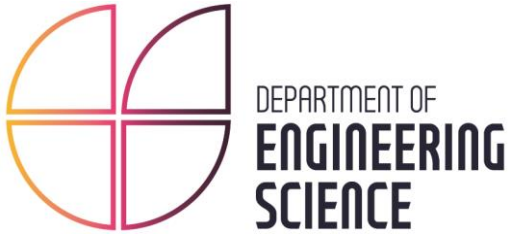


An Introduction to HPC and Scientific Computing

Lecture six: How to multi-task on CPUs using OpenMP.

Karel Adámek

Oxford e-Research Centre,
Department of Engineering Science

Thanks to Ian Bush for the materials

Overview

In this lecture we will learn about:

- The Basic Idea Behind Parallelism
- What is OpenMP
- How to divide work amongst threads
- How to synchronise threads
- How to go about parallelising a simple program with OpenMP

Parallelism – How do we split up the work?

Scenario: The application takes too long to run!

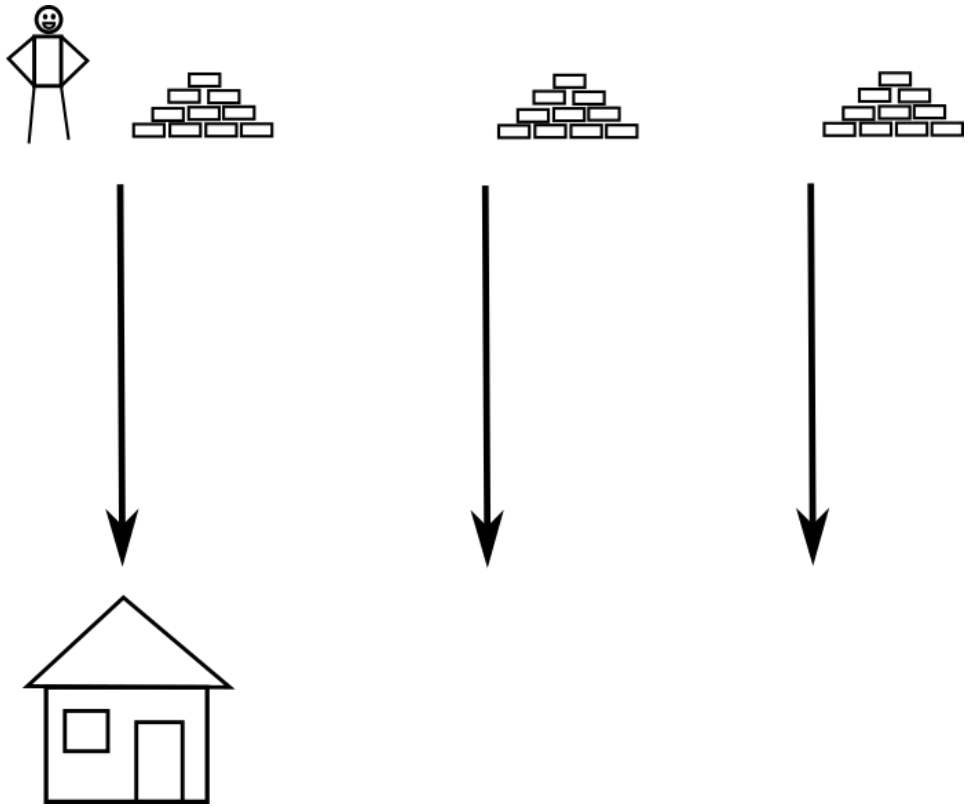
Parallel processing can shorten the run-time by:

- Parallelism in the workload (in algorithms and data) and
- Assign portions of the workload to different workers (processors)

Parallelism – How to do we split up the work?

Analogy:
Shorten the time it takes to build an entire row of houses by:

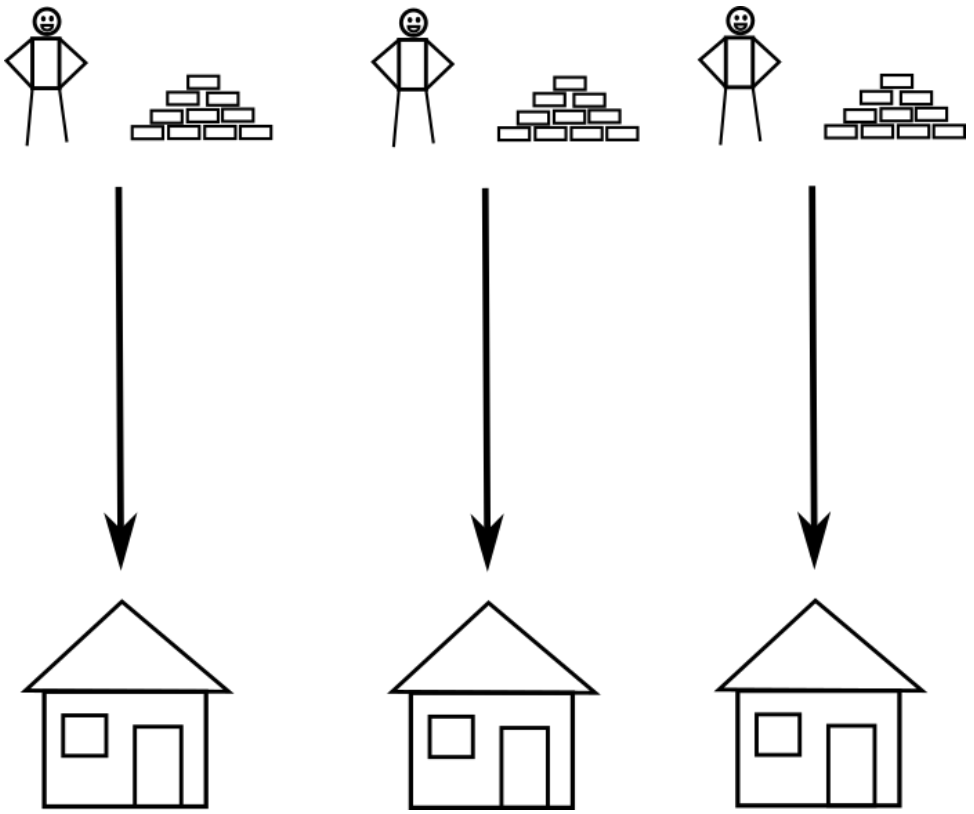
-
-



Parallelism – How to do we split up the work?

