Parallel design patterns ARCHER course

Loop parallelism and fork/join

Reusing this material

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

Program structures

SPMD

Master/worker

Loop parallelism

Fork/join

Active messaging

Vectorisation

Data structures

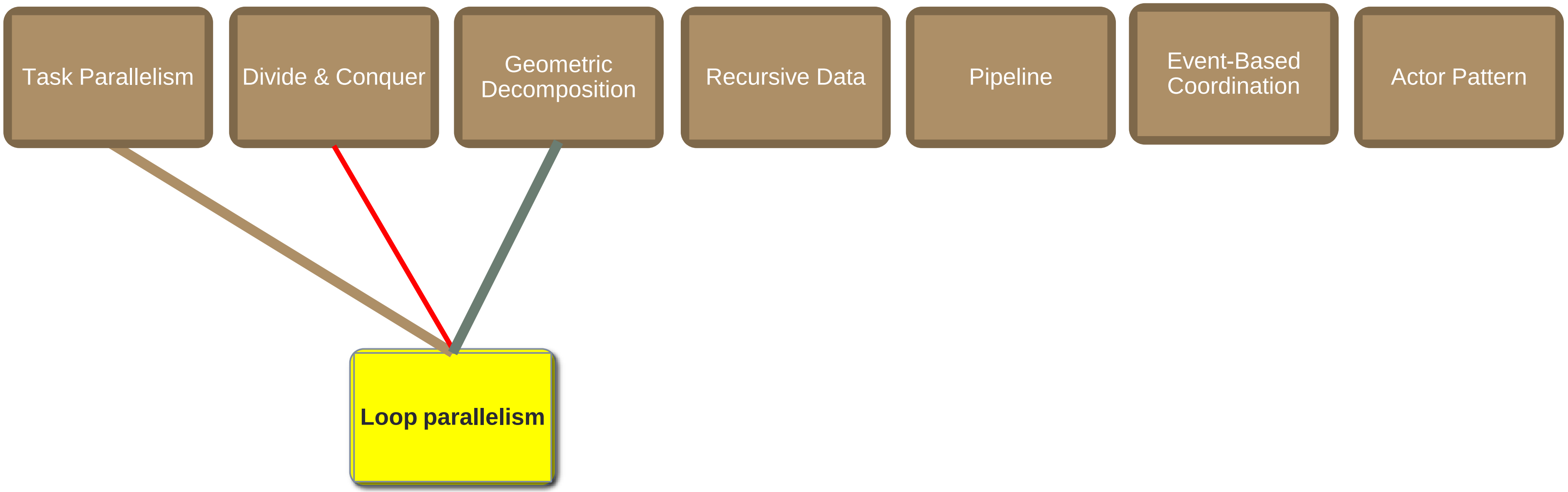
Shared data

Shared queue

Distributed array

Supporting structures

Loop Parallelism: The Problem



- Loop Parallelism is an *Implementation Strategy*
- The Problem: Given a serial program whose run time is dominated by a set of computationally intensive loops, how can this be translated into a parallel program?

Loop Parallelism: Context

- There are many existing loop-based programs, particularly in scientific and engineering applications
- This type of parallelism can be added to a code incrementally
  - Particularly important for large, well-established codes
- Often, little or no restructuring of the code is required
- Not suited to all programs with loops
- Not suited to all system architectures
- Works best with small-scale parallelism
  - Not as much of a limitation as you might think, especially with prevalence of multi-core
  - Can also be used as part of a hybrid solution

Loop Parallelism: Forces

- Sequential Equivalence
  - Identical results when run on one or many UEs.

