THE UNIVERSITY *of York*

High Performance Computing - Introduction to Parallel Programming & OpenMP

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Amdahl’s Law and Scaling
- Auto-Parallelising Compilers
- Dependencies
- Data Parallel
- Thread Parallel
- Message Passing

What is Parallel Computing?

- Conventional (serial) computing has only a single CPU
  - Hence there is a single logical sequence of operations within a program
  - CPU executes instructions in order, only 1 operation in action at one time.
- Parallel computing uses many CPUs to produce the same result in less time, or to handle larger problem sizes.
  - Need to be divide up problem into different tasks to be handled by different CPUs.
- How effective is it? What kind of speed up can we get? Amdahl's Law gives some insights …

Amdahl's Law

- Amdahl's Law states:
  - Any problem/program can be broken up into inherently serial (*S*) and potentially parallel parts (*P*)
  - Then on a single CPU, the execution time is $T(1)=S+P$
  - But on a parallel machine with *N* processors, the execution time is $T(N)=S+P/N$
- Hence by using by parallelism we can make the program take less time.
  - The *parallel speedup* is given by $T(1)/T(N)$ and so for ideal ($S=0$) scaling this should be equal to *N*
  - The *parallel efficiency* is the parallel speedup/*N* i.e. parallel speedup per processor.

Amdahl's Law Example

- Consider a simple code, which contains 20% sequential code (e.g. the problem set-up phase) and 80% parallel code (e.g. the solution phase)
  - If on a given single CPU, the program takes 100 minutes, then with a perfectly parallel implementation on a perfectly scaling 4-CPU system the problem would take 20+80/4=40 minutes, i.e. a parallel speedup of 2.5
- If we use more processors then we quickly find execution time is dominated by the serial part:

N	**2**	**4**	**8**	**16**	**32**	**64**	**∞**
T(1)/T(N)	1.666	2.5	3.333	4	4.444	4.706	5

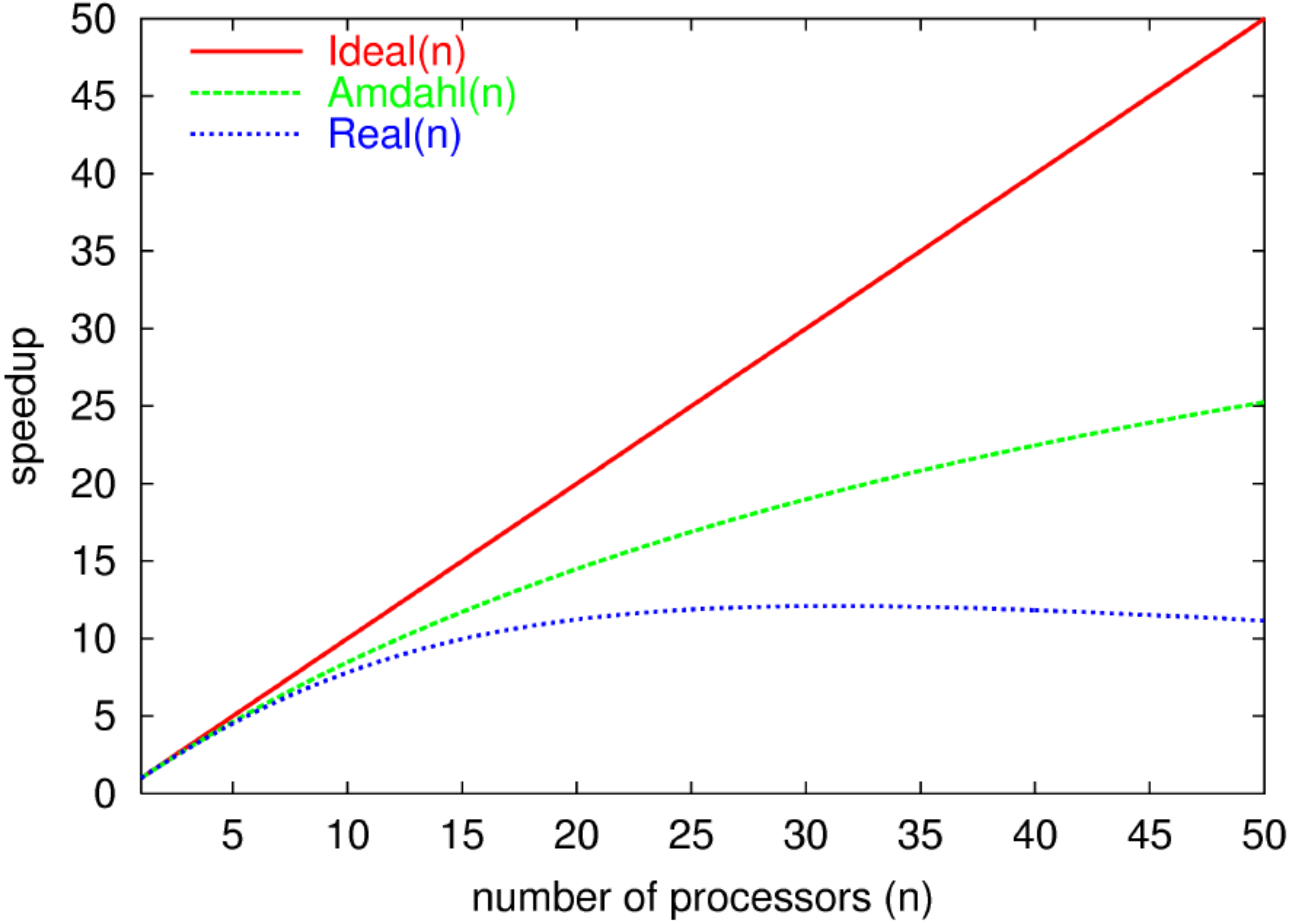
The Bad News of Amdahl’s Law

- On the face of it, Amdahl’s Law suggests that there is no point trying to build a massively parallel computers
  - When first discussed in 1967, surveys of typical codes suggested parallelism of 60-95% max
  - But programmers of the day were not thinking parallel – need different strategies when coding in parallel and hence can get this up to 99.9% or better with effort for some problems
  - And neglected the effect of problem scaling – Gustafson’s Law!

Gustafson’s Law

- “If the size of most problems is scaled up sufficiently, then any required efficiency can be achieved on any number of processors” (1987)
  - For example the serial part of a program (e.g. I/O) might scale linearly with the size of the problem whereas the parallel part (e.g. matrix multiplication or diagonalisation) might scale as square or cube of problem size
- Hence by making the problem bigger, you get a better parallel potential
  - Hence best use of parallel computers is to solve “bigger” problems, *not* “same size” problems in less time!

Scaling in Practice: P/S=49



Real-Life Scaling

- Amdahl's Law is only valid in ideal world
  - With P/S=49 it predicts a useful speedup with up to 50 CPUs
  - In real life get a maximum speedup with 25 and then gets slower! Why?
- Have neglected the cost of parallelisation – communication between processors:
  - The ideal time taken to transmit a message between processors is *time*= *latency* + *size/bandwidth*
  - But size of typical message ~ 1/N
  - Hence as N increases, message size decreases until size<latency*bandwidth whereupon latency dominates.
  - i.e. communications cost of extra processors outweighs computational gain.

Real-Life Amdahl's Law

- Hence, for any given code and parallel computer, there will be an optimum number of processors to use on any particular size problem.
- Hence need a modified form of Amdahl's Law:

  $T(N)=S+P/N+C_{BW}+N*C_L$
  - where C_{BW} is the cost of sending messages between processors due to the finite bandwidth,
  - and C_L is the cost due to the latency of each message.
- Hence the importance of the interconnect technology as discussed in earlier lectures

Auto-Parallelising Compilers

- Some compilers have switches which claim to auto-parallelise your code – job done?
  - As well as recognise instruction-level parallelism (pipelined and superscalar), compiler might also be able to recognise certain loops and do thread-level parallelism. Limited!
  - Normally only applicable to SMP paradigm.
  - OK as long as explicit enough for compiler to spot and *no dependencies*, e.g.

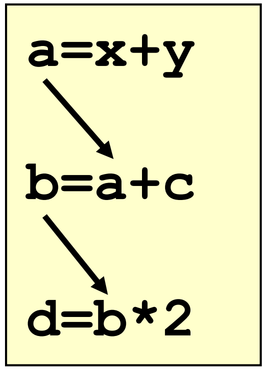
```
do i=1,16000
   a(i)=b(i)*two_pi
end do
```

$\longleftarrow$ No dependencies so loop can be computed in any order, parallelised and/or vectorised.

