

An Introduction to Parallel Programming

Paul Burton
January 2016

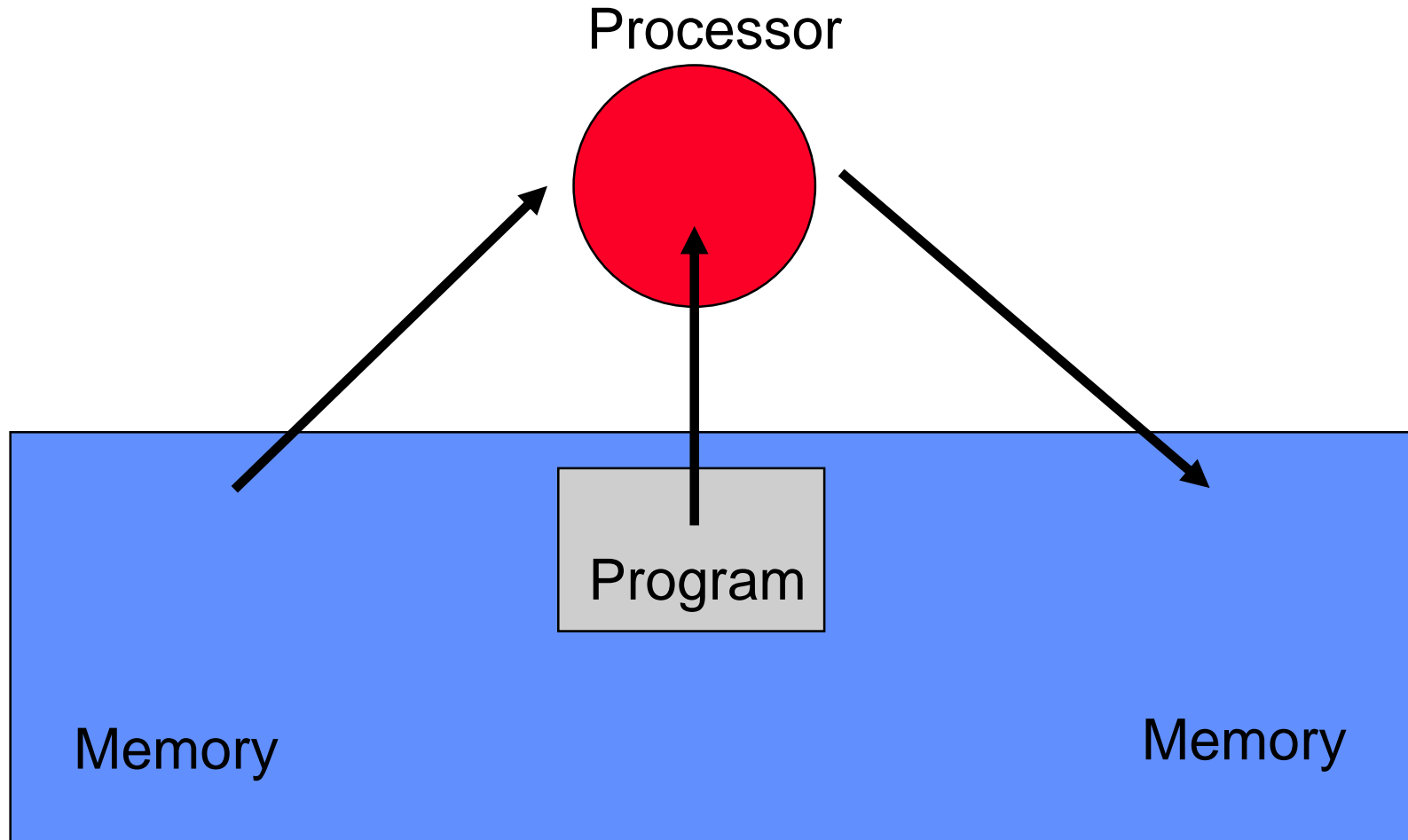
Introduction

- **Syntax is easy**
 - And can always be found in books/web pages if you can't remember!
- **How to think about parallel programming is more difficult**
 - But it's essential!
 - A good mental model enables you to use the OpenMP and MPI we will teach you
 - It can be a struggle to start with
 - Persevere!
- **What this module will cover**
 - Revision : What does a parallel computer look like
 - Different programming models and how to think about them
 - What is needed for best performance

What do we see? - How do we see!?



What does a computer do?



How do we make it go faster? [1]

- **Make the processor go faster**
 - Give it a faster clock (more operations per second)
- **Give the processor more ability**
 - For example – allow it to calculate a square root
- **But...**
 - It gets very expensive to keep doing this
 - Need to keep packing more onto a single silicon chip
 - Need to make everything smaller
 - Chips get increasingly complex
 - Take longer to design and debug
 - Difficult and very expensive for memory speed to keep up
 - Produce more and more heat

How do we make it go faster? [2]

- **Introduce multiple processors**

- **Advantages:**

- “Many hands make light work”
- Each individual processor can be less powerful
 - Which means it’s cheaper to buy and run (less power)

- **Disadvantages**

- “Too many cooks spoil the broth”
- One task – many processors
 - We need to think about how to share the task amongst them
 - We need to co-ordinate carefully
- We need a new way of writing our programs

Limits to parallel performance?