Analogy:
Shorten the time it takes to build an entire row of houses by:

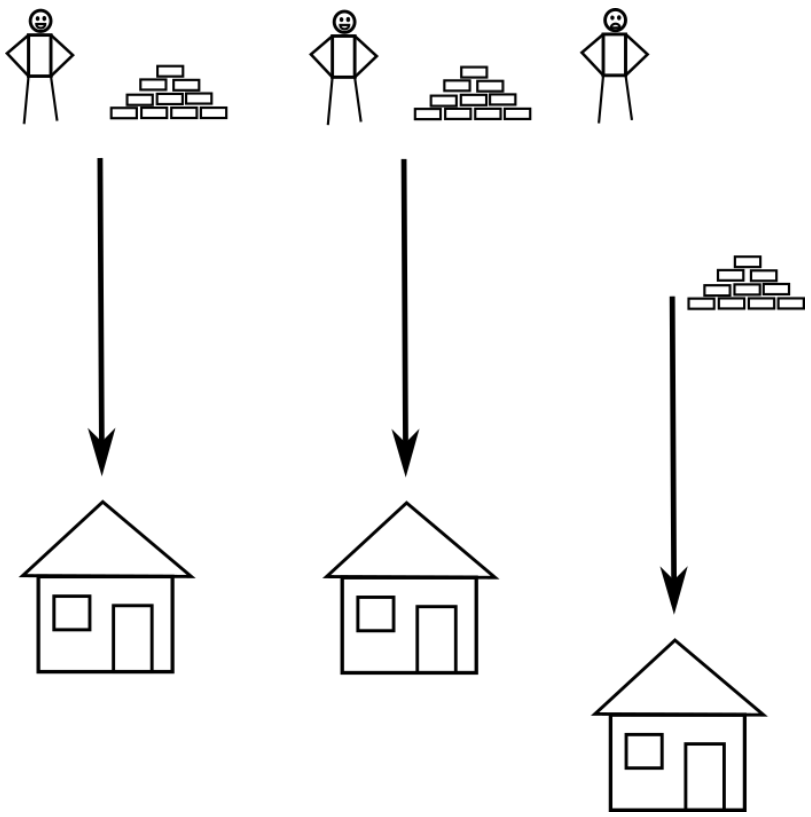
- dividing the work into parts that can be carried out in parallel and building multiple houses at the same time as one
-



Parallelism – How to do we split up the work?

Analogy:
Shorten the time it takes to build an entire row of houses by:

- dividing the work into parts that can be carried out in parallel and building multiple houses at the same time as one
- Oh no! Supply issues!

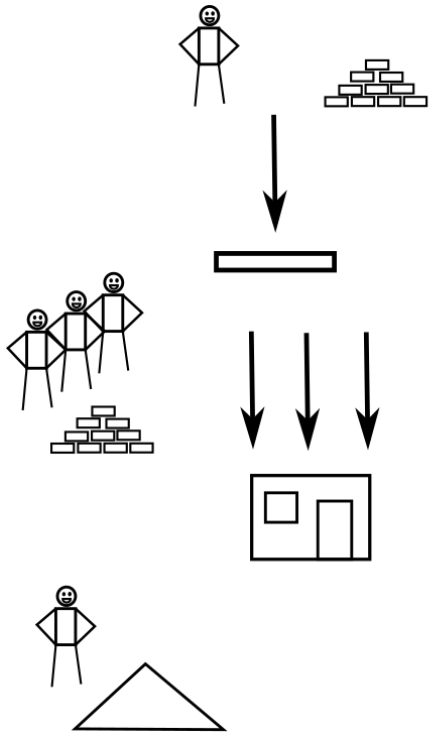


Parallelism – How to do we split up the work?

Perfect (linear) scaling is the holy grail:

- execution time is inversely proportional with the number of processors
- by analogy: halve the time by using twice the workers
- in practice, scaling is not linear but can be close enough

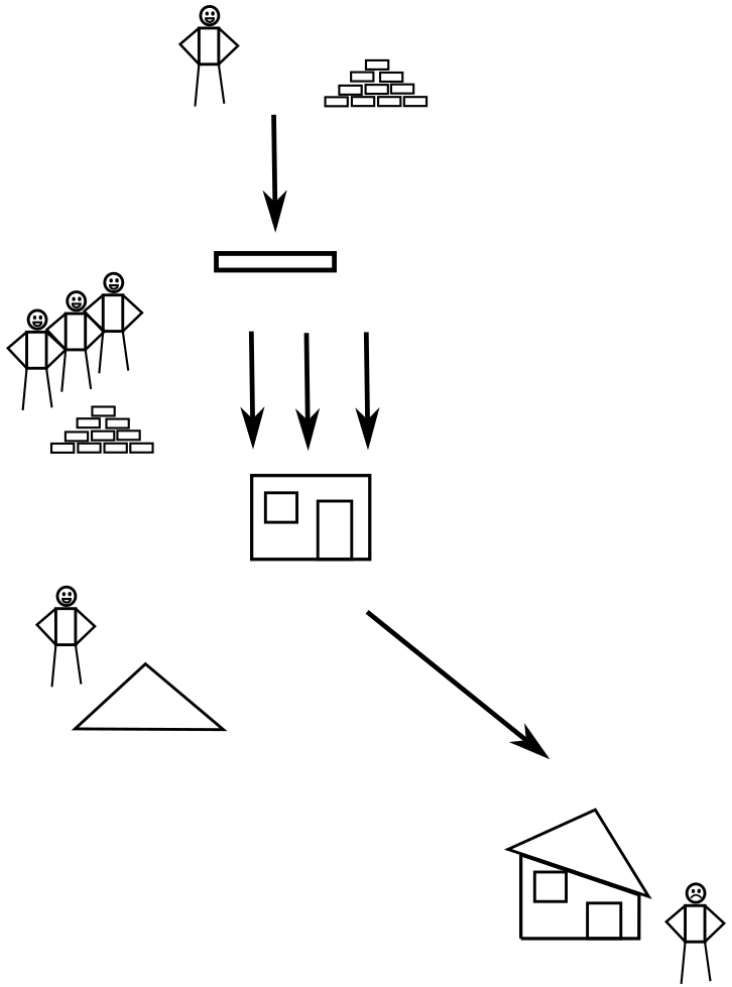
Parallelism – How to do we split up the work?