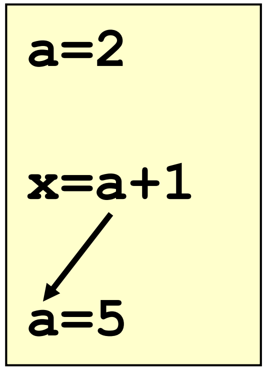
Dependencies

- If event *A* must be performed before event *B* then *B* is dependent upon *A*
- Dependencies inhibit auto-parallelisation
  - Data dependency is due to memory operations or calculation results
  - Control dependency is due to switches and branches
  - If these can be eliminated (by code-rearrangement) then potential for parallelism is increased.
  - Compiler instruction scheduling can do a lot of this for you, but if you write clear code then it helps a lot!
  - Most important in loops – a well-designed loop can expose a lot of parallelism, but one unresolvable dependency and it must all execute in serial! Costly …
  - Hence need to understand sources of dependencies and eliminate as much as possible to help the compiler!

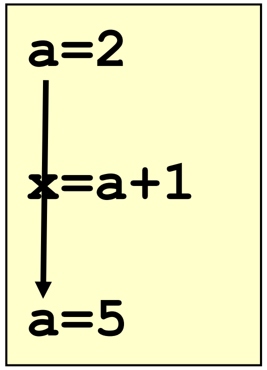
Types of Dependency



Flow dependency



Anti-dependency



Output dependency

- Any dependency must be one of these three kinds
- Arrow starts at source of dependency and ends at statement that must be delayed by the dependency
  - Second statement cannot start until first has completed

Loop Flow Dependency

```
do i=2,N
   a(i)=a(i-1)+b(i)
end do
```

Unroll
twice

```
do i=2,N,3
   a(i)   =a(i-1)+b(i)
   a(i+1)=a(i)   +b(i+1)
   a(i+2)=a(i+1)+b(i+2)
end do
```

- By mentally unrolling the loop see that have a *flow dependency*, where every iteration depends on previous one (backwards dependency)
  - e.g. solving equations by Gaussian elimination and back-substitution
  - Sometimes impossible to fix
  - Can be a function of the way the loop is written, e.g.

```
do i=2,N,2
   a(i)   =a(i-1)+b(i)
   a(i+1)=a(i-1)+b(i)+b(i+1)
end do
```

a(i)

Manually rewritten to expose more parallelism – slower on serial but might be useful on SMP?

Loop Anti-dependency

```
do i=1,N
   a(i)=b(i)   *e
   b(i)=a(i+2)*c
end do
```

Unroll twice

```
do i=1,N,3
   a(i)   =b(i)   *e
   a(i+1)=b(i+1)*e
   a(i+2)=b(i+2)*e

   b(i)   =a(i+2)*c
   b(i+1)=a(i+3)*c
   b(i+2)=a(i+4)*c
end do
```

Wrong! `b(i)` supposed to use "old" and not "new" value of `a(i+2)`!

- This is an example of an *anti-dependency* with a *dependency distance* of 2 – must make sure that the "later" instruction which uses `a(i+2)` completes before the "earlier" one redefines it.
- If `a` and `b` arrays are small then would be better to either save a copy of `a` or store all of `b` in a temporary array until loop completes.

Loop Output Dependency

```
do i=1,N
   a(i)=b(i)   *e
   b(i)=a(i+2)*c
end do
```

Unroll
twice

```
do i=1,N,3
   a(i)   =b(i)   *e
   b(i)   =a(i+2)*c
   a(i+1)=b(i+1)*e
   b(i+1)=a(i+3)*c
   a(i+2)=b(i+2)*e
   b(i+2)=a(i+4)*c
end do
```

- This is an alternative unrolling of the same loop which now has an *output dependency* of distance 2.
- Instructions in red cannot be performed at the same time.
- Can usually eliminate output dependency by adding temporary variables

Ambiguous References

- What if dependency is ambiguous?
  - E.g. if dependency distance is given by a variable ?

```
do i=1,N
   a(i)=b(i)*e
   b(i)=a(i+k)*c
end do
```