- **Parallelisation is not a limitless way to infinite performance!**
- **Algorithms and computer hardware give limits on performance**
- **Amdahl's Law**
 - **Consider an algorithm (program!)**
 - **Some parts of it (fraction “p”) can be run in parallel**
 - **Some parts of it (fraction “s”) cannot be run in parallel**
 - **Nature of the algorithm**
 - **Hardware constraints (writing to a disk for example)**
 - **Takes time “t” to run on a single processor**
 - **On “n” processors it takes : $T = s \times t + (p \times t)/n$**

Consequences of Amdahl's Law [1]

- **$T = s \times t + (p \times t)/n$**
 - Looks simple, but “s” has devastating consequences!
- **Consider the case as the number of processors “n” grows large, then we get:**
 - $T = s \times t + [\text{something small}]$
- **So our performance is limited by the non-parallel part of our algorithm**

Consequences of Amdahl's Law [2]

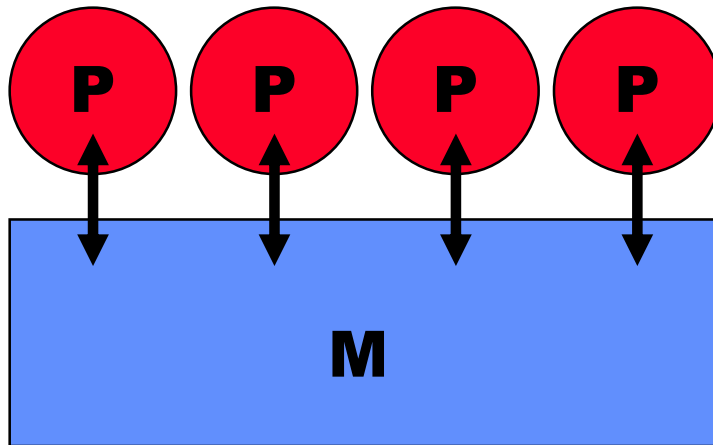
- For example, assume we can parallelise 99% of our algorithm, which takes 100 seconds on 1 processor.
- On 10 processors we get : $T[10] = 0.01 \cdot 100 + (0.99 \cdot 100) / 10$
 - $T[10] = 1 + 9.9 = 10.9$ seconds
 - 9.2 times speedup : not too bad - we're "wasting" 8%
- But on 100 processors we get :
 - $T[100] = 1 + 0.99 = 1.99$ seconds
 - 50 times speedup : not so good – we're "wasting" 50%
- And on 1000 processors we get :
 - $T[1000] = 1 + 0.099 = 1.099$ seconds = 90 times speedup : terrible!
 - We're "wasting" 91%!

How do we program a parallel computer?

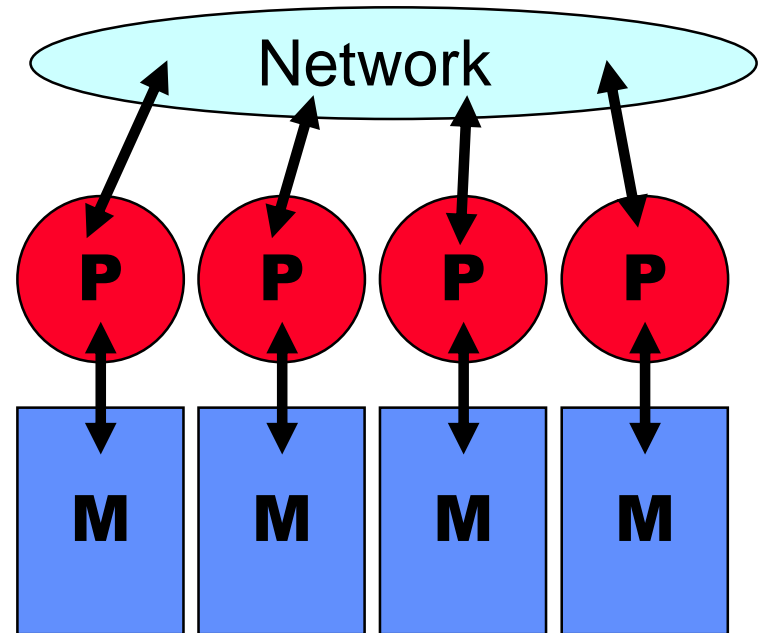
- **Decompose (split) into parts**
 - Algorithm (the program) [*eg. Car production line*]
 - or
 - Data [*eg. Telephone call centre*]
- **Distribute the parts**
 - Multiple processors work simultaneously
- **Algorithmic Considerations (algorithm/ data dependencies)**
 - Need to ensure the work is properly synchronised
 - Possibly need to communicate between processors
- **Hardware Considerations**
 - What parallel architecture (hardware) are we using?

