Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Parallel Programming

Lecture 2

István Reguly
reguly.istvan@itk.ppke.hu

So how do we parallelise an application?

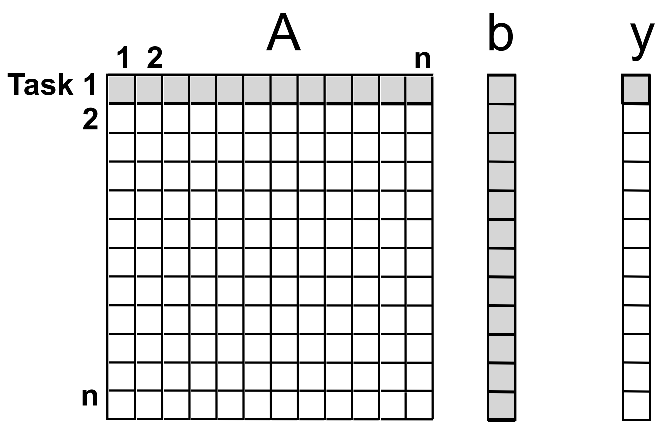
Understand the problem

- Find the hotspots/bottlenecks
  - Know where most of the time is spent, where real work is done
  - Identify the underlying algorithm, and try to parallelise it
- Can it even be parallelised?
  - Yes: calculate the potential energy for each of 10 thousand independent conformations of a molecule. When done, find the minimum energy conformation.
  - No: calculate the Fibonacci series by using the formula F(n) = F(n-1)+F(n-2)

Typical steps

- Identify what pieces of work can be done concurrently
- Partition concurrent work onto independent processors
- Coordinate accesses to shared data (avoid conflicts)
- Ensure proper order of work using synchronisation
- What is “typical”
  - In some cases, some of these steps are unnecessary
  - Mapping of work to processors can be done either manually or automatically

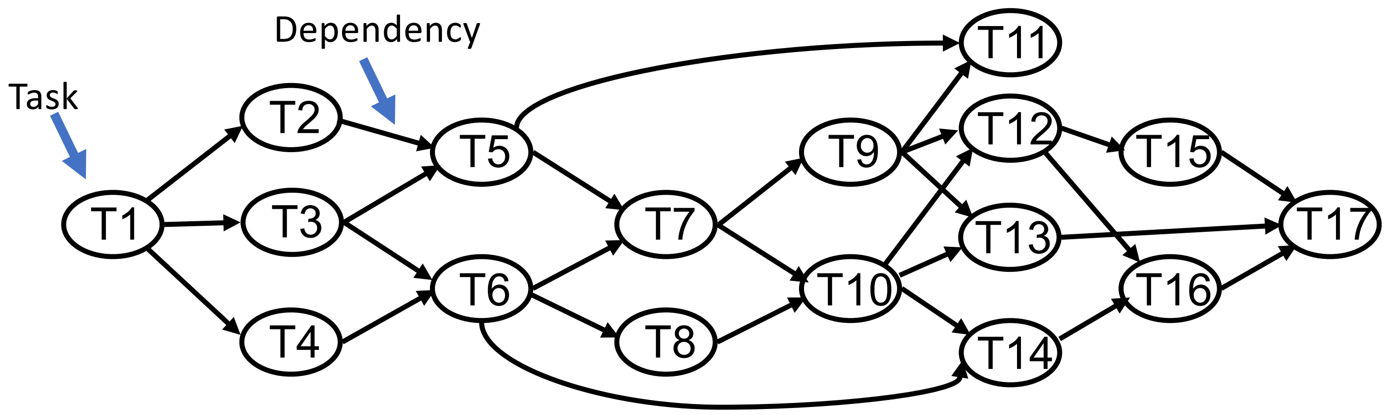
Example: Dense matrix-vector product



- Computing each element of the output **y** is independent
- Easy to decompose into tasks: one per element in y
- Properties:
  - Task size is uniform
  - No dependence between tasks
  - Tasks share b (read)

Creating tasks

- Divide work into a number of tasks, identify how they are interrelated
- Many different decompositions possible
- Tasks may be the same, different or varying sizes
- Conceptualise with a Directed Acyclic Graph (DAG)



Example: Database Query

- Consider the execution of the query:

MODEL = “CIVIC’ AND YEAR=2001 AND (COLOR=“GREEN” OR COLOR=“WHITE”)