- Incremental parallelism / refactoring
  - This is really what makes this pattern powerful, and a bit different from some of the others. It comes into its own when there is already an existing serial code
  - It would be nice to test each bit of parallelism as we add it

- Loop independence & optimisation
  - Can trade off against the other two

Loop Parallelism: Solution

- This pattern is closely aligned with the style of programming usually employed with OpenMP

- Find the bottlenecks
- Eliminate loop-carried dependencies
- Parallelise the loops
- Optimise the loop schedule

- Sometimes, to maintain efficiency by minimising the parallel loop overhead, it is necessary to
  - Join neighbouring loops, or
  - Merge nested loops

Finding The Bottlenecks

- Very important!
  - Because the incremental parallelisation approach lends itself to making changes to a code immediately, it can be tempting to pick a loop (the first one?) and put some OpenMP directives around it
    - …but just because you *can* doesn't mean you *should*!

- Identify computationally intensive loops *taking into account representative data sets* either through
  - Inspection and theoretical analysis of code, or more commonly
  - Measuring the performance of the code with performance analysis tools
- Also bear in mind that if the runtime is not dominated by the loops, or if not all loops can/will be parallelised, the parallel performance will be ultimately limited by Amdahl's Law.

Eliminating Loop-Carried Dependencies

- Loop iterations must be nearly independent
- Remove dependencies where possible:
  - Replace iterative series with closed forms
  - Separable dependencies:
    - Replicate data, execute task, recombine data

- Use explicit synchronisation to protect shared data
  - One-at-a-time execution (often overly conservative)
    - OMP Critical
    - Owner UE in MP environment
  - Non-interfering operations
    - OMP Critical with named sections
  - Reader/Writer locks
  - More details in *Shared Data* pattern (later in the lecture)

Replacing with the closed form

```
int ii=0;jj=0;
for (int i=0;i<N;i++) {
  ii++;
  d[ii]=time_consuming_work(ii);
  jj=jj+i;
  a[jj]=large_calculation(jj);
}
```

- ii and jj create a dependency between iterations (tasks)
- But ii = i
- And jj is the sum of 0 through i

```
for (int i=0;i<N;i++) {
  d[i]=time_consuming_work(i);
  a[(i*i+i)/2]=large_calculation((i*i+i)/2);
}
```

Parallelising The Loops

- Once you've dealt with the dependencies, this is the easy bit!
- OpenMP has constructs exactly for this purpose
  - which are *semantically neutral*
- Loops can be parallelised one at a time
  - and tested at each stage

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

Optimising the loop schedule

- !$OMP PARALLEL DO SCHEDULE(type, chunk_size)
  - static, dynamic, guided, (runtime, auto)

- Again, this can be added incrementally
- Dynamic is very similar in effect to a task farm
- The DO loop cannot be a DO WHILE, so you can't do a task farm with an unknown number of tasks
- Choice can sometimes be chosen if performance of iterations is well understood, but often the best approach is to experiment

Other loop optimisations

- Compute times for the loop iterations should be large enough to offset the parallel overhead.
  - Merge loops (fusion)

```
for (i=0;i<n;i++) {
   function_a(i);
}
for (i=0;i<n;i++) {
   function_b(i);
}
```

→

```
for (i=0;i<n;i++) {
   function_a(i);
   function_b(i);
}
```

- More loop iterations per UE give greater scheduling flexibility
  - Coalesce loops

```
for (i=0;i<n1;i++) {
  for (j=0;j<n2;j++) {
     function_a(i,j);
  }
}
```

→

```
for (c=0;c<n1*n2;c++) {
  i=c/n1;
  j=c%n2;
  function_a(i,j);
}
```

Other Loop Optimisations

- Stripmining
  - Enables the use of vector or SIMD instructions

```
for i = 0…n
    A(i)=f(i) + k(i)
```

```
for i = 0…n by B
    for j = i…i+B
        A(j)=f(j) + k(i)
```

- Interchange
  - Change order of iterations (i.e. column major)

```
for i=0..n
  for j=0..n
    A(i,j)=f(i,j)
```

```
for j=0..n
  for i=0..n
    A(i,j)=f(i,j)
```