Parallel architectures (revision)

- **Parallel programming technique will reflect the architecture**



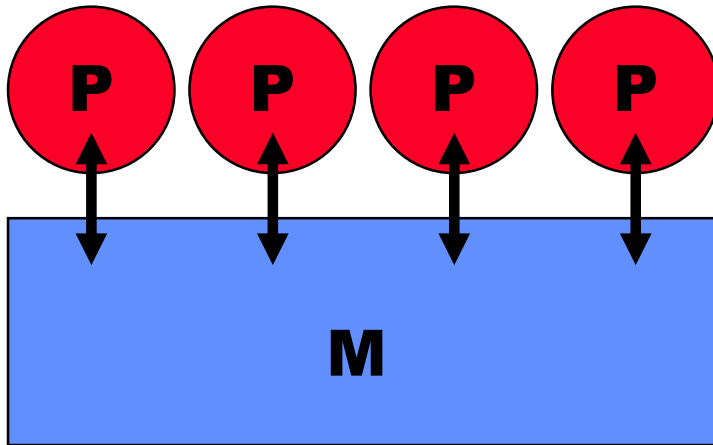
Shared Memory



Distributed Memory

Shared memory programming

Each processor runs a single “thread”



- **Split (decompose) the computation**
 - “Functional parallelism”
- **Each thread works on a subset of the computation**
- **No communication**
 - Implicit through common memory
- **Advantages**
 - Easier to program
 - no communications
 - no need to decompose data
- **Disadvantages**
 - Memory contention?
 - How do we split an algorithm?

A simple program

```
INTEGER, PARAMETER      :: SIZE=100
REAL, DIMENSION (SIZE) :: A,B,C,D,E,F
INTEGER                  :: i
```

```
! Read arrays A,B,C,D from a disk
CALL READ_DATA ( A , B , C , D , 100 )
```

```
! Calculate E=A+B
DO i = 1 , SIZE
  E(i) = A(i) + B(i)
ENDDO
```

```
! Calculate F=C*D
DO i = 1 , SIZE
  F(i) = C(i) * D(i)
ENDDO
```

```
! Write results
CALL WRITE_DATA( E , F , 100 )
```

We'll ignore this for
now...

A shared memory approach

- **Split the function across the threads**
 - In the example we have two functions:
 $E=A+B$ and $F=C*D$
 - But we have 4 processors (threads) – two would be idle ☹️
- **So what we do is split the computation of each loop between the threads**
- **We need some new syntax to tell the compiler/computer what we want it to do**
 - OpenMP – compiler directive
 - For now we'll just use some descriptive text
- **We don't really care which processor/thread does which computations**
 - The shared memory means that each processor/thread can read/write to any array element

Shared memory program

```
INTEGER, PARAMETER      :: SIZE=100
REAL, DIMENSION (SIZE) :: A,B,C,D,E,F
INTEGER                 :: i

! Read arrays A,B,C,D from a disk
CALL READ_DATA ( A , B , C , D , 100 )

! Calculate E=A+B and F=C*D
! (Merged loops to fit onto slide!)
! OpenMP : Distribute loop over NPROC threads
! OpenMP : Private variables : i
DO i = 1 , SIZE
  E(i) = A(i) + B(i)
  F(i) = C(i) * D(i)
ENDDO

! Write results
CALL WRITE_DATA( E , F , 100 )
```

This is easy on a shared memory machine as all threads can read/write to the whole of each array

Directives

- **Usually before a loop**
- **Tells the computer**
 - **How many threads to split the iterations of the loop between**
 - **Any variables which are “private” (default is that variables are “shared”)**
 - **“private” – each thread has an independent version of the variable**
 - **“shared” – all threads can read/write the same variable**
 - **The loop index must be private - each thread must have its own independent loop index so that it can keep track of what it’s doing**
 - **Optionally some tips on how to split the iterations of the loop between threads**

How to think about it

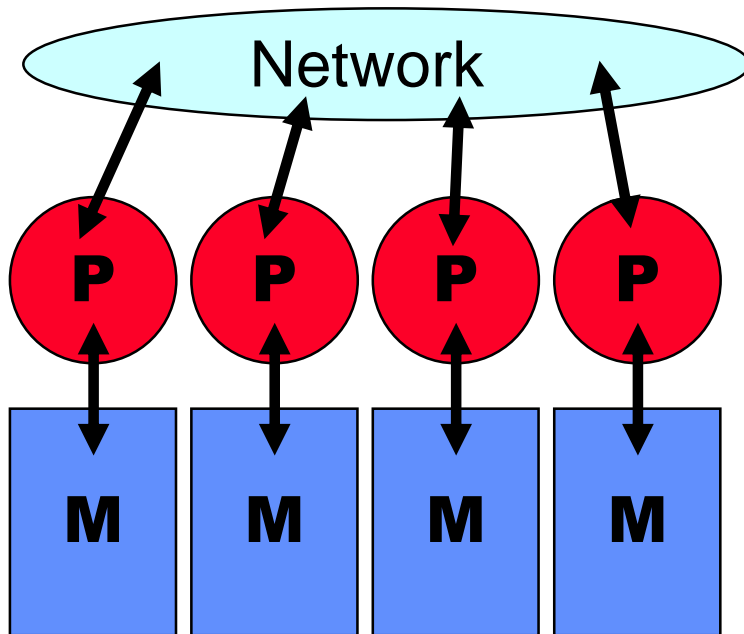
- **The program runs on a single processor P1 – as a single thread.**
- **Until...**
 - **It meets an OpenMP directive (typically before a loop)**
 - **This starts up the other processors (P2,P3,P4) – each running a single “thread”**
 - **Each thread takes a “chunk” of computations**
 - **This is repeated until all the computations are done**
 - **When the loop is finished (ENDDO) all the other processors (P2,P3,P4) go back to sleep, and execution continues on a single thread running on processor P1**