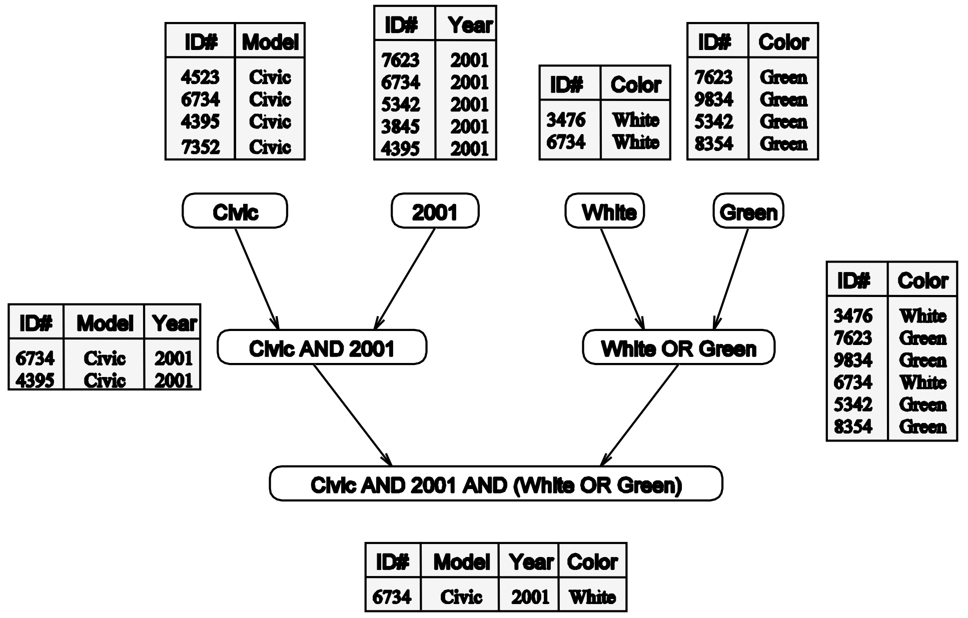
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Example: Database Query

- Task: select set of elements that satisfy a predicate
- Edge: output of one task is the input of the next

MODEL = "CIVIC" AND YEAR = 2001 AND

(COLOR = "GREEN" OR COLOR = "WHITE")



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

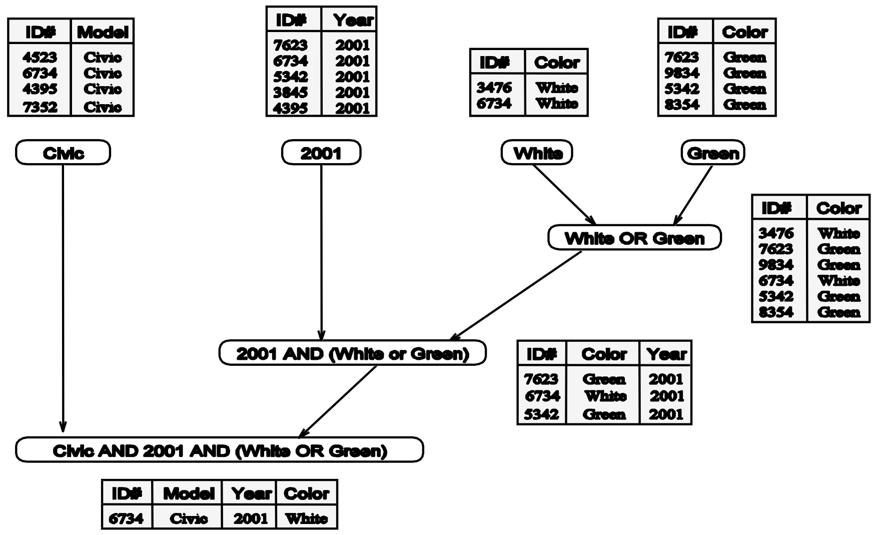
ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

Example: Database Query

MODEL = "CIVIC" AND YEAR = 2001 AND

(COLOR = "GREEN" OR COLOR = "WHITE")


Civic
2001
White
Green
White OR Green
2001 AND (White or Green)
Civic AND 2001 AND (White OR Green)

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

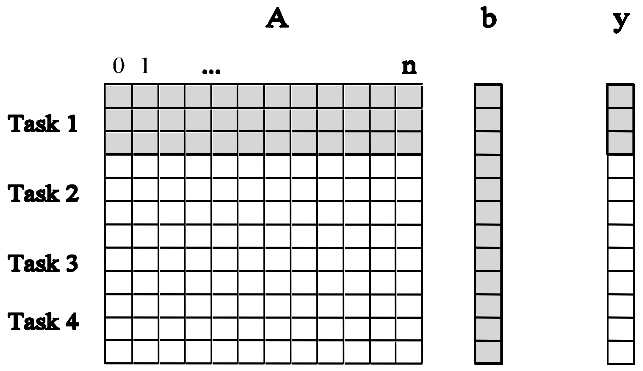
ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

Different decompositions can lead to different parallelism and different amounts of work

Task granularity

- Granularity = task size
- Fine-grain: large number of small tasks
- Coarse-grain: small number of large tasks
- For dense matrix-vector multiply:
    - Fine-grain: one task per element of y
    - Coarse-grain: one task per 3 elements of y