Division of work for one house:

- We can divide the work on a single house between teams
- Lay the foundations (one worker)
- Build walls (three workers)
- When to build a roof?

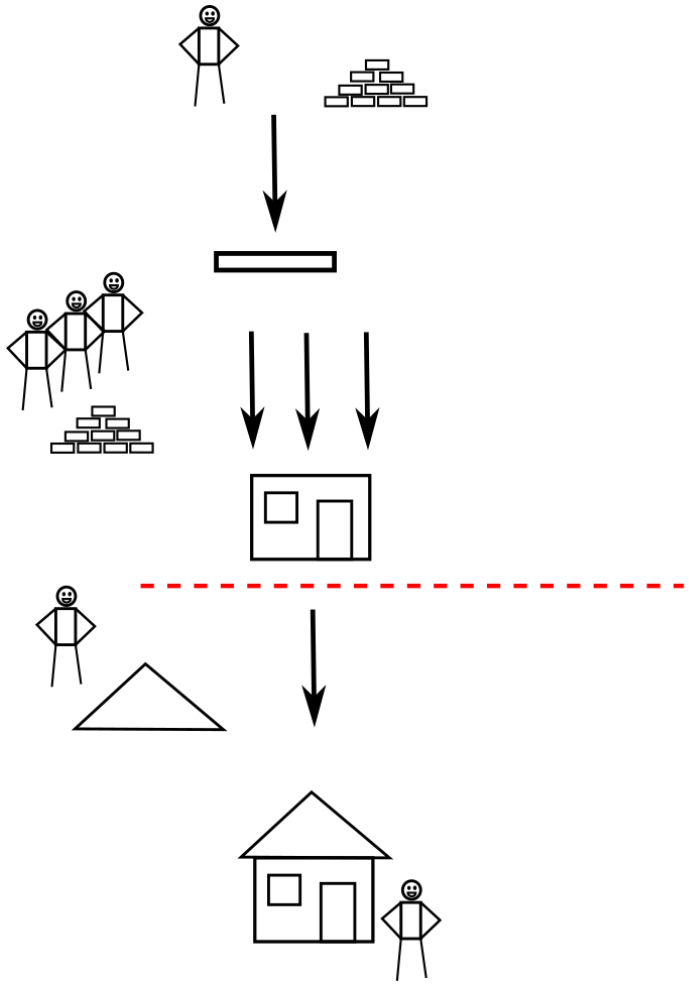
Parallelism – How to do we split up the work?



Division of work for one house:

- We can divide the work on a single house between teams
- Lay the foundations (one worker)
- Build walls (three workers)
- When to build a roof? Whenever we want?

Parallelism – How to do we split up the work?



Division of work for one house:

- We can divide the work on a single house between teams
- Lay the foundations (one worker)
- Build walls (three workers)
- When to build a roof? Whenever we want? NO!
- After walls are build!

Parallelism – A maths example

- Consider adding two vectors:

$$a_i = b_i + c_i$$

- The operations required for each value of *i* are independent of any other value
- Thus for parallelism we require *independent operations*
- We can perform the operations *in any order*
- We also require *independent data*
  - Consider the apparently very similar $a_i = b_i + a_{i-1}$
  - Can't calculate a_i until has been a_{i-1} calculated

Parallelism – Communication and Synchronisation

- Now consider a dot product

$$a = \sum b_i \times c_i$$

- Could give a subset of the indices to each process and form the local contribution to a
- Sum together all the local contributions for final answer
- Typical way parallelism work inside real codes (Divide-and-conquer)
  - Independent iterations
  - Coordinate local results into a global one
  - If one finishes before the others it will have to wait for them