$k=0 \Rightarrow$ dependency within loop

$k<0 \Rightarrow$ loop flow dependency

$k>0 \Rightarrow$ loop anti-dependency

  - If the value of k is unknown then know nothing about the dependencies that may be present! This is an example of an *ambiguous reference*. Hence compiler has to treat this loop as serial
  - Also arises in array lookups, pointers, or (potential) memory aliasing, etc so AVOID these in HPC programs!

```
do i=1,N
   A(k(i)) = a(k(i)) + B(j(i))*c
end do
```

Taking Control of Parallel Computing

- What if we want more control than given by an auto-parallelising compiler?
  - E.g. If want a solution that will scale to more processors?
- Then need to start thinking in parallel
  - Need to consider devising parallel algorithms and strategies, e.g. *problem decomposition*
- Different strategies will be appropriate for different problems and different machine architectures
  - Want to keep all CPUs busy (*load balancing*) whilst minimising communications in order to get maximum parallel efficiency

Parallel Computer Architectures

Flynn's Taxonomy:

- Single Instruction Single Data (SISD)
  - i.e. a conventional serial computer.
  - The CPU executes one instruction on one piece of data at each step.
- Single Instruction Multiple Data (SIMD)
  - has the same instruction operating on different chunks of data, e.g. image processing (might have hardware support e.g. SSE instructions)
  - Vector architectures and multi-threaded CPUs. Can be programmed easily. Great with *data-parallel* languages e.g. HPF.
- Multiple Instruction Multiple Data (MIMD)
  - has several independent CPUs each (maybe) performing independent instructions, (maybe) operating on independent data.
  - Very flexible architecture, with most complex programming models.
  - Can be further divided according to memory model: shared or distributed Memory. Shared memory best programmed using *threads* (e.g. OpenMP) or message passing (e.g. MPI), distributed memory with *message passing*.

Data Parallel Programming

- Originates from vector computer days
  - Shared memory model
- Programming language has parallel intrinsic functions (e.g. HPF) that operates on data that is shared between all processors
- High level abstraction
  - data distribution and communication handled by compiler so easier to use, debug and port
  - BUT less flexible, limited applicability, harder to get good performance, very reliant on good compilers
  - Big hope of the mid-90's but seems to have died – brief resurgence in early 2000s when the Earth Simulator used HPF to shock everyone with very high %peak achieved …

Thread Parallel

- A popular paradigm for programming shared-memory machines is to use a conventional language with additional *compiler directives*
  - i.e. source lines that look like comments but can be understood by aware compilers
  - e.g. OpenMP with C or F90 – started off as SGI proprietary and then made open standard
- Threads add to existing process not separate forks
  - Share a common memory space for code & global data
  - Each thread has a private area for own local variables
  - Need to tell compiler which variables to share and which are private when starting new threads

Message Passing

- First open standard was PVM (parallel virtual machine) for distributed memory machines
  - Designed for heterogeneous system – now obsolete
- Superseded by MPI (message passing interface)
  - Designed originally for homogeneous system (but also heterogeneous) by many of the original authors of PVM – hence improved and simplified w.r.t. PVM
- Both implemented as calls in conventional languages (Fortran or C) to library so highly portable
- Very flexible, powerful and efficient
  - But requires programmer to take a lot of responsibility
  - See later lectures for details …

Thread Parallel with OpenMP

- Directives to specify parallel start and end
  - e.g. do loops in parallel
  - Can also specify critical regions (e.g. for I/O in serial)
- Need to handle loop dependencies explicitly
- Can specify which variables are shared (all threads have same value and same memory location) and which are private (each thread gets own copy)
  - Can specify how data is copied to privates at start/end of parallel sections – firstprivate, lastprivate, reduction, etc.
- Can specify static scheduling or dynamic scheduling
  - Static: fixed number of loop iterations per thread
  - Dynamic: each thread assign given size chunk of data and is assigned new chunk upon completion of task

Starting OpenMP

OpenMP functionality requires interfaces to library routines.

```
#include <omp.h>
```

C/C++

```
use omp_lib

use omp_lib_kinds
```

Fortran – have one or two `use` statements before the `implicit none`

Compile with:

GNU: `-fopenmp` for gcc/gfortran

PGI/Pathscale: `-mp`

Intel: `-openmp`

Controlling OpenMP

Behaviour of OpenMP depends on environment variables set on the *command line*, e.g.

```
[mijp1@will0w]$ export OMP_NUM_THREADS=2
```

Bash shell

`OMP_NUM_THREADS` sets the number of threads to use (e.g. set to number of CPUs or hyperthreads)

Can also specify nested parallelism using 1:4:2 syntax etc

Other useful OMP environment variables include:

`OMP_DYNAMIC=TRUE/FALSE` determines if the programmer is able to change the number of threads at run time.