How to do it

- **Identify parts of the algorithm (typically loops) which can be split (parallelised) between processors**
- **Possibly rewrite algorithm to allow it to be (more efficiently) parallelised**
 - In our example we merged two loops – this can be more efficient than starting up all the parallel threads multiple times
- **For a given loop, identify any “private” variables**
 - eg. Loop index, partial sum etc.
- **Insert a directive telling the computer how to split the loop between processors**

Distributed memory programming

Each processor runs a single “task”



- **Split (decompose) the data**
 - “Data Parallelism”
- **Each processor/task works on a subset of the data**
- **Processors communicate over the network**
- **Advantages**
 - Easily scalable (assuming a good network)
- **Disadvantages**
 - Need to think about how to split our data
 - Need to think about dependencies and communications

A distributed memory approach [1]

- **Split (decompose) the data between the tasks**
- **We'll need to do something clever for input/output of the data**
 - We'll ignore this for now
- **Each task will compute its share of the full data set**
 - Shouldn't be any problem with load balance (if we decompose the data well)
- **Computation is easy in this example**
 - No dependencies between different elements of the arrays
 - If we had expressions like
$$A(i) = B(i-1) + B(i+1)$$
we would need to be a bit more clever...

A distributed memory approach [2]

- **Split the data between processors**

- Each processor will now have 25 (100 / 4) elements per array
- `REAL, DIMENSION (SIZE/4) :: A, B, C, D, E, F`

- **Processor 1**

- `A(1) .. A(25)` corresponds to
`A(1) .. A(25)` in the original (single processor code)

- **Processor 2**

- `A(1) .. A(25)` corresponds to
`A(26) .. A(50)` in the original (single processor code)