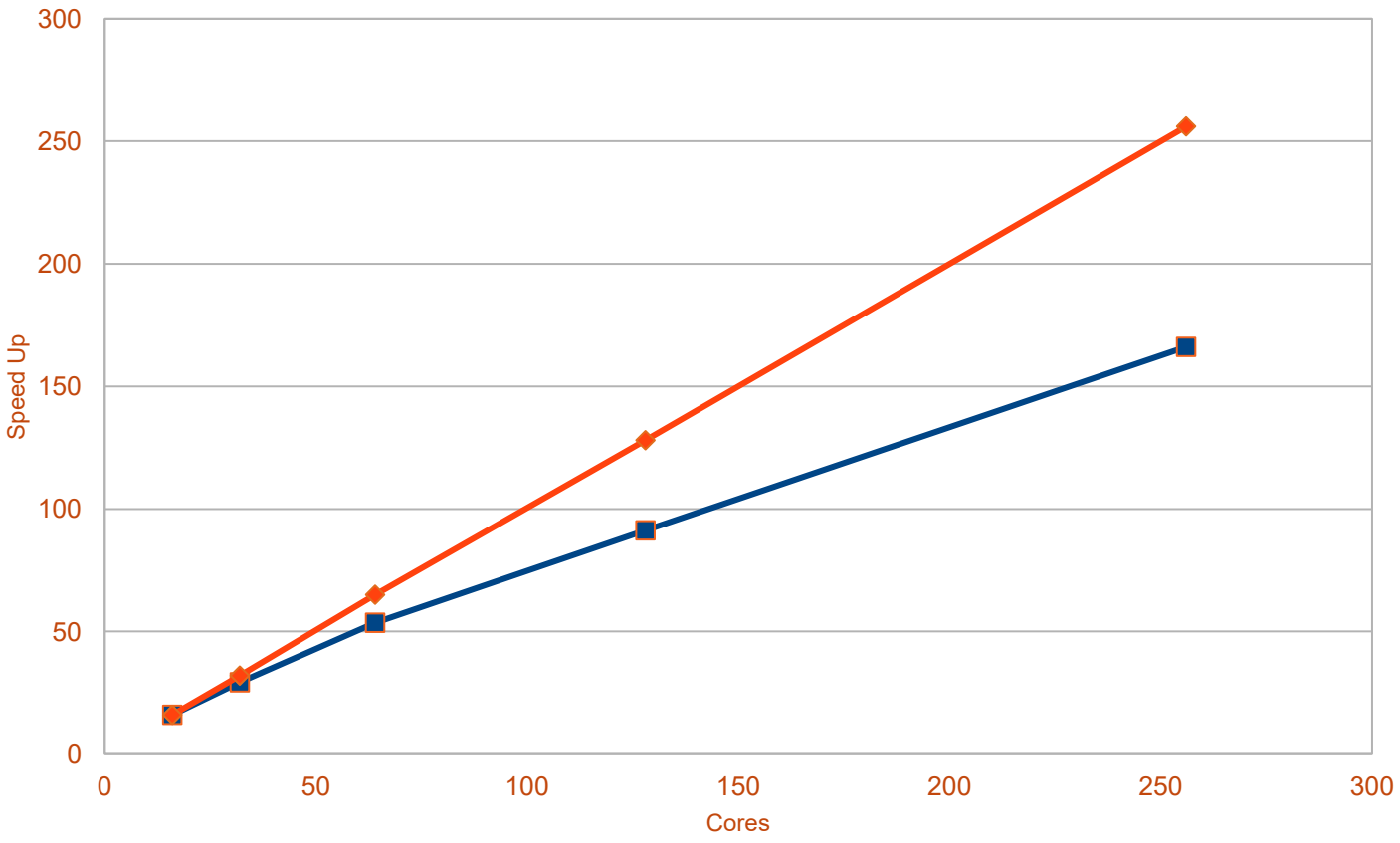
Parallel Scaling

- Consider the time to solution for our dot product.
- Vectors of order N
- P threads or processes
- The time to compute the local sum will be proportional to N/P
  - A **decreasing** function of the number of threads
- The time to synchronise is more complicated
  - But it will be an **increasing** function of the number of threads and independent of N
  - A deeper analysis shows a suitable model is log(P)
- Thus our model is:
- $$t(P) = \alpha N/P + \beta \log(P)$$

- Thus parallelism is the most efficient
  - For large N
  - For small P
  - Too many threads may cause your program to slow down!

Measuring Scalability – Speed Up

S = t(1)/t(P)

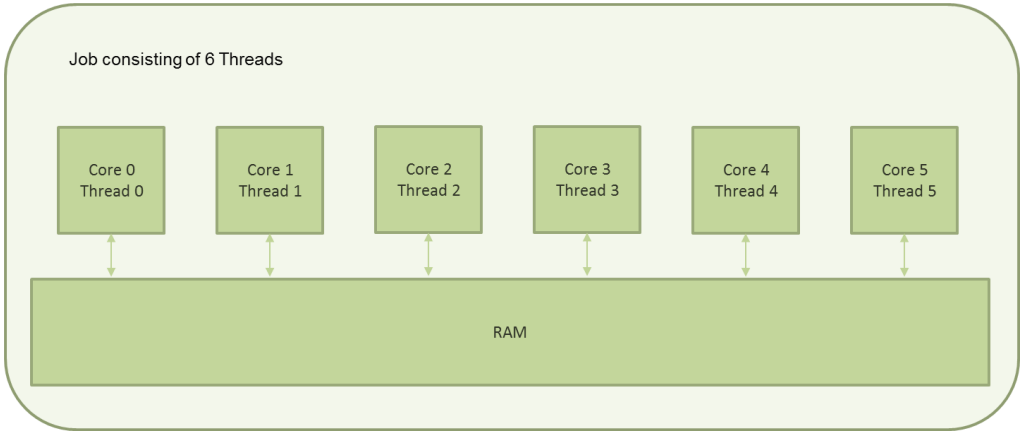


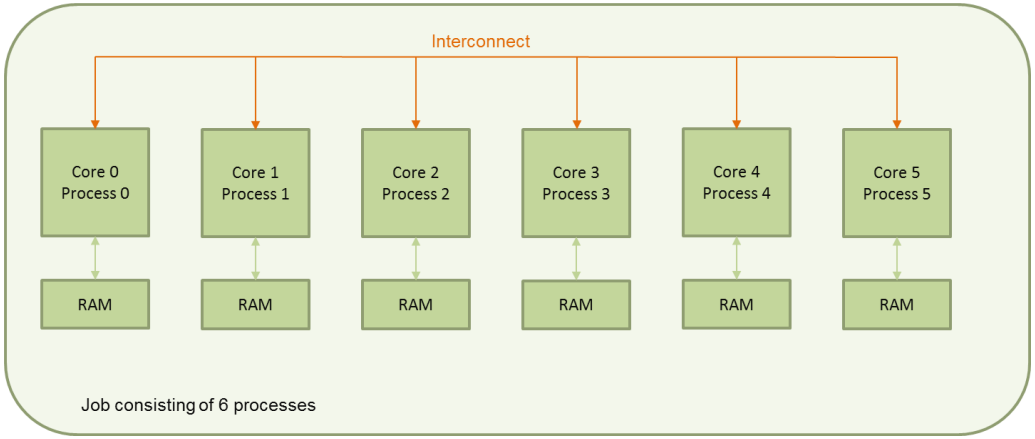
Further reading:

- Amdahl's law
- Gustafson's law

Parallelising Scientific Programs

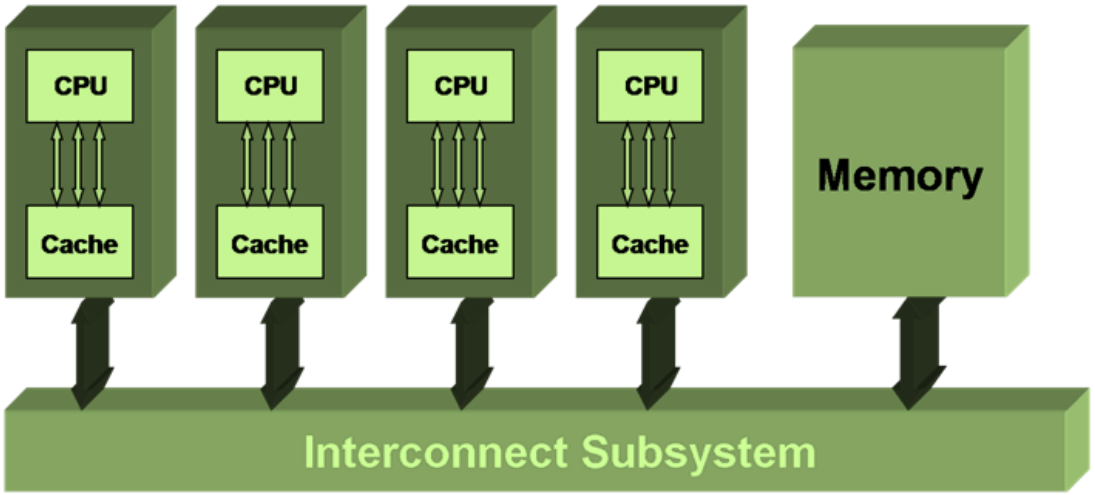
- **Shared memory programming**
  - Each core can see all of the memory
  - This is generally done via *OpenMP*
  - Generally limited to a single node
- **Distributed memory programming**
  - Each core can only see its memory
  - Have to use the interconnect
  - Can use an unlimited number of cores
  - Generally done via *MPI*
  - We don't cover it here but there are free ARC and ARCHER courses
- Hybrids of the two are possible and are used





Shared memory model

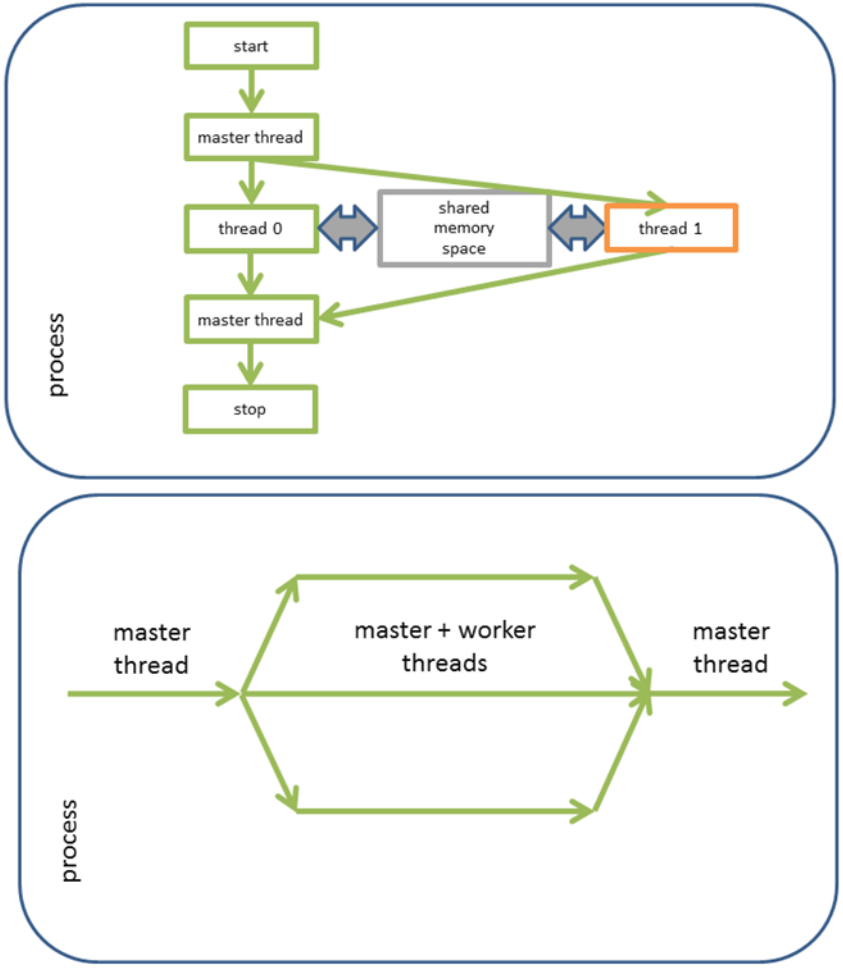
- Represents computing on a multi-processor architecture which links multiple (identical) CPUs to a single unified main memory;
- Uses a shared-memory programming model, with data that can be shared between processors
- Also called SMP, Symmetric Multi-Processor systems (for historical reasons)



Processes and Threads

In shared-memory computing:

- An “application” is a single process – with many threads
- The “process” is the “master thread”
- Master thread can spawn/destroy other threads (fork/join strategy)



Thread programming

- Explicit, low-level thread programming:
  - start several threads and explicitly tell each what to do
  - tedious, difficult to write and difficult to maintain
  - standards not universal (POSIX pthreads, Windows threads)
  - may be suited to task parallelism
- High-level programming using the OpenMP API:
  - instruct the compiler what can be done in parallel
  - let the compiler do the tedious stuff
  - threads are generated at runtime and scheduled by the OS
  - "standard" supported by a large number of compilers and operating systems
  - by design, suited to data parallelism

OpenMP Status

- OpenMP specification is not a formal standard.