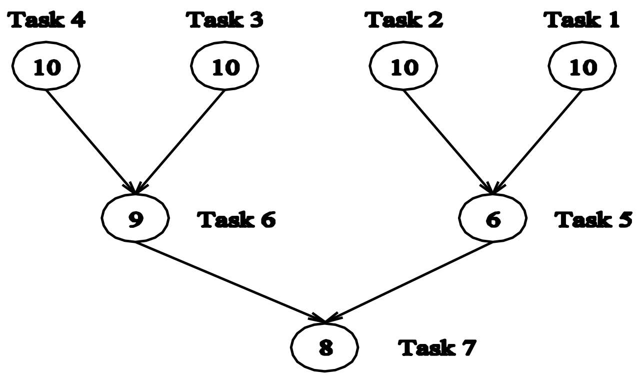


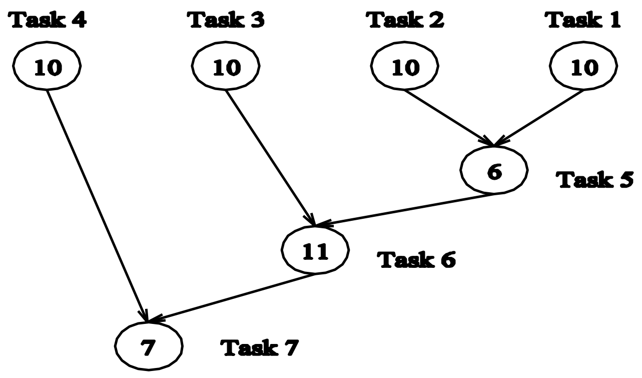
Degree of concurrency

- Definition: the number of tasks that can execute in parallel
- May change during execution
- Metrics:
  - Maximum degree of concurrency
    - Largest number of concurrent tasks at any point
  - Average degree of concurrency
    - Average number of tasks that can be processed in parallel
- Degree of concurrency vs. task granularity
  - Inverse relationship

Critical path

- Edge in task dependency graph represents task serialization
- Critical path = longest path through the graph
  - Represents a lower bound on parallel execution time





Number in vertex is cost

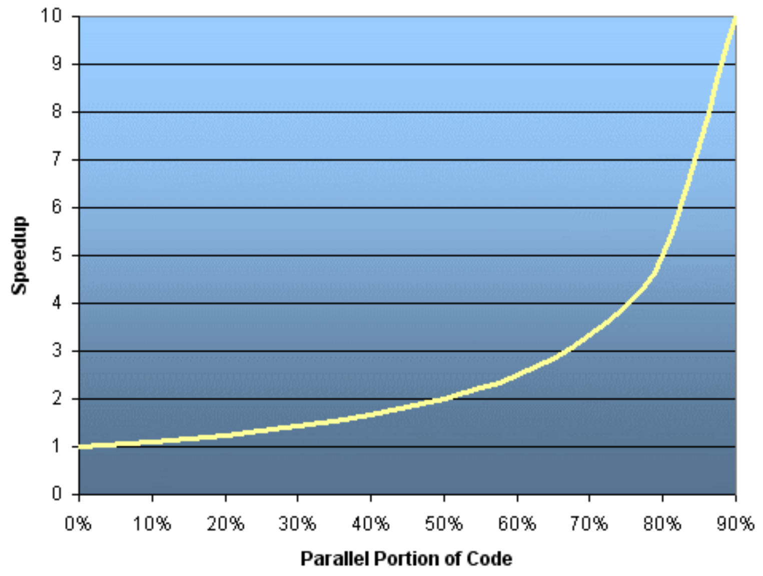
Thought experiment

- N people, how many handshakes?
  - In serial?
  - In parallel?
- Molecular dynamics: N atoms – what is the total force on each
  - Each has a position – compute pairwise forces, then compute displacement

Limits of Parallel programming

- Amdahl's law: the potential program speedup is defined by the fraction of work(P) that can be parallelised:
- P=0, speedup=1 (none)
- P=1, speedup infinite (in theory)

$$speedup = \frac{1}{1-P}$$

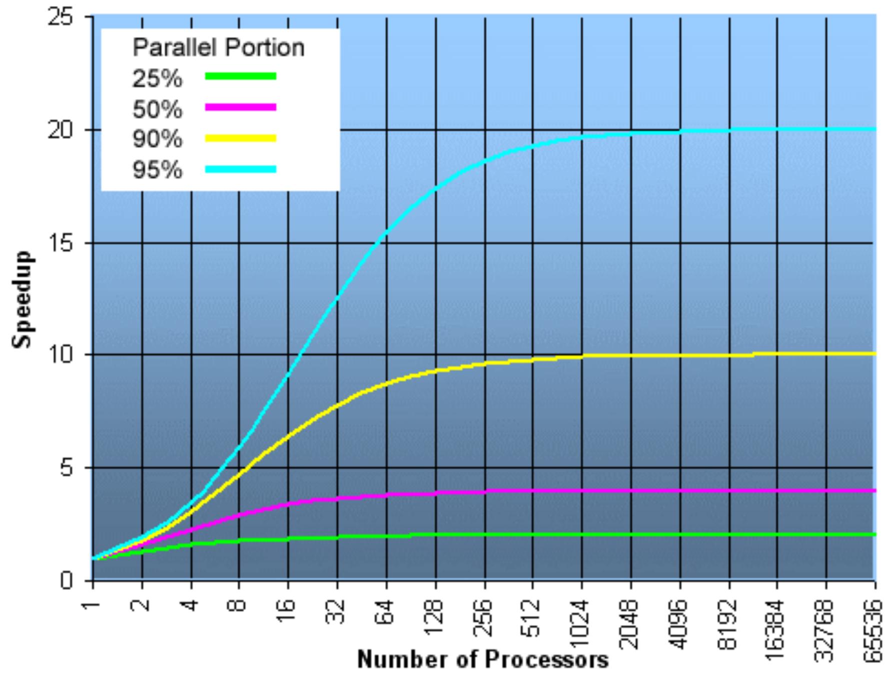


Amdahl’s Law

- Given N processors:
- Limits to the scalability!
- In most cases, some serial fraction is un-avoidable

$$speedup = \frac{1}{\underbrace{(1-P)}_{\text{Serial}} + \underbrace{P/N}_{\text{Parallel}}}$$

All is lost?