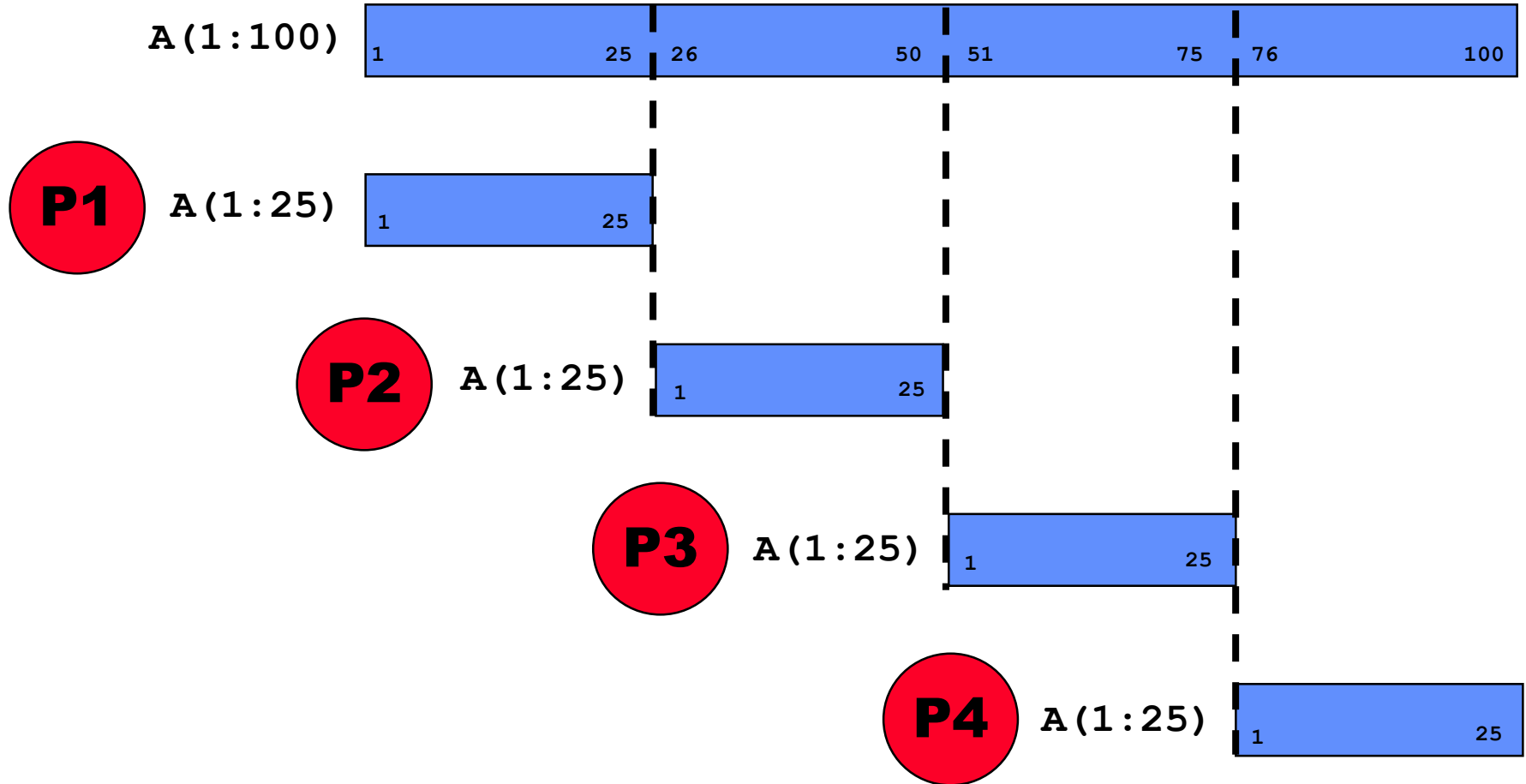
- **Processor 3**

- `A(1) .. A(25)` corresponds to
`A(51) .. A(75)` in the original (single processor code)

- **Processor 4**

- `A(1) .. A(25)` corresponds to
`A(76) .. A(100)` in the original (single processor code)

Distributed memory data mapping (array "A")



Distributed memory program

```
INTEGER, PARAMETER      :: NPROC=4  
INTEGER, PARAMETER      :: SIZE=100/NPROC  
REAL, DIMENSION (SIZE) :: A,B,C,D,E,F  
INTEGER                  :: i
```

```
! Read arrays A,B,C,D from a disk  
CALL READ_DATA ( A , B , C , D , 100 )
```

```
! Calculate E=A+B  
DO i = 1 , SIZE  
  E(i) = A(i) + B(i)  
ENDDO
```

```
! Calculate F=C*D  
DO i = 1 , SIZE  
  F(i) = C(i) * D(i)  
ENDDO
```

```
! Write results  
CALL WRITE_DATA( E , F , 100 )
```

We'll ignore this for now
...
But it is very important
and will need attention!

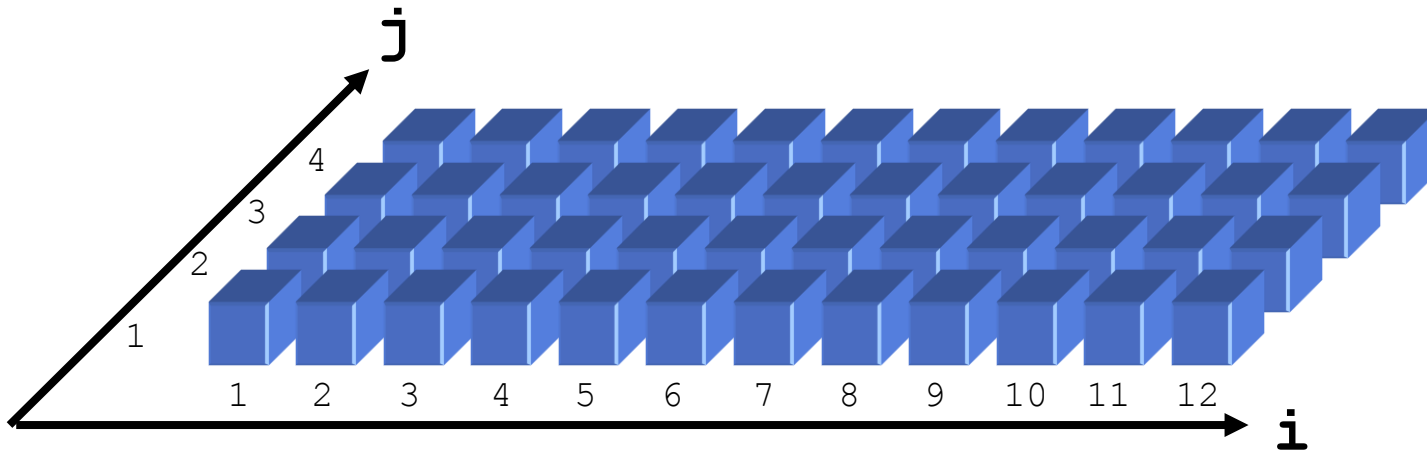
How to think about it

- **Each task runs its own copy of the program**
- **Each task's data is private to it**
- **Each task operates on a subset of the data**
- **Sometimes there are dependencies between data on different tasks**
 - **Tasks must explicitly communicate with one another**
 - **Message Passing key concepts**
 - **One task sends a message to one or more other tasks**
 - **These tasks receive the message**
 - **Synchronisation : All (or subset of) tasks wait until they have all reached a certain point**

How to do it

- **Think about how to split (decompose) the data**
 - Minimize dependencies (which array dimension should we decompose?)
 - Equal load balance (size of data and/or computation required)
 - May need different decompositions in different parts of the code
- **Add code to distribute input data across tasks**
 - And to collect when writing out
- **Watch out for end cases / edge conditions**
 - For example code which implements a wrap-around at the boundaries
 - First/Last item in a loop isn't necessarily the real "edge" of the data on every task
 - Maybe some extra logic required to check
- **Identify data dependencies**
 - Communicate data accordingly
 - Add code to transpose data if changing decomposition