- Version 1.0 (1997)
- Version 2.0 (2002)
- Version 2.5 (2005)
- Version 3.0 (2008)
- Version 3.1 (2011)
- Version 4.0 (2013)
- Version 5.0 (2018)
- Version 5.1 (2020)
- Version 5.2 (2021)

It is live and well

The OpenMP API

- OpenMP is
  - An API for portable and scalable shared memory programming
  - Defined and supported by a group of major vendors
  - Supported by most compilers for Fortran, C, C++
- OpenMP programming is
  - Standardised parallelism (fine-grained – loops, coarse-grain – regions)
  - If we're careful, usable serial code
- OpenMP API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables

OpenMP “Hello World” in C

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main( void ) {
  int my_thread;

  /* Scope the variables and create the threads */
  #pragma omp parallel default( none ) private( my_thread )
  {
    my_thread = omp_get_thread_num();
    printf( "Hello from thread %d\n", my_thread );
  }
  return EXIT_SUCCESS;

}
```

OpenMP “Hello World” in C

```
$ gcc -fopenmp -O -std=c89 -Wall -Wextra -pedantic hello.c -o
hello
$ export OMP_NUM_THREADS=5
$ ./hello
Hello from thread 2
Hello from thread 4
Hello from thread 3
Hello from thread 0
Hello from thread 1
$ export OMP_NUM_THREADS=3
$ ./hello
Hello from thread 0
Hello from thread 2
Hello from thread 1
```

Running In Batch

```
#!/bin/bash

# set the number of nodes, which is always 1 for OpenMP codes
#SBATCH --nodes=1

# set number of cores per node -- grab the whole node (why?)
#SBATCH --ntasks-per-node=16

# set max wallclock time
#SBATCH --time=00:10:00

# set name of job
#SBATCH --job-name hello

# Set up the software environment
module purge
module load intel-compilers

# set the number of threads we will use
export OMP_NUM_THREADS=5

# run the OpenMP program
./my_prog
```

A More Complex Example

```
#include <stdlib.h>
#define N 50

int main( void ) {
  double a[ N ], b[ N ], c[ N ];
  int i;
  for( i = 0; i < N; i++ ) {
    b[ i ] = ( (double) rand() ) / RAND_MAX;
    c[ i ] = ( (double) rand() ) / RAND_MAX;
  }

/* Scope the variables for the parallel region and create the
threads */
#pragma omp parallel default( none ) shared( a, b, c ) private( i )
  {
    /* A work sharing directive */
    #pragma omp for
    for( i = 0; i < N; i++ ) {
      a[ i ] = b[ i ] + c [ i ];
    }
  }
  return EXIT_SUCCESS;
}
```

Scoping and Work Sharing

There are two main steps

1. Create the threads with a parallel directive and at that point “scope” the variables.
   - `default none` forces you to scope all the variables – USE THIS.
   - Variables can be one of
     - `Shared` – each thread can access the variable
     - `Private` – each thread has its own, unique copy
   - Note thread creation is *expensive.*

2. Once the threads are created you split the work up between them with a “work sharing” directive
   - Without a work sharing directive all threads will do exactly the same thing on the same data!
   - You can have as many work sharing constructs as you want within a parallel region
   - As thread creation is expensive exploit this – don’t create threads every time you need to share work, reuse the threads as many times as you can!

Problems Of Sharing Data

- Shared variables makes writing parallel code easy
- But what happens if multiple threads are trying to update (the same part of) a shared variable at the same time?
  - The result is not well defined!
  - And will probably vary from run to run!
- This is called a “race condition” and OpenMP does NOTHING by default to protect you from them
- If multiple threads need to access a shared variable you will need to synchronize the threads somehow
- Rule of thumb: **If a shared variable is on the left hand side of an = sign it is time to stop and think about thread synchronisation**
- Race conditions are very easy to write and very hard to debug! Be careful

Synchronising Threads

- So how can we synchronise threads?

- OpenMP provides a number of mechanisms. Here we will cover
  - Barrier: a given thread can only proceed once all threads have reached the barrier
    - **Important**: By default there is an implicit barrier at the end of each worksharing construct
  - Critical region: Only 1 thread can be executing codes within a critical region. All other threads must wait at the start until the thread executing the region has exited it
  - Reduction: Combining multiple private values into a single shared value

Summary: OpenMP Directives So Far

parallel region constructs	blocks of code executed by all threads
	(most data shared by default)
		`#pragma omp parallel (C,C++)`
		`!$omp parallel (Fortran)`
		• shared data are visible to all threads
		• private data is local to thread and invisible outside, created on entry to scope, destroyed on exit

work-sharing constructs (within a parallel region)	parallel DO loops (distribute iterations over active threads)	`#pragma omp for` `!$omp do`
critical section	protects action on the shared variables (only one thread at a time allowed)	`#pragma omp critical` `!$omp critical`

synchronisation	barriers	`#pragma omp barrier`
		`!$omp barrier`
		Threads can proceed only after all execute the barrier; implied at end of parallel region and loop (unless overridden by nowait)

Bringing It Together – A More Complex Example

Task:
Orthogonalise 2 NORMALIZED vectors **x** and **y** on 2 threads.

There are several problems to watch out for.

Thread 1

```
z = 0
for i = 1 to n/2
    z = z + x_i * y_i
end for
[ w = w + z ]
< barrier >
for i = 1 to n/2
 y_i = y_i - w * x_i
end for
```

Thread 2

