in a large amount of generality, but it ends up being too complicated and confusing.

Usually, you should divide such problems divide into contiguous blocks, which gives good locality for simple, uniform problems. It is best to start with such a design, remembering to parameterise the block layout; this always means the size of blocks, and often their shape. Dividing areas into strips or volumes into layers usually involves more communication, and so is less efficient.

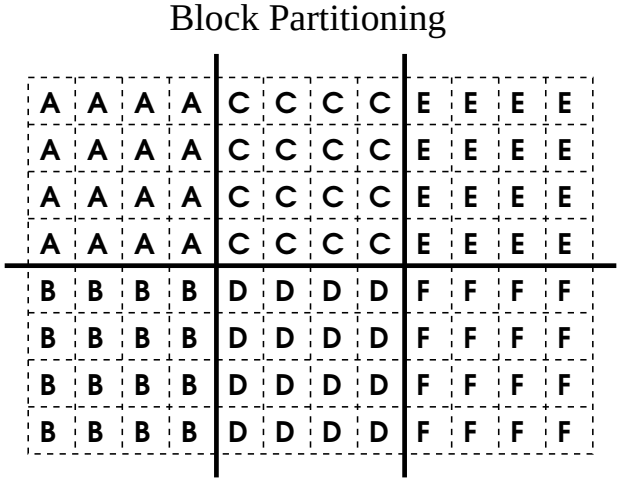

Figure 10.3

You can sometimes do better with a cyclic distribution, of which there are many variants. You can also combine designs, such as cycles of blocks, or cyclic in one dimension and blocked in another, or whatever makes sense for your problem. As usual, the simplest solution that works is the best one.

2−D Cyclic Partitioning (1)

A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F

Figure 10.4

2−D Cyclic Partitioning (2)

A	B	C	D	E	F	A	B	C	D	E	F
F	A	B	C	D	E	F	A	B	C	D	E
E	F	A	B	C	D	E	F	A	B	C	D
D	E	F	A	B	C	D	E	F	A	B	C
C	D	E	F	A	B	C	D	E	F	A	B
B	C	D	E	F	A	B	C	D	E	F	A
A	B	C	D	E	F	A	B	C	D	E	F
F	A	B	C	D	E	F	A	B	C	D	E

Figure 10.5

Problems are very commonly non-uniform – the flow of a fluid around a sharp corner is a classic example. The numerical difficulty and hence the amount of work can be much higher near the corner than a long way away from it. Uniform partitioning may not work very well, and you have no option but to balance the workload better. That is much more complicated to program and tune, but sometimes gives vastly improved efficiency, and makes parallelising a problem worthwhile where it was not with a uniform grid. There are many different ways of doing this, including multi-grid methods, mesh refinement and transforming the coordinates of the grid corners. But, remember, always start with simple partitioning, and complicate only when you have to.