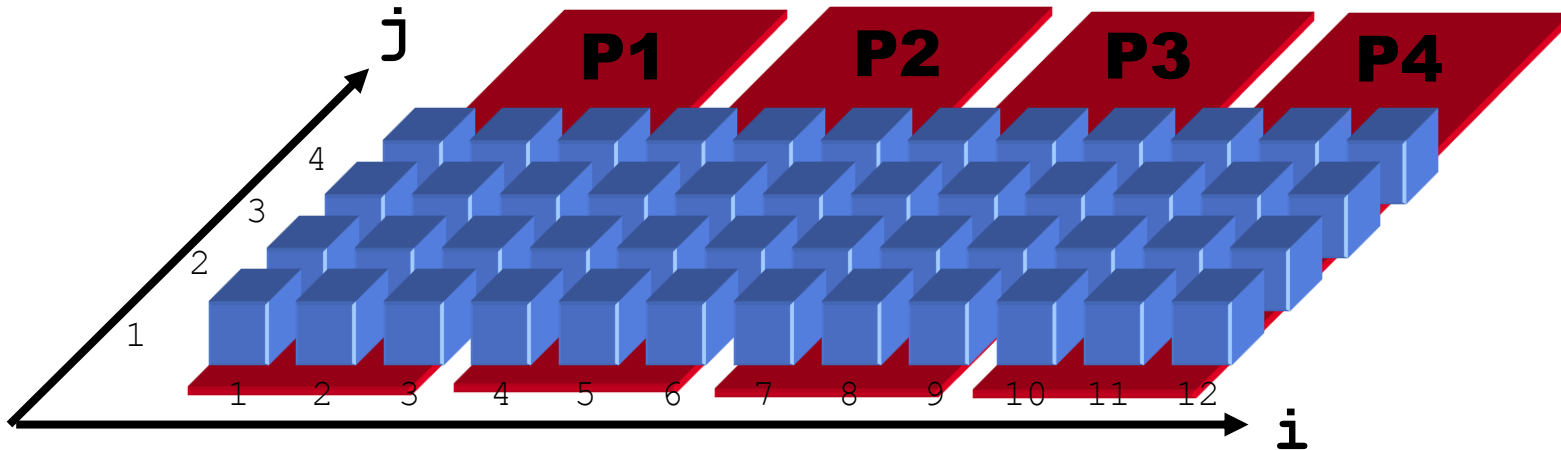
Decomposing Data [1]



```
REAL, DIMENSION (12,4) :: OLD,NEW  
  
DO j=1,4  
  DO i=2,11  
    NEW(i,j)=0.5*(OLD(i-1,j)+OLD(i+1,j))  
  ENDDO  
ENDDO
```

Decomposing Data [2]

- Let's think about decomposing the “i” dimension

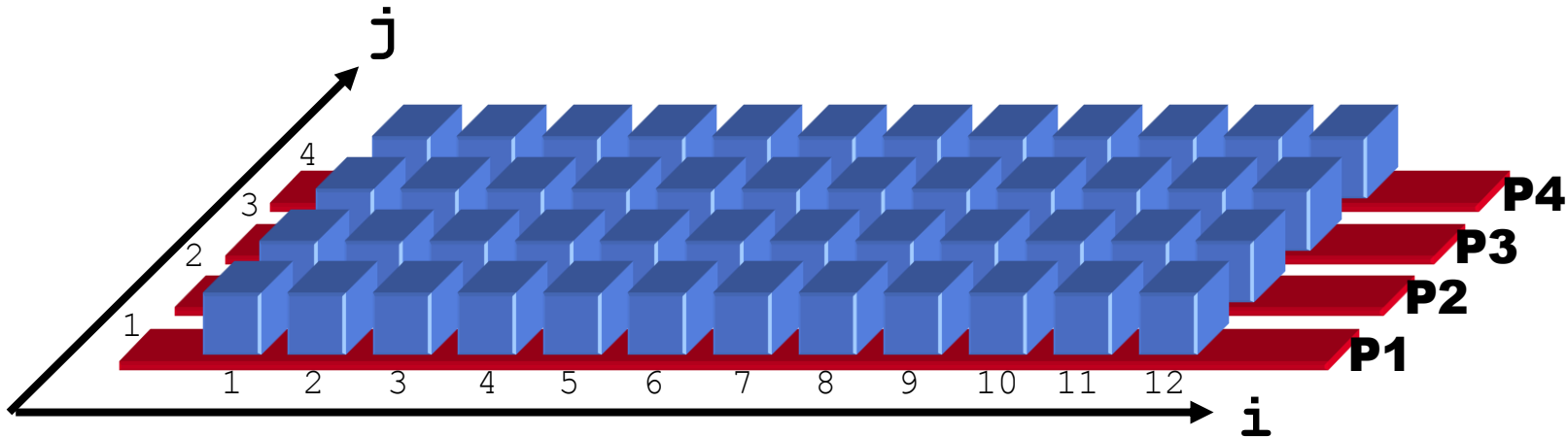


$$\text{NEW}(i, j) = 0.5 * (\text{OLD}(i-1, j) + \text{OLD}(i+1, j))$$

- How do we calculate element (3,1) – on P1?
 - We need element (2,1) which is on P1 – OK
 - And element (4,1) which is on P2 – Oh!
- So we need to do some message passing

Decomposing Data [3]