```
z = 0
for i = n/2+1 to n
    z = z + x_i * y_i
end for
[ w = w + z ]
< barrier >
for i = n/2+1 to n
 y_i = y_i - w * x_i
end for
```

Problem
A thread may finish bef
other has updated **w**

This now works!

z private

OpenMP program example

Fortran

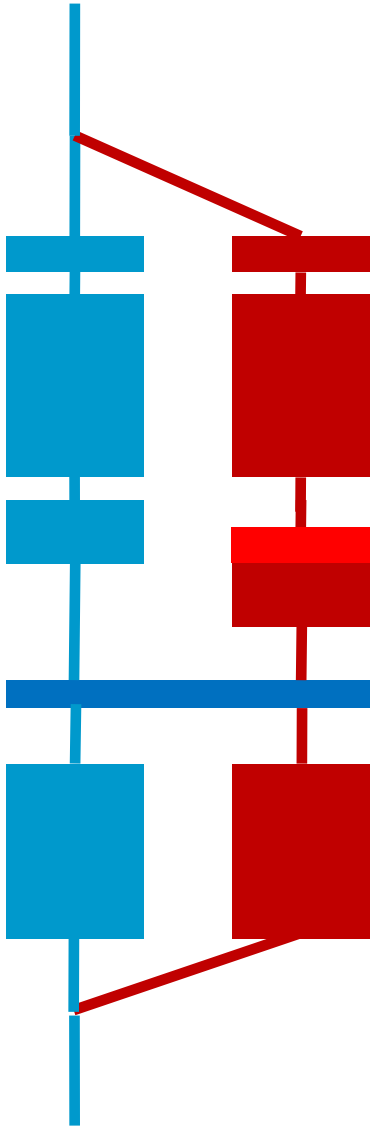
```
w = 0.0

!OMP PARALLEL SHARED(n,w,x,y), &
!OMP          PRIVATE (i,z)
   z = 0.0
!OMP DO
   DO  i = 1, n
      z = z + x(i)*y(i)
   END DO
!OMP END DO
!OMP CRITICAL
   w = w + z
!OMP END CRITICAL
!OMP BARRIER
!OMP DO
DO  i = 1, n
   y(i) = y(i) - w*x(i)
END DO
!OMP END DO
!END PARALLEL
```

C

```
w = 0.0;

#pragma omp parallel shared (n,w,x,y) \
                  private(i,z)
{
   z = 0.0;
   #pragma omp for
   for (i=0; i<n; i++) {
      z = z + x[i]*y[i];
   }
   #pragma omp critical
   {
      w = w + z;
   }
   #pragma omp barrier
   #pragma omp for
   for (i=0; i<n; i++) {
      y(i) = y(i) - w*x(i);
   }
}
```

OpenMP program example

```
w = 0.0;

#pragma omp parallel shared (n,w,x,y) \
                    private(i,z)
{
   z = 0.0;

   #pragma omp for
   for (i=0; i<n; i++) {
      z = z + x[i]*y[i];
   }

   #pragma omp critical
   {
      w = w + z;
   }

   #pragma omp barrier


   #pragma omp for
   for (i=0; i<n; i++) {
      y[i] = y[i] - w*x[i];
   }
}
```



Serial code

Spawn threads

Same code

Do loop
Each thread execute a different set of loop iterations

Only one thread at a time can execute this

Barrier
All threads synchronise

Do loop
Each thread execute a different set of loop iterations

Destroy threads

Serial code

Reductions

- In the above example we are adding up the value of the private z's across all threads to produce a shared value in w
  - So we are reducing many values to a single one via the + operator
- This is so common that reduction operations are supplied directly by OpenMP
- And if it is applicable reductions should be used rather than critical as it will be faster
  - Critical is all purpose, reduction is the precision tool
- Reductions can be applied to whole parallel regions or parallel do/for work share constructs
  - In the latter the reduction variable should be shared

```
w = 0.0;

#pragma omp parallel shared (n,w,x,y) private(i)
{

   #pragma omp for reduction(+:w)
   for (i=0; i<n; i++) {
      w = w + x[i]*y[i];
   }

   #pragma omp for
   for (i=0; i<n; i++) {
      y[i] = y[i] - w*x[i];
   }
}
```

Reduction Operators

- C/C++ Syntax:
  reduction (operator : list)
- Where
  - operator is one of: `+  *  -  &  ^  |  &&  ||  min  max`

OpenMP: best practices

- thread creation / destruction can be expensive, so threads should be re-used
  - maximise parallel loops (large loops has more opportunities for reuse of cached data)
  - avoid parallel regions in inner loops (aim for the coarsest data parallelism)
  - minimise the number of times parallel regions are entered/exited
  - example: parallel inner loop moved to parallel outer region