Other loop optimisations

- Tiling
  - Many cache blocking algorithms are built on this.
  - Stripmine several loops and perform interchanges to bring these forward

```
for i = 0..n
  for j = 0..n
    for k = 0..n
      C(i,j) += A(i,k) * B(k,j)
```

```
for ii = 0..n by B
  for jj = 0..n by B
    for kk = 0..n by B
      for i = ii..ii + B
        for j = jj..jj + B
          for k = kk..kk + B
            C(i,j) += A(i,k) * B(k,j)
```

Other Loop Optimisations

- Fission
  - Split the loop

```
for i = 0…n
   for j= 0..n
     A(i,j)=B(i,j)+ C(i,j)
     D(i,j)=A(i,j-1) * 2
```

↓

```
for i = 0…n
   for j= 0..n
     A(i,j)=B(i,j)+ C(i,j)
   for j= 0..n
     D(i,j)=A(i,j-1) * 2
```

Other Loop Optimisations

- Unrolling
  - Replicate body to reduce overhead

```
for i = 0…n
    A(i)=B(i)+C(i)
```

```
for i = 0…n by 4
    A(i)=B(i)+C(i)
    A(i+1)=B(i+1)+C(i+1)
    A(i+2)=B(i+2)+C(i+2)
    A(i+3)=B(i+3)+C(i+3)
```

- Unroll and jam
  - Unroll outer loop, merge copies of inner loop

```
for i=0..n
  for j=0..m
    A(i)=A(i)+B(j)
```

```
for i=0..n by 2
  for j=0..m
    A(i)=A(i)+B(j)
    A(i+1)=A(i+1)+B(j)
```

Performance considerations

- Assumption is that there is a shared address space with uniform access time
  - Not necessarily true, NUMA architectures
- First touch principal is important
  - Data is located local to a thread that first touched it, therefore locate initialisation and compute on the same UE.
- False sharing
  - Data is not shared, but resides on the same cache line
  - These are repeatedly invalidated

False sharing example

```
N=4
M=1000
double A[N] = 0.0

#pragma omp parallel for private(j,i)
for (j=0; j<N; j++) {
    for (i=0; i<M; i++) {
        A[j]+=work(i,j)
    }
}
```

```
#pragma omp parallel for private(j,i,temp)
for (j=0; j<N; j++) {
    temp=0.0
    for (i=0; i<M; i++) {
        temp+=work(i,j)
    }
    A[j]+=temp;
}
```

Loop Parallelism / SPMD

- You can have loops in an SPMD program
- Key point with loop parallelism is that you never explicitly mention a thread ID

- Often SPMD is process based whereas loop parallelism is thread based
  - Requires a fundamental difference in thinking between shared nothing and shared everything
  - These patterns can be mixed (i.e. hybrid MPI-OpenMP) which might give extra performance/scalability at the cost of code complexity

Loop Parallelism => OpenMP?

- Often synonymous with OpenMP on CPUs
- Possible in OO languages with parallel iterators
- HPF
  - forall
- UPC
  - upc_forall(init; test; update; affinity)
- Fortress
  - Loops are parallel by default!
- Others
  - par (parallel) and for (sequential)

