Numerical Simulation Methods

Part 2 - Applied Concepts

Lecture 16: Parallel Processing

Roadmap

Applied Concepts

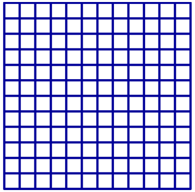
- Introduction to meshing
- The finite volume method
- Jameson's scheme
- Solution storage approaches and their implications
- Advanced implicit methods
- Introduction to computer hardware and high performance computing

  Today: Parallel decomposition and efficiency
  - **Domain decomposition**
  - **Load balancing**
  - **Parallel efficiency**

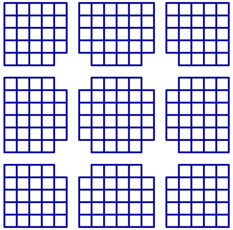
Parallel Processing

There has been substantial activity in the last 20 years in developing *parallel* computers.
Instead of buying one extremely expensive, very powerful single vector processor with large RAM (50-100Gbytes), the approach is to obtain massive computing power by coupling very many (tens to thousands) of relatively cheap processors together.
By splitting the computational task over the processors, and getting the processors to work simultaneously on their piece of the problem, very fast processing speed is possible. A crude estimate is that the effective speed is equal to the speed of one processor times the number of processors. Of course there are additional overheads involved in splitting up the task, but efficiencies of over 95% are possible.

Consider a 12×12 computational grid.


Computational Grid

Assuming the finite-volume stencil uses five points (one neighbour required for each cell face), if this mesh is uniformly split over 9 processors, each processor requires the cells below. Each block has a 4×4 block of inner points plus halo cells required to compute fluxes.

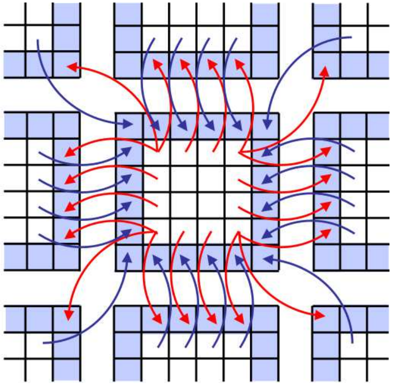

Split Computational Mesh Blocks

These halo cells contain the solution values from neighbouring blocks, i.e. stored and updated by different processors. Hence, parallel solution introduces a communication overhead. After each time-step the solution values in the halo cells have to be passed between processors. Clearly the more points there are on each processor the better, since the halo cells become a smaller percentage of the total, and the communication overhead reduces accordingly.
If the stencil is bigger the communications get more expensive and more complex.

calculation cost:
scales as n^3

communication cost:
scales as n^2

strong scaling:
increasing P
decreasing n
comms will dominate



Now consider the code structure. Consider the operations performed on a 100 x 100 cell mesh. The time-stepping loop is structured as follows:

```
 DO NTIME=1,NTMAX
   Apply boundary conditions at I=1 and 100, and J=1 and 100

   DO J=1,100
     DO I=1,100
       UPDATE CELL SOLUTION (I,J)
     ENDDO
   ENDDO

ENDDO
```

Hence, 100x100 = 10000 cell updates per time-step.

Now assume we have 100 CPUs so can send each block to a separate processor, so on each processor:

```
 DO NTIME=1,NTMAX
   Apply boundary conditions if block face on boundary
   Get halo data from/send halo data to neighbour processors

   DO J=1,10
      DO I=1,10
         UPDATE CELL SOLUTION (I,J)
      ENDDO
   ENDDO

ENDDO
```

Hence, $10 \times 10 = 100$ cell updates per time-step per processor, so 100 times faster (ignoring data passing).

Multi-block Load Balancing

If we have more mesh blocks than processors - we need to put more than one block on a processor.

But what if our blocks each have different numbers of cells?

- We will have a different number of cells on each processor
- Each processor will take a different amount of time to update solution
- BUT: Must wait for all processors to finish before starting next iteration
- THEREFORE: time-per iteration is determined by the processor with the most cells

Load balancing: for 100% efficiency (ignoring communications) we want all processors to have exactly the same number of cells.

Unstructured mesh doesn't have blocks: can easily divide the mesh for an arbitrary number of processors.

Simple Example Figures

Consider a steady state computation, which requires 1000 time-steps on the 12×12 cells.

$$\Rightarrow 1000 \times 12 \times 12 = 144,000 \text{ operations}$$

Divide this task over 9 processors as above. If all processors work simultaneously the run time is the time for each processor to complete its task

$$\Rightarrow 1000 \times \{4 \times 4 + \text{processor communications}\}$$

Assume communication takes 25% of time to update a cell, halo cells are 2/3 of inner cells (24 boundaries $\times 4$ cells)

$$\Rightarrow 1000 \times \{4 \times 4 + 4 \times 4 \times 2/3 \times 0.25\} = 18,667 \text{ operations}$$