Scaling the problem

- In many cases, trying to solve a larger problem helps: the serial overhead is constant.
    - E.g. 2-factor PDE computation, for a given size:

```
2D calculation: 85 seconds   85%
Serial overhead:     15 seconds   15%
```

- Doubling the discretisation points in each dimension and 2x the timesteps:

```
2D calculation:680 seconds   97.84%
Serial overhead:     15 seconds    2.16%
```

Leaves to Gustafson' Law...

Measuring parallel performance

- We have a number of theoretical metrics
  - Ratio of sequential and parallel parts (Amdahl's Law)
  - Average/Maximum degree of parallelism
  - Task granularity, etc...
  - But these only help with the analysis of parallel algorithms, and give some theoretical bounds – they do little in the way of predicting how fast our implementation will be
  - Furthermore, they are usually focused on computations, not data movement
- The most straightforward way to measure parallel performance is to compare the execution times of the sequential program to the parallel one
  - Speedup = sequential time / parallel time

Matrix-Matrix multiply OpenMP

```
#pragma omp parallel for
for (int i=0; i < N; i++)
    for (int j=0; j < N; j++)
      for (int k=0; k < N; k++)
        c[i*N+j] += a[i*N+k] * b[k*N+j];
```

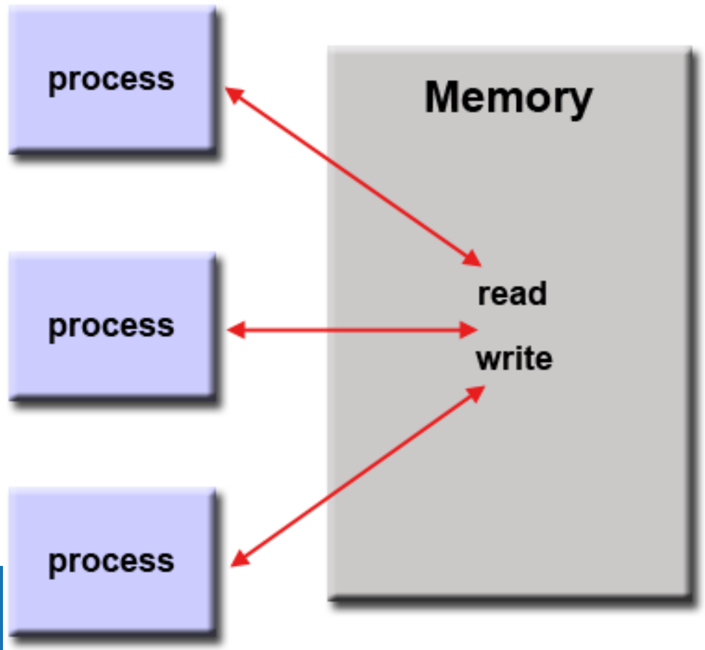
- It’s that simple with OpenMP
  - Task granularity is chosen automatically
  - But there is only N-way concurrency and N^3 operations – could this be improved somehow?

Exercise

- Implement a matrix-matrix multiplication
  - Complete matmat.cpp skeleton
  - Test the –Ofast and the –O0 flags' effect
- Add `#pragma omp parallel for` in front of the outermost loop and compile with –fopenmp
  - Observe the speedups and overheads
- Compute theoretical number of operations
  - Compute operations/second
- Optional exercise: create a graph which shows achieved operations/second at different matrix sizes – 100 to 5000
  - And with/without OpenMP

Shared Memory Model

- Tasks (processes/threads) share a common address space, which they read/write to asynchronously
- Mechanisms to control access and prevent race conditions (locks, semaphores)
- No data "ownership" – equal access -> fairly simple
- **Data locality:** difficult to manage
  - Keeping data local to where we work on it speeds up accesses
  - Difficult to control



Shared memory programming

- Most parallel programming models work in a shared memory environment
  - All SIMD models are naturally shared memory
- Question is how do we control parallelism, synchronisation, data access to shared and private data
  - We tend to call concurrent streams of instructions that can address the same shared memory “threads”
  - Threads are spawned (created) by a master process, they have their own state (e.g. where in the instruction stream they are, and some private data)
  - Different programming models give different ways to control threads

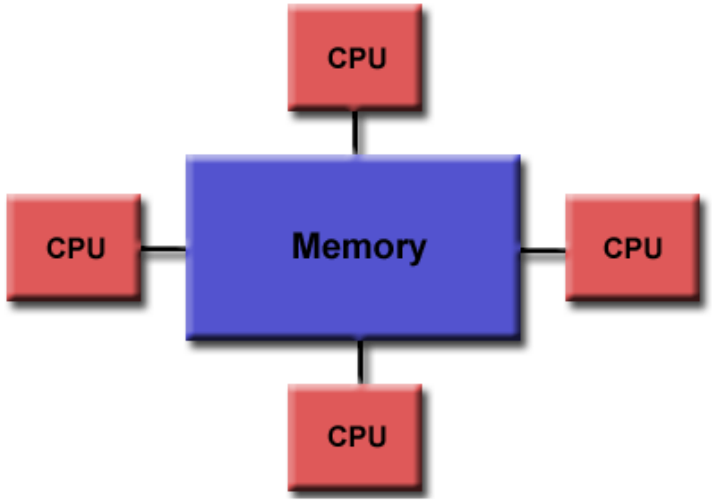
Threads model

- Implementation:
  - From a programming perspective, we need a set of API calls from within the source code, as well as compiler directives in the code to indicate parallel execution of tasks
- OpenMP
  - Industry standard: jointly defined by a group of HW/SW organizations
  - Compiler directive based
  - Portable (OS and languages, C/Fortran)
  - Easy to use – can be added incrementally