Irregular Partitioning

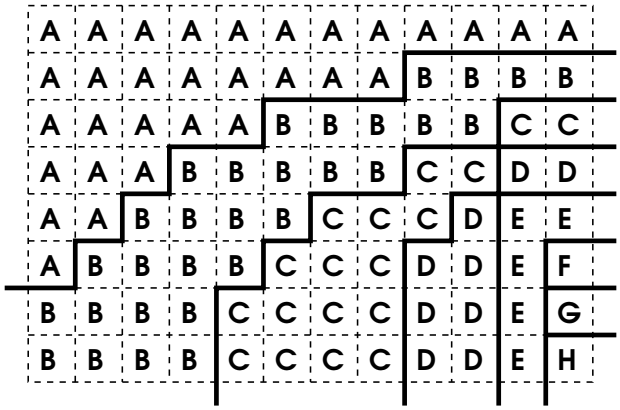

Figure 10.6

Mesh Refinement

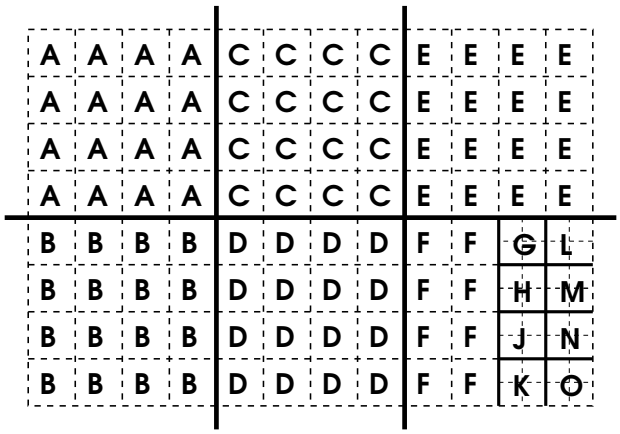

Figure 10.7

Transformed Mesh

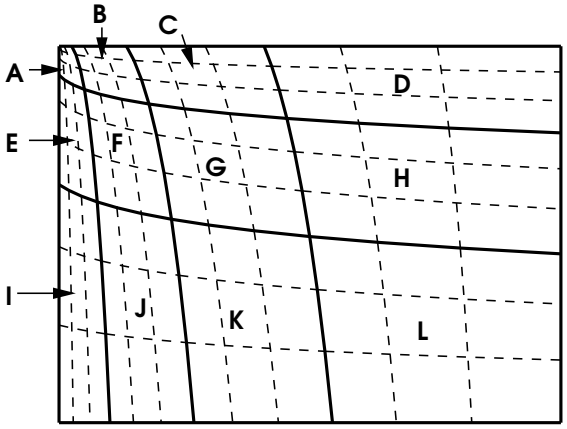

Figure 10.8

Rectangles are not the only space-filling shapes. Triangles (in 3-D, tetrahedra) are common, too, and are widespread in finite-element work. These are regular, just like rectangular blocks, but trickier to code; they introduce no new principles. You may also come across other ones, not all of which will be regular in strict mathematical terms.

Triangles (or Hexagons)

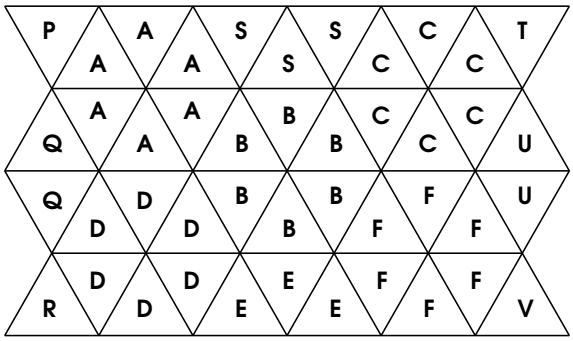

Figure 10.9

There are also *Voronoi diagrams*, also known as *Dirichlet tesselation*, also known as *Delaunay triangulations* and so on; you may be taught these in mesh generation lectures.. They are generally used for irregular problems – as far as MPI is concerned, program them as a form of graph partitioning.

They have some very useful mathematical properties. A Voronoi diagram divides the space into regions by the point it is closest to, and a Delaunay triangulation is the one with the numerically best-defined triangles (i.e. fewest long, thin ones). They can be used for efficient multi-dimensional searching and in many other ways.

Voronoi Diagram (a.k.a. Tesselation)