```
// initial
for (n=0;n<NN;n++)
  #pragma omp parallel private(m)
  #pragma omp for
  for (m=0;m<MM;m++)
```

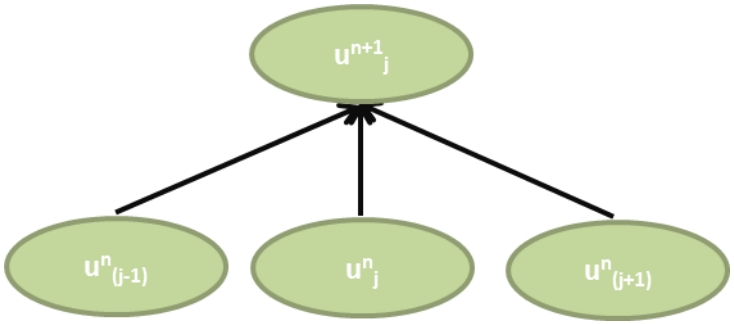
```
// improved
#pragma omp parallel private (n,m)
for (n=0;n<NN;n++)
  # pragma omp for
  for (m=0;m<MM;m++)
```

OpenMP: best practices

- Do not overuse synchronisation and explicit barriers
  - Remember every workshare construct by default has an implicit barrier on exit – this is normally enough
- Scope all your variables and use default(none)
- Think carefully about potential race conditions
  - Remember – a shared variable on the left hand side of an = should start alarm bells ringing

Example: 1D Heat Equation

- The **physics**: find the time-varying distribution of temperature along a rod, starting from an initial distribution, given the fixed temperature at the ends
- The **maths** (an initial value problem): find u(x,t) on the interval [0, 1] given
  - $\frac{\partial u}{\partial t} = k \frac{\partial^2 u}{\partial x^2}$
  - Boundary conditions u(0, t) = u(1, t) = 0
  - Initial conditions u(x,0) = $u_0(x)$.



- The **numerics** (a finite difference scheme):
  - sample the interval [0, 1] at equidistant points x_j=j*dx (j=0,…,J-1) and fixed time intervals t_n=n*dt, n>0 and find the values of all temperature samples u^n_j at the sample points
  - discretised equations give:

$$u^{(n+1)}_j = u^n_j + nu*(u^n_{(j+1)} - 2*u^n_j + u^n_{(j-1)})$$

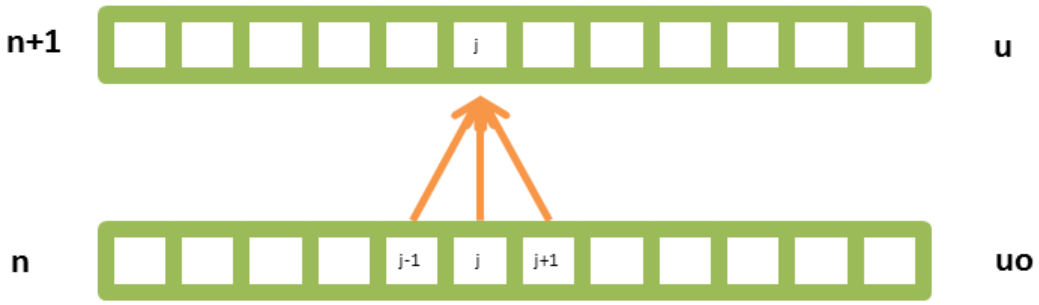
  - 3 point stencil

Example: 1D Heat Equation

- The core of the program is to use two vectors u and uo (u “old”).

At every time step,

- copy u into uo (not efficient but simple to understand)
- apply the finite difference scheme (the 3 point stencil)



- apply boundary conditions: $u_1=u_J=0$

Example: 1D Heat Equation

```
// boundary conditions
   u[0]   = 0.0; uo[0]   = 0.0;
   u[J-1] = 0.0; uo[J-1] = 0.0;

// time loop
  for (n=0; n<n_time_steps; n++) {

    // store solution
    for (j=1; j<J-1; j++) {
      uo[j] = u[j];
    }

    // finite difference scheme
    for (j=1; j<J-1; j++) {
      u[j] = uo[j] + nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
    }

}
```

Problem: 1D heat equation in Parallel – The Easy Way

```
// time loop
for (t=0; t<n_time_steps; t++) {
   // store solution



   for (j=1; j<n-1; j++) {
      uo[j] = u[j];
   }

   // finite difference scheme



   for (j=1; j<n-1; j++) {
      u[j] = uo[j] +
      nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
   }

}
```

```
// time loop
for (t=0; t<n_time_steps; t++) {
  // store solution
  #pragma omp parallel shared(n,u,uo,J) private(j)
  {
    #pragma omp for
    for (j=1; j<n-1; j++) {
      uo[j] = u[j];
    }
  }
  // finite difference scheme
  #pragma omp parallel shared(n,u,uo,nu,J) private(j)
  {
    #pragma omp for
    for (j=1; j<n-1; j++) {
      u[j] = uo[j] +
      nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
    }
  }
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Problem: 1D Heat Equation in Parallel – The Better Way

```
// time loop
for (t=0; t<n_time_steps; t++) {
  // store solution
  #pragma omp parallel shared(n,u,uo,J) private(j)
  {
    #pragma omp for
    for (j=1; j<n-1; j++) {
      uo[j] = u[j];
    }
  }
  // finite difference scheme
  #pragma omp parallel shared(n,u,uo,nu,J) private(j)
  {
    #pragma omp for
    for (j=1; j<n-1; j++) {
      u[j] = uo[j] +
      nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
    }
  }
}
```

```
// time loop
  # pragma omp parallel default( none ) \
          shared(n,n_time_steps,u,uo,nu,) \
          private(t,j)
  {
    for (t=0; t<n_time_steps; t++) {
      // store solution
      # pragma omp for
      for (j=1; j<n-1; j++) {
           uo[j] = u[j];
      }

      // finite difference scheme
      # pragma omp for
      for (j=1; j<n-1; j++) {
        u[j] = uo[j] +
        nu*(uo[j-1]-2.0*uo[j]+uo[j+1]);
      }
    }
  }
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

What have we learnt?

We have learnt

- The basic ideas behind parallelism
- How OpenMP implements the shared memory programming model
- How to create threads with #pragma omp parallel
- How to scope variables
- How to share the work between the threads with #pragma omp for
- How to synchronise the threads with barriers, critical regions and reductions
- How to parallelise a small but realistic program

Further learning

- There is more to OpenMP
  - More run-time library routines (times, locks, execution)
  - More work-sharing directives (sections, work share, task, …)
  - More synchronization directives (atomic, flush, …)
  - More environmental variables

- The best way to learn OpenMP is to go on a longer course run by the national supercomputing service, ARCHER2
  - https://www.archer2.ac.uk/training/
  - https://www.archer2.ac.uk/training/courses/211005-advanced-openmp/

Oxford ARC also offers trainings: https://www.arc.ox.ac.uk/training

In the next lecture…

Introduction to GPUs

- Hardware architecture
- Software tools
- NVIDIA libraries