- Let's think about decomposing the “j” dimension

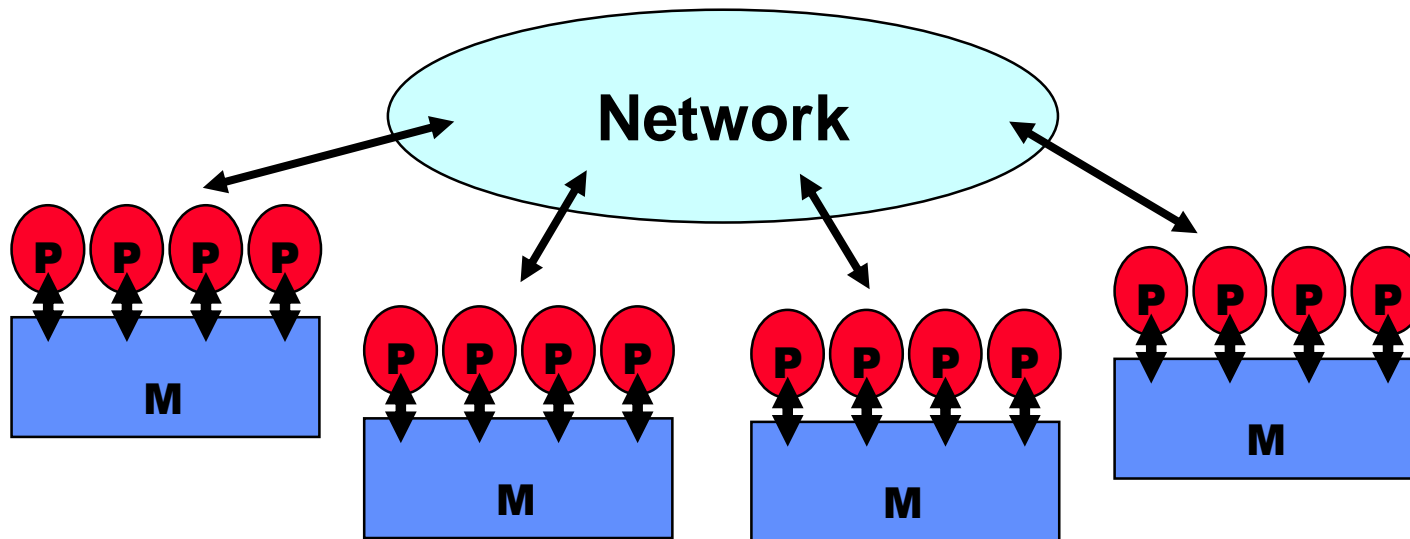


$$\text{NEW}(i, j) = 0.5 * (\text{OLD}(i-1, j) + \text{OLD}(i+1, j))$$

- Now no communication is needed
 - This is a much better decomposition for this problem
- Not so easy in real life!
 - Real codes often have dependencies in all dimensions
 - Minimize communication or transpose

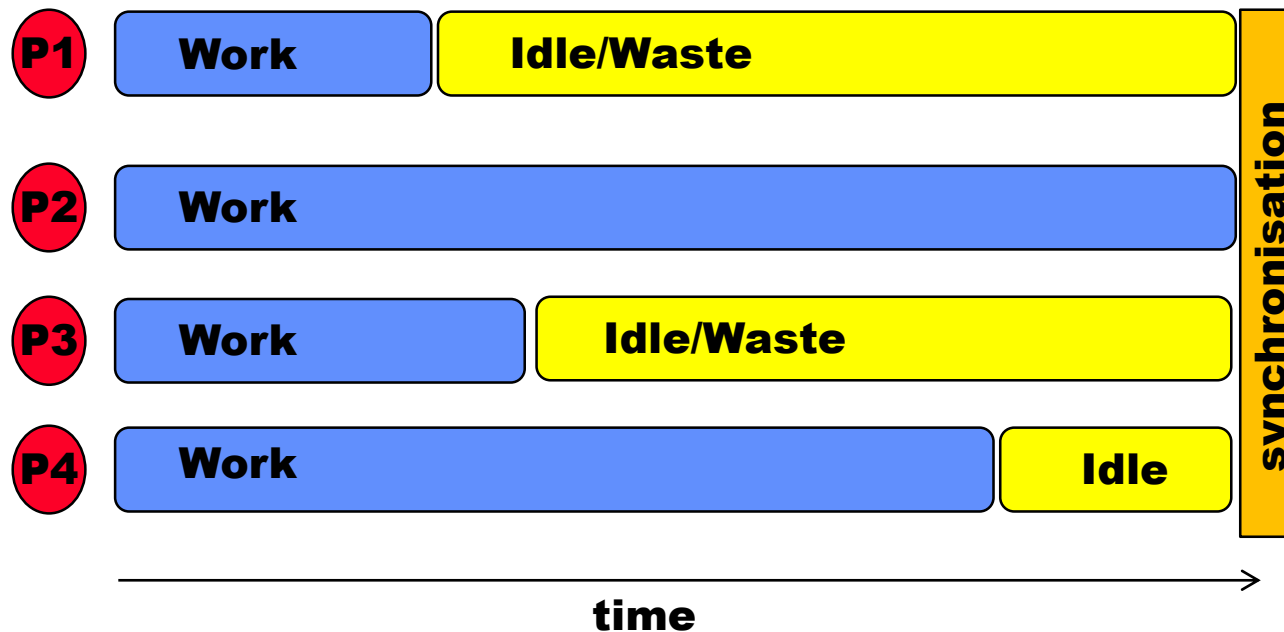
Shared & Distributed Memory programs

- **Many (most!) HPC systems combine architectures**
 - A node is often a shared memory computer with a number of processors and a single shared memory
 - Memory is distributed between nodes
- **Shared memory programming on a node**
- **Distributed memory programming between nodes**