OpenMP

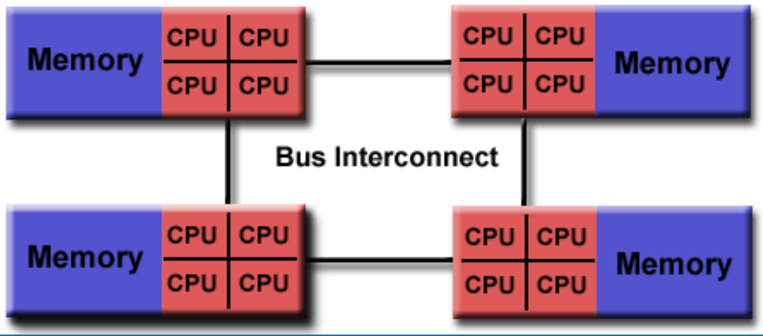
- Open Multi-Processing
- Three components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- High-level model
  - Implicit mapping and load balancing of work
- Standard
- Portable
- For more: https://computing.llnl.gov/tutorials/openMP/

OpenMP view of memory



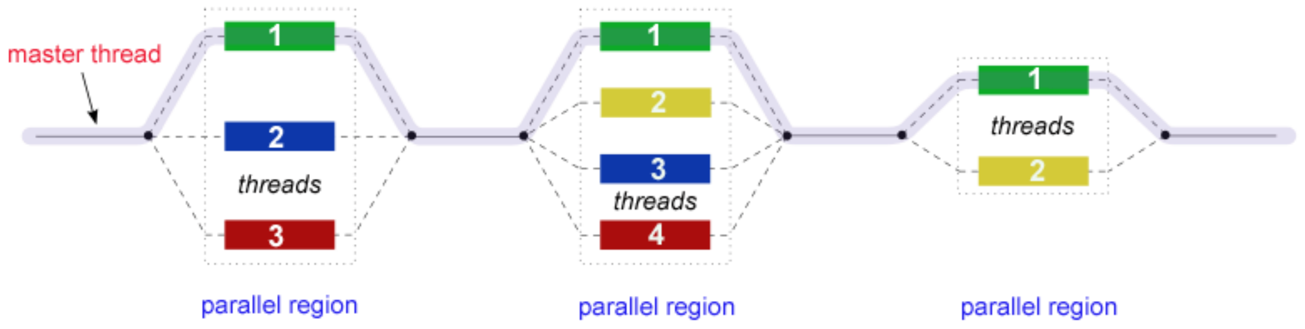
Uniform Memory Access

Reality may be:
Uniform Memory Access

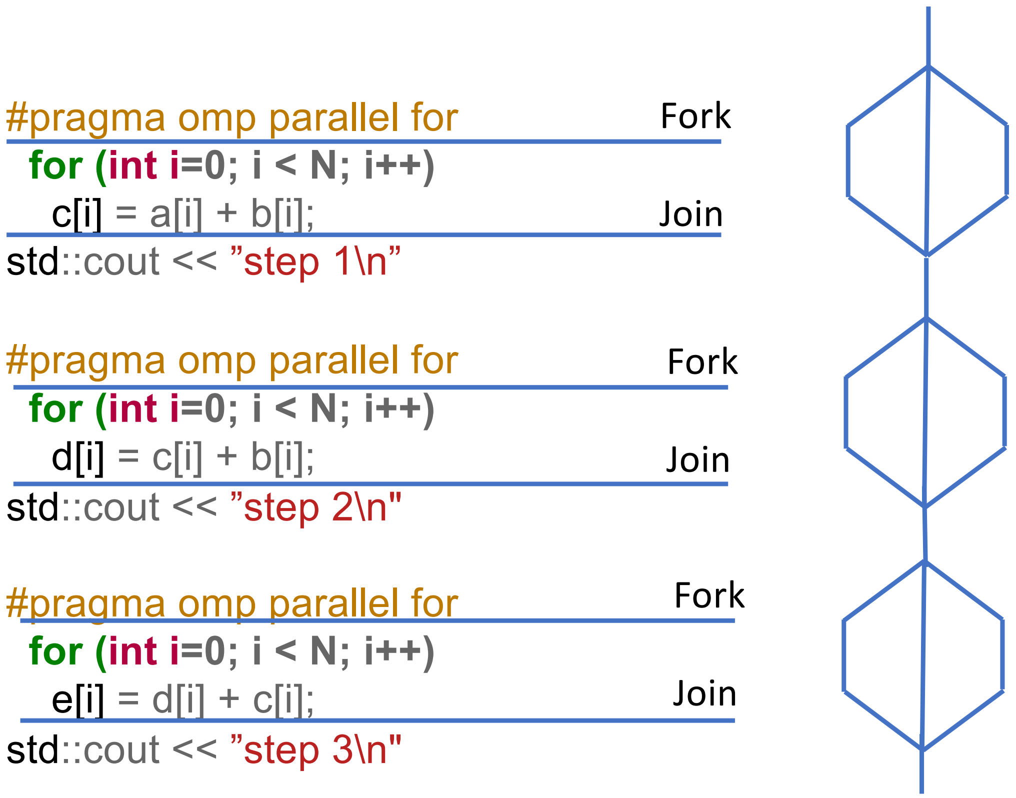


Fork-join model

- All programs begin as a single process: the **master thread**. It executes sequentially until the first **parallel region** construct.
- FORK: master thread creates a team of parallel threads
- Statements within the parallel region are executed in parallel among the different threads
- JOIN: Having completed the parallel region, threads synchronize and terminate, leaving the master thread.