`OMP_NESTED=TRUE/FALSE` if want to serialize inner parallelism

Plus some extra specialized OMP environment variables …

OpenMP Functions

```
integer :: ncpu, nthreads
logical :: dynamic

ncpu      = omp_get_num_procs()
dynamic   = omp_get_dynamic()
nthreads = omp_get_num_threads()
```

$\}$ Can be *set* as well as *get*

- `omp_get_num_procs` returns #cores available.
    - Reports 12 on a hyperthreaded hex-core i7
- `omp_get_num_threads` returns #threads in use
- Default values of `num_threads` and `dynamic` are taken from the appropriate environment variables.

Explicitly Parallel Regions

```
!$omp parallel private(my_thread)
my_thread = omp_get_thread_num()
print *,'Hello world from thread number',my_thread
!$omp end parallel
```

```
#pragma omp parallel private(my_thread)
{
  my_thread = omp_get_thread_num();
  printf("Hello world from thread number %f",my_thread);
}
```

Code within the region will be executed by all threads. Note that threads are numbered from zero.

OpenMP Example I (Fortran)

```
! vector addition
sum=0
!$OMP parallel do private(i) shared(a,b) reduction(+:sum)
do i=1,1000
   sum = sum + a(i) + b(i)
end do
!$OMP end parallel do
```

- `!$OMP` is *sentinel* that directs compiler to OpenMP
- `parallel do` defines start of parallel region where threads created from original master thread begin
- `end parallel do` defines end of parallel region where all threads apart from master thread suspended
- `private(i)` so each thread can work with its own values of loop index but `shared(a,b)` so all threads read same areas of memory
- `reduction(+:sum)` to combine the separate values at the end of the `parallel do` using '+' with original value

OpenMP Example I (C)

```
! vector addition
sum = 0;
#pragma omp parallel for private(i) shared(a,b)reduction(+:sum)
for (i=0;i<1000;i++){
    sum += a[i] + b[i];
}
```

- `#pragma omp` directs compiler to OpenMP
- `parallel for` defines start of parallel region where threads created from original master thread begin
- No '`end for`' needed as not necessary in C style.
- `private(i)` so each thread can work with its own values of loop index but `shared(a,b)` so all threads read same areas of memory
- `reduction(+:sum)` to combine the separate values at the end of the `parallel do` using '+' with original value