Load Balancing

- Aim to have an equal computational load on each processor
 - Some processors sit idle waiting for others to complete some work
 - Maximum efficiency is gained when all processors are working



Causes of Load Imbalance

- **Different sized data on different processors**

- **Array dimensions and NPROC mean it's impossible to decompose data equally between processors**
 - **Change dimensions, or collapse loop:**
 $A(13, 7) \rightarrow A(13*7)$
- **Regular geographical decomposition may not have equal work points (eg. land/sea not uniformly distributed around globe)**
 - **Different decompositions required**

- **Different load for different data points**

- **Physical parameterisations such as convection, short wave radiation**

Improving Load Balance : Distributed Memory

- **Transpose data**

- **Change decomposition so as to minimize load imbalance**
- **Good solution if we can predict load per point (eg. land/sea)**

- **Implement a master/slave solution**

- **If we don't know the load per point**

```
IF (L_MASTER) THEN
  DO chunk=1,nchunks
    Wait for message from a slave
    Send DATA(offset(chunk)) to that slave
  ENDDO
  Send "Finished" message to all slaves
ELSEIF (L_SLAVE) THEN
  Send message to MASTER to say I'm ready to start
  WHILE ("Finished" message not received) DO
    Receive DATA(chunk_size) from MASTER processor
    Compute DATA
    Send DATA back to MASTER
  ENDWHILE
ENDIF
```


Improving Load Balance : Shared memory

- **Generally much easier**
- **In IFS we add an extra “artificial” dimension to arrays**
 - **Allows arrays to be easily handled using OpenMP**
- **So we write loops like this:**

```
REAL, DIMENSION (SIZE/NCHUNKS,NCHUNKS) :: A,B
! OpenMP : Distribute loop over NPROC (NPROC<=NCHUNKS) processors
! OpenMP : Private variables : chunk,i
DO chunk=1,NCHUNKS
  DO i=1,SIZE/NCHUNKS
    B(i,chunk)=Some_Complicated_Function(A(I,chunk))
  ENDDO
ENDDO
```

- **Make NCHUNKS >> NPROC**
 - **Load balancing will happen automatically**
- **Other performance benefits by tuning inner loop size**

Granularity

- **The ratio between computation and communication**
- **“Fine-grain” parallelism**
 - Small number of compute instructions between synchronisations
 - Reduces the changes needed to your algorithm
 - Can amplify load balance problems
 - Gives a high communications overhead
 - Eventually the communications time will swamp the computation time
 - Gets worse as you increase NPROC or decrease problem size
- **“Coarse-grain” parallelism**
 - Long computations between communications
 - Probably requires changes to your algorithm
 - May get “natural” load balancing with more work with different inherent load balance
- **Best granularity is a dependent on your algorithm and hardware**
- **Generally “coarse-grain” improves scalability**

Steps to parallelisation (1)

- **Identify parts of the program that can be executed in parallel**
- **Requires a thorough understanding of the algorithm**
- **Exploit any inherent parallelism which may exist**
- **Expose parallelism by**
 - **Re-ordering the algorithm**
 - **Tweaking to remove dependencies**
 - **Complete reformulation to a new more parallel algorithm**
 - **Google is your friend!**
 - **You're unlikely to be the first person to try and parallelise a given algorithm!**

Steps to parallelisation (2)

- **Decompose the program**

- **Probably a combination of
 - **Data parallelism (hard!) for distributed memory**
 - **Functional parallelism (easier, hopefully!) for shared memory****
- **If you're likely to need more than a few 10's of processors to run your problem then a distributed memory solution will be required
 - **Shared memory parallelism can be added as a second step, and can be added to individual parts of the algorithm in stages****
- **Identify the key data structures and data dependencies and how best to decompose them**

Steps to parallelisation (3)

- **Code development**

- **Parallelisation may be influenced by your machine's architecture**
 - **But try to have a flexible design – you won't use this machine for ever!**
- **Decompose key data structures**
- **Add new data structures to describe and control the decomposition (eg. offsets, mapping to/from global data, neighbour identification)**
- **Identify data dependencies and add the necessary communications**

- **And finally, the fun bit : CAT & DOG**

- **Compile And Test**
- **Debug, Optimise and Google!**

Some questions to think about...

- **Which do you think is easier to understand?**
 - **Distributed memory parallelism (message passing) or shared memory parallelism**
- **Which do you think is easier to implement?**
- **Which do you think might be easier to debug?**
 - **Can you imagine the kind of errors that you might make and how you might be able to find them?**
- **Do you think one may be more scalable than the other? Why?**
- **Why should we have to do all this work anyway. Why can't the compiler do it all for us?**