Fork-join model



```
#pragma omp parallel for
  for (int i=0; i < N; i++)
    c[i] = a[i] + b[i];
std::cout << ”step 1\n”

#pragma omp parallel for
  for (int i=0; i < N; i++)
    d[i] = c[i] + b[i];
std::cout << ”step 2\n"

#pragma omp parallel for
  for (int i=0; i < N; i++)
    e[i] = d[i] + c[i];
std::cout << ”step 3\n"
```

Compiler directives

- Appear as comments in the code and ignored by the compiler unless instructed to (compiler flag)
- OpenMP directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Syntax:

```
Sentinel    directive-name            [clause, ...]
#pragma omp parallel default(shared) private(beta,pi)
```

Runtime Library Routines

- The OpenMP API includes a growing number of library routines
- Used for many things:
  - Setting and querying the number of threads
  - Querying the threads unique identifier
  - Setting/querying dynamic threads, nested parallelism
  - Querying time
  - etc...

```
#include <omp.h>
int omp_get_num_threads(void)
```

Environment variables

- OpenMP provides several environment variables to control the execution of parallel code
- Can be used to:
  - Set number of threads
  - Specify how loop iterations are divided
  - Binding threads to physical processors
  - Controlling nested parallelism, dynamic threads
  - etc.

```
export OMP_NUM_THREADS=4
```

Directives

#pragma omp	Directive-name	[clause,...]	newline
Required for all directives	A valid OpenMP directive.	Optional. In any order and repeated as necessary	Required. Precedes the structured block which is enclosed

```
#pragma omp parallel default(shared) private(beta,pi)
```

Rules:

- Case sensitive
- Only one directive-name per directive (there are a few combinations)
- Each directive applies to at most one succeeding statement, which must be a structured block
- Long directive lines can be ”continued” on succeeding lines with “\”

Parallel region construct

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
- Format:

Optional parameters

```
#pragma omp parallel [clause ...]  newline
                      if (scalar_expression)
                      private (list)
                      shared (list)
                      default (shared | none)
                      firstprivate (list)
                      reduction (operator: list)
                      copyin (list)
                      num_threads (integer-expression)

   { ... structured_block... }
```

Parallel region

- When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master is a member, and has ID 0.
- Starting from the beginning of the region, the code is “duplicated” and all threads will execute that code
  - Variables defined inside a parallel region, will be replicated too
- There is an implied barrier (synchronisation point) at the end of the parallel section, and only the master continues execution beyond it.
- If a thread terminates within a parallel region, all threads in the team will terminate and the work done is undefined

```
#include <omp.h>
main ()  {
int var1, var2, var3;

Serial code
      .
Beginning of parallel section. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
  Other OpenMP directives
  Run-time Library calls
  All threads join master thread and disband
  }
Resume serial code
         .
}
```

Scoping

```
int main() {
  #pragma omp parallel
  {
         ...
         myfun()
  }
}
void myfun()
...
#pragma omp critical
{
         ...
}
...
#pragma omp sections
{
         ...
}
...
```

Directive Scoping

- Static Extent:
  - The code textually enclosed between the beginning and the end of a structured block following a directive
  - The static extent of a directive does not span multiple routines or source files
- Orphaned directive:
  - An OpenMP directive that appears independently from another enclosing directive: exists outside of another directive’s static extent
- Dynamic extent:
  - Includes both its static extent and the extents of its orphaned directives
- Why? – There are some scoping rules on two directives can associate and nest within each other

Scoping

```
int main() {
  #pragma omp parallel
  {
         ...
         myfun()
  }
}
void myfun()
...
#pragma omp critical
{
         ...
}
...
#pragma omp sections
{
         ...
}
...
```

Static Extent

Orphaned Directives

Dynamic Extent

How many threads?

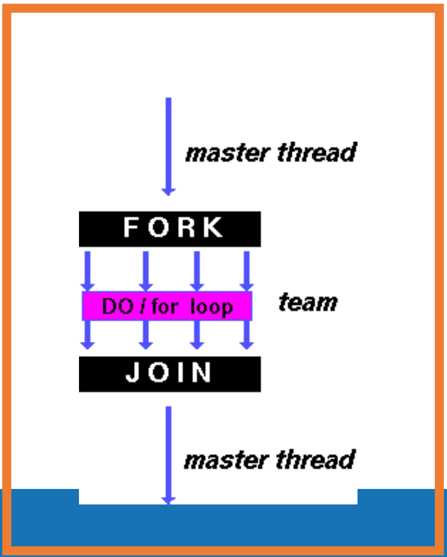
- The number of threads depends on the following:
  1. The if() clause (has to be true, otherwise serial)
  2. The num_threads() clause
  3. Use of the set_omp_num_threads() API
  4. The OMP_NUM_THREADS environment variable
  5. Implementation default (number of CPUs)
- Numbering is from 0 (master) to N-1