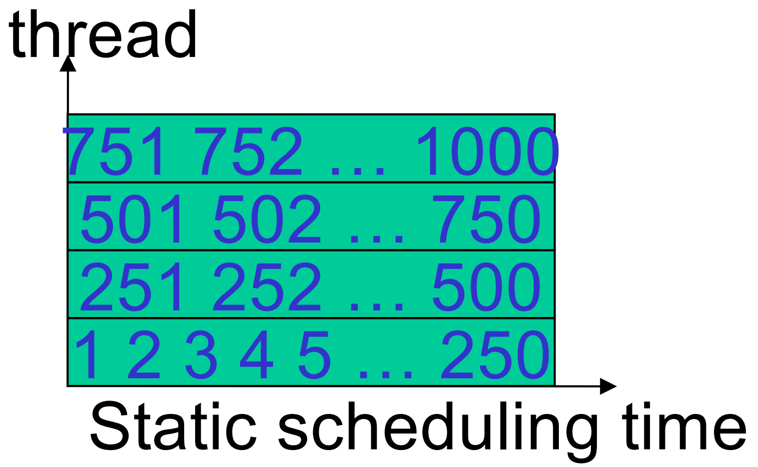
OpenMP parallel loops

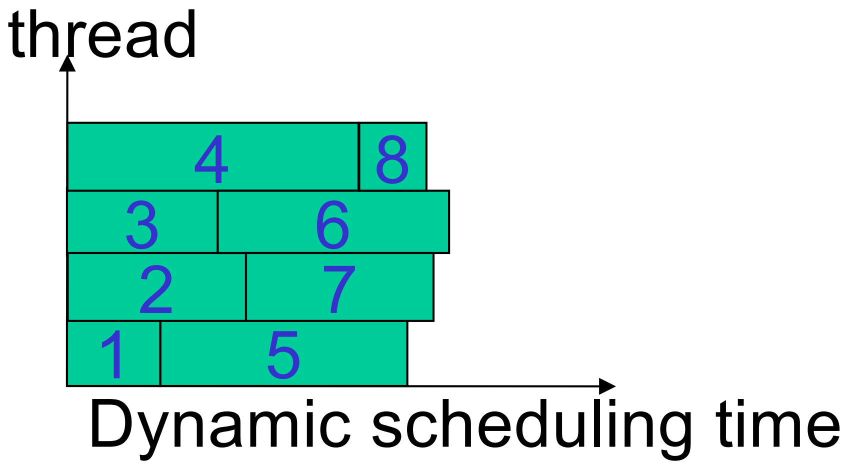
- Simplest way to get parallel speedup
- Restrictions on the loop:
  - OpenMP must know in advance #iterations
  - Hence cannot jump out of loop with break/exit commands but can use cycle/continue
  - Loop index is automatically private and not allowed to be modified inside loop
  - Loop index update must be a constant
  - Not a 'while' loop!
- Can change the division of iterations to threads by changing the `schedule` ...

OpenMP Example II

```
! MonteCarlo update
!$OMP parallel do private(i,ranval) schedule(dynamic)
do i=1,1000
   ranval=random_number(i)
   call MC_update(ranval)
end do
!$OMP end parallel do
```

- `private(i,ranval)` so each thread can work with its own values of loop index and random number and nothing is `shared`
- `schedule(dynamic)` as each iteration will take variable amounts of time to get load balancing





OpenMP Example III (Fortran)

```
! vector addition
sum=0
N=25
!$OMP parallel do default(none) private(i) &
!$OMP & shared(a,b) reduction(+:sum) if(N>100)
do i=1,N
   sum = sum + a(i) + b(i)
end do
!$OMP end parallel do
```

- `default(none)` requires all variables to be explicitly stated as private or shared etc
- `!$OMP &` is a F90 continuation line to split an over-long line (essential if >132 chars)
- `if(logical)` parallelizes following construct only if logical=true else code is serialised.

OpenMP Example III (C)

```
! vector addition
sum=0;
N=25;
#pragma omp parallel for default(none) private(i) \
        shared(a,b) reduction(+:sum) if(N>100)
{
   for (i=1;i<=N;i++)
        sum += a(i) + b(i);
}
```

- `default(none)` requires all variables to be explicitly stated as private or shared etc
- `#pragma` ihandled by preprocessor so just use \ to split line as many times as necessary
- `if(logical)` parallelizes following construct only if logical=true else code is serialised.

OpenMP Example IV

```
!$OMP PARALLEL default(none) private(i,me) shared(A,B)
!$ me=omp_get_thread_num()

!$OMP DO
  do i=1,N
     A(i)=0
     B(i)=10*i
  end do
!$OMP END DO

!$OMP DO
  do i=1,N
     A(i)=A(i)+B(i)
  end do
!$OMP END DO

!$ print *,'thread ',me,'A(N)=',A(N)

!$OMP END PARALLEL
```

NB DO not PARALLEL DO
as we want to share work
over the team of threads
created by outer
PARALLEL region

- `!$` means only compile if using OMP
- `!$OMP DO` as workshare threads created by outer `PARALLEL` and not make a new team of threads

More OpenMP

- Warning – there can be issues with timing
  - e.g. F95 `CPU_TIME` returns the total CPU time *not* the time per thread
- Best to use OpenMP routines
  - e.g. `OMP_GET_WTIME` function gives time per thread, with a timing resolution given by `OMP_GET_WTICK` function
- Much more functionality e.g.
  - Parallel sections construct, workshare construct, teams, tasks, accelerator support, GPU support, SIMD etc ...

Further Reading

- Chapter 6 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC Press (2011).

- Lots of tutorials and guides at http://openmp.org
- Nice tutorial/guide at https://computing.llnl.gov/tutorials/openMP