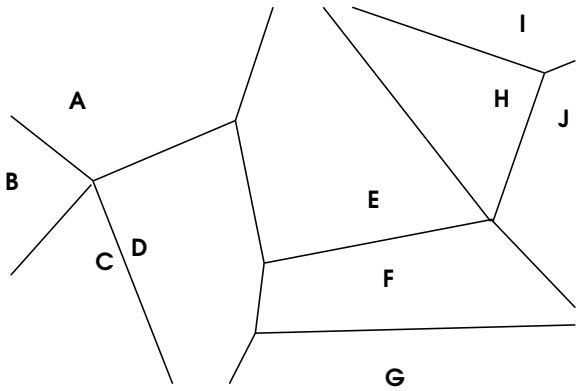

Figure 10.10

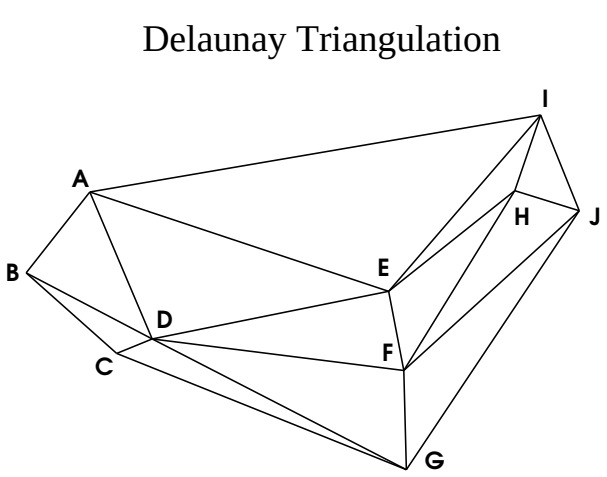

Figure 10.11

10.5 Communication

Sometimes this is explicit in the problem, such as modelling systems of active components, as described above. This is very common when the problem has graph structure, and the links are the communication paths. So you can just code it using MPI's facilities, usually point-to-point, and all you need to do is to minimise the wait time and the amount of data transferred.

More often communication comes in the form of access to non-local data, especially with regular grids. There are two common variants of this:

- Direct access, where a process accesses the data immediately the code needs it, just as if it were local. This is often called virtual shared memory.
- Division of the computation into time-steps, and communicating the data between processes between time-step. This is probably the most common design.

You are **strongly** advised to avoid virtual shared memory, because it is extremely hard to design and use correctly, and introduces some very nasty race conditions. Some designs (e.g. *Fortran coarrays*) support this, but experts spend a long time designing them to make sure that they are implementable and usable. The problem is not with MPI, but with the design and discipline, and they will often be built on a basis of MPI. If you need to implement virtual shared memory, use one of those designs, and do not invent your own.

10.6 Time-Step Designs

Time-step designs are very common for problems like PDEs and ODEs, at the basic numeric level (e.g. *Runga-Kutta*), so your application may well be using them anyway. You can then resynchronise your data between processes each time-step, and you can do it either by writing or reading. In the former, each processes creates updates to other processes' data, and sends them to the other processes between each time-step; that is less commonly used and not covered further. In the latter case, each process broadcasts its

local data between each time-step, and uses the data broadcast by the others during the next time-step.

The general approach is that each process owns a subset of the global data, which it updates – other processes can only read the data it owns. In the simplest case all processes broadcast all data to all processes between time-steps, so that each has read access to the complete, global data during the next time-step. This is good if there are long time-steps, and the amount of data is small, but often there may be too much data, or the broadcast cost is too high. Its advantage is that it is very easy to design and use correctly.

More commonly, you distribute only the data that will be needed, which is typically the data near the boundary of processes. PDEs are an obvious example, because they need only the nearby data. This is obviously more complicated to design and program, but it can reduce the memory needed a great deal (because each process no longer needs a copy of all the data) and it can reduce communication cost even more (because only the boundary data are distributed).

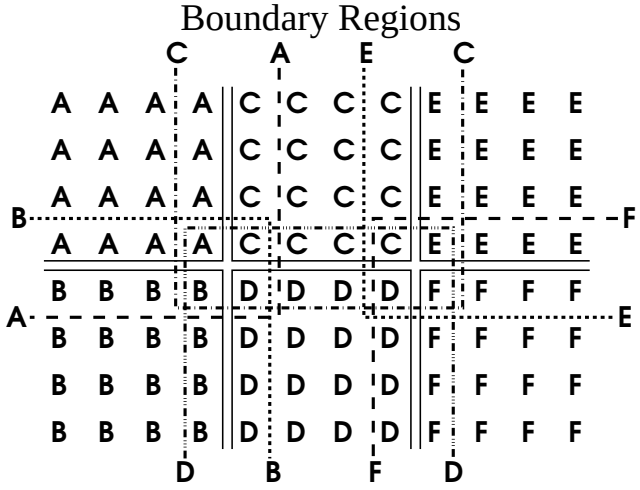

Figure 10.12

The usual way of implementing this is that each process stores its own data plus copies of the data in the boundaries of neighbouring processes. It updates only its own data, not the boundaries that it does not own, and gets the updated boundaries from the other processes that own them. It sends its own updated edge data (i.e. the data that it owns that form the boundaries of other processes) to its neighbouring processes. As usual, the guidelines are to keep your design simple, code it carefully, and do not worry about minor inefficiencies.