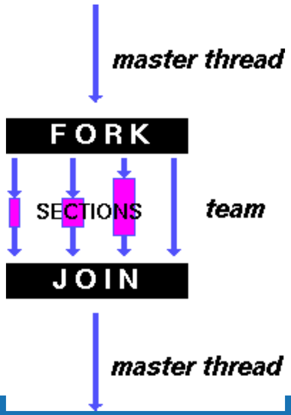
Exercise - omp_hello.cpp

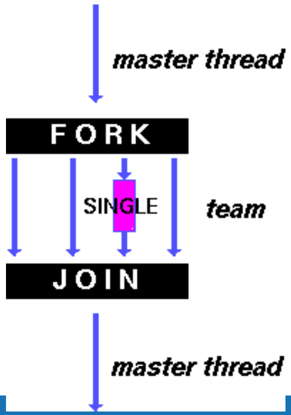
```
#include <omp.h>
#include <iostream>

int main()  {
int nthreads, tid;

/* Fork a team of threads with each thread
   having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  std::cout << "Hello World from thread = " << tid << std::endl;

  /* Only the master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    std::cout << "Number of threads = " << nthreads << std::endl;
    }
  }  /* All threads join master thread and terminate */
}
```

Work-sharing constructs

- Divides the execution of the enclosed code region among the members of the team that encounter it
- Does not launch new threads
- No implied barrier upon entry, but there is one at the exit.
- Has to occur within parallel region, encountered by all members, in the same order







for Directive

- The for directive specifies that the iterations of the loop immediately following it must be executed in parallel.

Optional parameters

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

Clauses

- Schedule: describes how iterations are divided among the threads. No default, the optimal one depends on the problem
  - **Static**: pieces of size *chunk*, statically assigned to threads. Default *chunk* is size/num_threads.
  - **Dynamic**: pieces of size *chunk*, dynamically scheduled. Default *chunk* is 1.
  - **Guided**: Like dynamic, but *chunk* size continuously decreases, starts out with size/num_threads, then leftover/num_threads, etc.
  - **Runtime**: deferred, depends on OMP_SCHEDULE environment variable.
  - **Auto**: let the compiler/runtime decide

Clauses

- **Nowait**: no synchronisation at the end of the loop
- **Ordered**: requires that some parts of the iterations are executed as in a serial program + `#pragma omp ordered`
- **Collapse**: specifies how many nested loops to be collapsed into one large iteration space and then divided up
- <u>Restrictions:</u> has to be for loop with integer iteration variable

Vector add

```
#include <omp.h>
#include <iostream>
#include <chrono>
#include <vector>
#define N     100000000
int main (int argc, const char **argv)
{
  std::vector<float> a(N), b(N), c(N);
  /* Some initialisation */
  for (int i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  auto t1 = std::chrono::high_resolution_clock::now();
#pragma omp parallel for
  for (int i=0; i < N; i++)
    c[i] = a[i] + b[i];
  auto t2 = std::chrono::high_resolution_clock::now();
  std::cout << "took "
    << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).count()
    << " milliseconds\n";
}
```

Combinations

- Two combined directives for convenience:
  - Parallel for

```
#pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
```

Need for Synchronisation

```
THREAD 1:
increment(x)
{
    x = x + 1;
}
THREAD 1:
10  LOAD A, (x address)
20  ADD A, 1
30  STORE A, (x address)
```

```
THREAD 2:
increment(x)
{
    x = x + 1;
}
THREAD 2:
10  LOAD B, (x address)
20  ADD B, 1
30  STORE B, (x address)
```

A possible sequence:
1. Thread 1 loads x into register A -> 2. Thread 2 loads x into register B ->
3. Thread 1 adds 1 to A -> 4. Thread 2 adds 1 to B ->
5. Thread 1 writes A to x -> 6. Thread 2 writes B to x

Master directive

- Specifies a region that is to be executed only by the master thread of the team.
- No implied barrier

```
#pragma omp master  newline

   structured_block
```

Critical directive

- A region (section) of code that mast be executed by only one thread at a time

```
#pragma omp critical [ name ]  newline

    structured_block
```

- If a thread is in the critical section, and an other arrives, it will block until the first finishes
- Can be named, so multiple sections in code are treated as one – all unnamed sections are treated as the same section

Critical section

```
#include <omp.h>

main()
{

int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel section */
  printf(„%d\n“,x);
}
```

Can this possibly be faster?