Hence, the speedup is

$$\text{Speedup} = \frac{144000}{18667} = 7.71$$

and the efficiency (for 9 processors)

$$\text{Efficiency} = \frac{7.71}{9} = 85.7\%.$$

Simple Example Figures 2: Double mesh size

Consider a steady state computation, which requires 1000 time-steps now on the 24 × 24 cells.

$$\Rightarrow 1000 \times 24 \times 24 = 576,000 \text{ operations}$$

Divide this task over 9 processors as above. If all processors work simultaneously the run time is the time for each processor to complete its task

$$\Rightarrow 1000 \times \{8 \times 8 + \text{processor communications}\}$$

Again assume communication takes 25% of time to update a cell, halo cells are now 1/3 of inner cells (24 boundaries ×8 cells)

$$\Rightarrow 1000 \times \{8 \times 8 + 8 \times 8 \times 1/3 \times 0.25\} = 69,333 \text{ operations}$$

Hence, the speedup is now

$$\text{Speedup} = \frac{576000}{69333} = 8.31$$

and the efficiency (for 9 processors)

$$\text{Efficiency} = \frac{8.37}{9} = 92.3\%.$$

Parallel Decomposition and Timings

Consider a mesh containing $N \times N$ points, which are distributed uniformly over P processors. Four times are defined:

t_{proc} - time to execute one timestep for one mesh point on the processor

t_{comms} - time required to send the data associated with one boundary mesh point between processors

t_{global} - time for global data transfer

t_{serial} - time for serial part of code.

As a rough estimate, the total time to compute a single timestep on the parallel computer is thus the sum of these times, or

$$T_N(P) = \frac{N^2}{P} t_{proc} + \frac{N}{\sqrt{P}} t_{comms} + \log(P) t_{global} + t_{serial}$$

The *speedup*, $S_N(P)$, is the ratio of the time to execute a problem of size N on one processor to the time to execute the same problem on P processors.

$$S_N(P) = \frac{T_N(1)}{T_N(P)}$$

$$= \frac{N^2 t_{proc} + t_{serial}}{\frac{N^2}{P} t_{proc} + \frac{N}{\sqrt{P}} t_{comms} + log P t_{global} + t_{serial}}$$

$$= \frac{P t_{proc} + \frac{P}{N^2} t_{serial}}{t_{proc} + \frac{\sqrt{P}}{N} t_{comms} + \frac{P log P}{N^2} t_{global} + \frac{P}{N^2} t_{serial}}$$

The efficiency of the parallelisation is η

$$\eta = \frac{S_N(P)}{P}$$

$$= \frac{t_{proc} + \frac{t_{serial}}{N^2}}{t_{proc} + \frac{\sqrt{P}}{N} t_{comms} + \frac{P log P}{N^2} t_{global} + \frac{P}{N^2} t_{serial}}$$

}

Now consider fixed problem size N with varying number of processors. Consider first the ideal case with no global communication overhead and no serial component (i.e. $t_{global} = t_{serial} = 0$).

$$\eta = \frac{t_{proc}}{t_{proc} + \frac{\sqrt{P}}{N} t_{comms}} = \frac{1}{1 + \frac{\sqrt{P} t_{comms}}{N t_{proc}}}$$

If N is fixed, this *decreases* with increasing P. This is as expected, since as P increases the number of points on each processor decreases so the halo cells become a larger and larger percentage of the total cells on each processor. How quickly this reduction in efficiency takes effect depends on two factors, the *grain size*

$$n = \frac{N^2}{P}$$

i.e. the number of points per processor, and the ratio of communication time

$$\bar{t} = \frac{t_{comms}}{t_{proc}}$$

With these definitions we can rewrite the equation as

$$\eta = \frac{1}{1 + \frac{\bar{t}}{\sqrt{n}}}$$

The efficiency is seen to increase with decreasing $\bar{t}$ and increasing grain size n.

CFD Methodologies

The interprocessor communication depends on the flow solver.

- NEAR-NEIGHBOUR COMMUNICATION
  - Explicit finite-difference schemes
  - Explicit Finite-volume schemes

For these solvers only halo cell data has to transferred between neighbouring processors.

- GLOBAL DATA DEPENDENCY
  - Implicit finite-volume or implicit finite-difference schemes
  - Panel methods
  - Spectral methods
  - Point vortex models

In these cases every processor has to communicate with every other one.

Example: consider the matrix system for an implicit scheme. Two options:

1. Each processor sends entire matrix system to a master process to construct the global matrix and solve (maybe in parallel);
2. Each processor constructs its own small matrix system and solves it locally $\rightarrow$ global dependence is lost, unless a global dependence is reconstructed later.

Summary

• Domain decomposition: split the domain into a number of smaller ones, and send each one to a separate processor to be solved simultaneously. Each domain needs to communicate with some (or all) the other domains, so CPUs need to communicate resulting in a communication overhead.

• The amount of communication required for each block/domain depends on its boundary size, and so the efficiency depends on domain boundary/volume, i.e. surface area/volume. Hence, the efficiency depends on the grain size, the number of cells per processor, and efficiency drops as this decreases.

• This is 'conventional' parallel processing, based on domain decomposition and minimising message passing.