Note that a boundary of width N means that you need resynchronise only every N steps, which usually reduces communication time, at the cost of slightly increasing the computation time and memory requirement.

10.7 Other Aspects

Most boundary-distribution designs assume that the boundaries are very thin relative to the data owned by a process. If that is not the case, the scalability is usually extremely

poor, and it is usually better to broadcast all of the data. There are two guidelines for when to consider doing that:

- Most of the data is in some node's boundary, because you probably will not gain any performance by distributing only the boundaries.
- The boundaries go beyond the immediate neighbours (i.e. to the process on the other side of the neighbouring process); it is fiendishly hard to program boundary-only distribution correctly in such a case. Multi-cell boundaries are no problem, however.

Sometimes part of a program needs one form of partitioning, and other parts need different ones. You can repartition the data between those parts, though repartitioning all of your data can be extremely slow. The key to performance is to keep the parts of program very separate, almost like different programs merged into one; it is generally not worthwhile on simple programs or when the parts take only a short time.

Probably the simplest form of repartitioning is simple reblocking, which takes one blocked design and converts it to another; this is fairly often used even in simple programs. It might help to think of matrix transpose (such as in `MPI_Alltoall`), which is commonly used to implement n-D FFTs efficiently. For those, a serial FFT algorithm is used on one dimension, treating the other dimensions as vector data. This is most efficient if the processes divide up the vectors, which requires reblocking (i.e. a transpose) between each dimension's serial FFT.

MPI has some facilities for managing decompositions, especially parameterised ones. See later under *Topologies* in *Extra Lectures*.

As a reminder, partitioning is key to efficiency in many problems.

- Do not rush in – design it carefully
- Choose the one that best matches your problem
- *KISS – Keep It Simple and Stupid*

There are a couple of heavyweight practicals for this lecture, which are very similar to using MPI on real scientific codes. If you can do them without too much problem, you will be able to handle most MPI programming. Be warned that the second took me a good week's work to write the specimen answers, though the actual MPI coding was easy!

10.8 Extra Lectures

There are some other lectures that only have slides and no notes, and are not part of this course, mainly due to lack of time. If you simply follow the guidelines you have been given, you will be able to write some very serious programs and are unlikely to hit serious trouble; the extra information is needed mainly for writing portable production code. Some of them may be worth looking through a bit later, when you are happy using the facilities you have been taught; they expand on the guidelines given in this lecture. As a reminder, they are in:

```
MPI/
```

- *Topologies.* MPI has some potentially useful facilities for managing n-D indexing, which could help to make code clean and parameterisable. It also has facilities for

management of graph decompositions, which are definitely complicated, but that is the nature of graph decomposition.

- *Composite Types and Language Standards.* This is mainly on what **not** to do, but avoiding the "gotchas" is very important. The main ones have already been described. Little of this is actually about MPI as such, but more about the arcane properties of Fortran, C and C++.
- *Debugging, Performance and Tuning.* This includes an overview of debugging tools, but is mostly things that you need to know if you have a difficult tuning problem. You should follow the guidelines and try to avoid that.

There are some others that are in the Web pages, if you want to look through them, but you probably will not need to. They are likely to be relevant mainly if you have to work on a program that uses facilities that have not been covered in this course. They are:

- *Attributes and I/O.* This is mainly going though I/O handling in more detail, how to do it "properly", and describing some of the nastier system configurations you may encounter.
- *One-sided communication.* This describes some of the problems with one-sided communication and RDMA, summarises MPI's support for it, and teaches a simple and relatively safe way of using one-sided communication.
- *Advanced Completion Issues.* This is the aspect of point-to-point usage we have not covered. If you follow the recommendations given in the previous point-to-point lectures, you can ignore this.
- *Other Features Not Covered.* This is what the course has not covered and why. A few of the aspects are important to a very few people, but most people will be able to ignore this.

However, if you **do** read through those lectures, you will have at least some knowledge of almost the entirety of the current MPI standard and its use. In many cases, all you will know is that a facility exists, but that would enable you to look it up if you need it.