Critical section

```
typedef struct {
    int index;
    int value;
} Event;

...
    #pragma omp parallel for
    for(int i = 0; i < DATA_SIZE; i++) {
        if(data[i] > THRESHOLD) {
            #pragma omp critical
            {
                rareEvents[eventCount].index = i;
                rareEvents[eventCount].value = data[i];
                eventCount++;
            }
        }
    }
```

Barrier directive

- Synchronises all threads in the team
- When the barrier directive is reached, a thread will wait until all other threads have reached that barrier. Then all threads resume executing code after the barrier
- All threads (or none) must encounter the barrier

```
#pragma omp barrier  newline
```

Barriers

```
#pragma omp parallel shared(image)
{
    // Step 1: Apply filter to assigned section
    applyFilter(image_section);

    // Barrier: Wait for all threads to finish Step 1
    #pragma omp barrier

    // Step 2: Apply the second enhancement
    applyEnhancement(image_section);
}
```

Atomic directive

- Specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it.
- It’s like a mini critical section, applies to only one (following) statement

```
#pragma omp atomic  newline

   statement_expression
```

Atomic directive

```
#pragma omp parallel for
  for(int i = 0; i < DATA_SIZE; i++) {
      int bin = data[i] * NUM_BINS / MAX_VALUE; // Simple bin calculation
      #pragma omp atomic
      histogram[bin]++;
  }
```

Data sharing

- Important to understand how variables are shared and how to use data scoping
- By default most variables are shared
    - Except loop index variables and anything created on the stack within a parallel region (i.e. local variables)
- Data scope attributes:
    - Private, firstprivate, lastprivate, shared, default, reduction, copyin
- Combined with other directives to control the scoping of enclosed variables

Private

- Lists variables that are to be private to each thread
- Behaviour:
  - A new object of the same type is declared once for each thread in the team
  - All references to the original are replaced with references to the new one
  - Should be assumed **uninitialised**
- Firstprivate combines private and initialisation
- Lastprivate writes back the value from the last iteration or section

```
private (list)
firstprivate (list)
lastprivate (list)
```

Shared

- Declares listed variables to be shared among all the threads in the team
- Exists only in one memory location, and all threads can read or write to that address
- Programmer’s responsibility to ensure safe access

```
shared (list)
```

Reduction

- Performs a reduction on the variables in the list
- A private copy is created for each thread, and at the end they are reduced to the globally shared variable

```
    reduction (operator: list)

#pragma omp parallel for      \
  default(shared) private(i)  \
  reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);
```

Reduction

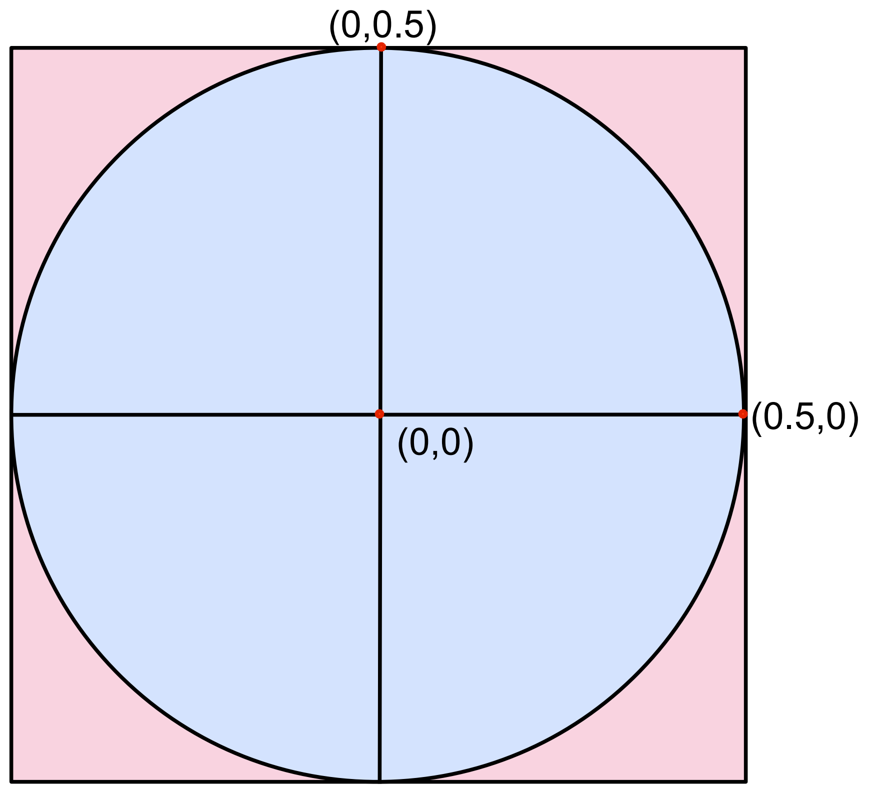
- Variables have to be scalar (no arrays or structures)
- Have to be declared shared in any enclosing region
- Assumes associativity!

```
x = x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x
```

x is a scalar variable in the list
expr is a scalar expression that does not reference ***x***
op is not overloaded, and is one of +, *, -, /, &, ^, |, &&, ||
binop is not overloaded, and is one of +, *, -, /, &, ^, |

Exercise - Compute Pi

- Open calc_pi.cpp
- Compile:
  - g++ -std=c++11 calc_pi.cpp – o calc_pi – Ofast -fopenmp
- Generate random points between [-0.5 0.5]
- Test if inside the circle
- Ratio of circle to square
- pi ≈ 4*(#inside/#total)
- double pi = 4.0*(inside/10000.0);



Semester project

- Computational fluid dynamics problem
- https://classroom.github.com/a/jmyW7aJf
- You can plot the output file with Matlab to see fluid flow

Assignment – due March 17 midnight

- Add timing to your code – time the “main time iteration”, print elapsed time at the end
- Parallelise the code with OpenMP
  - Reductions!
  - Make sure your code validates – gives (almost) the same velocities with/without OpenMP
- Push your changes to the GitHub Classroom repo – make sure it’s there by looking at it on the web interface