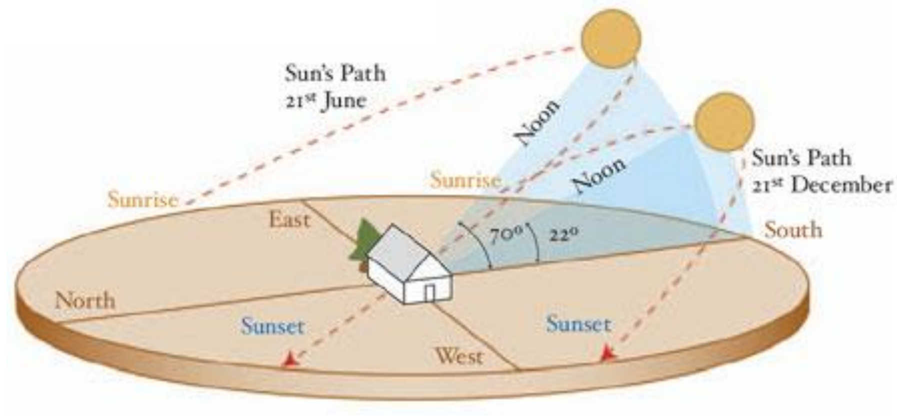
SunCast example

- Integrated Environmental Solutions is a Glasgow based SME that EPCC worked with a few years ago

- They are all about improving the energy efficiency of buildings
  - SunCast enables them to study the impact of the sun's rays on both existing and architectural designs
  - They can then understand the relation of the sun to the thermal properties of the building and general comfort
- Their algorithm was serial and they wanted to be able to run this on multi-core laptops

epcc

SunCast example



- There are quite a few different sun position scenarios that need to be calculated
  - Each of which is a loop
- There are also multiple rays from the sun hitting the building at any one time which need to be calculated
  - These rays are also in a loop
- Loop parallelism can therefore be applied at two levels – at each position & for each ray
  - Sped up calculation from a few hours to under an hour on a laptop

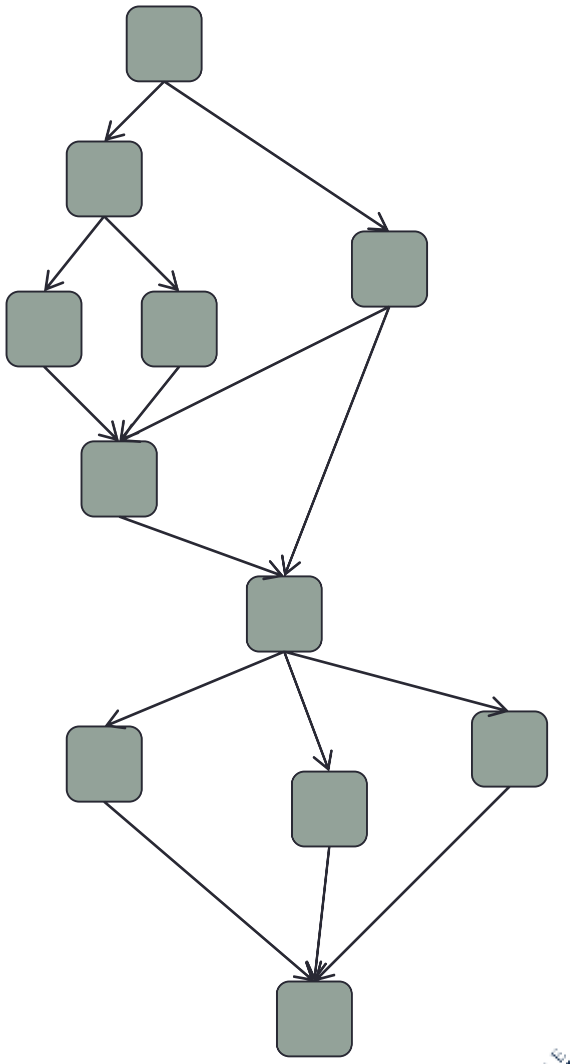
```
do i = 22 to 70
  do j = 1 to num_rays
    ......
  end do
end do
```

Loop parallelism: Summary

- Loop Parallelism has an unusual property – that it is an incremental parallelism pattern
- Loop Parallelism can also leave programs runnable in serial
- Useful since so many programs are loop based
- The programming model for OpenMP
- Some gotya’s to be aware of

Fork-Join: The Problem

- You have a problem where the number of concurrent tasks varies throughout the execution of the program and a simple control structure such as a parallel loop is not sufficient. How can a parallel program be constructed around the dynamic set of tasks?



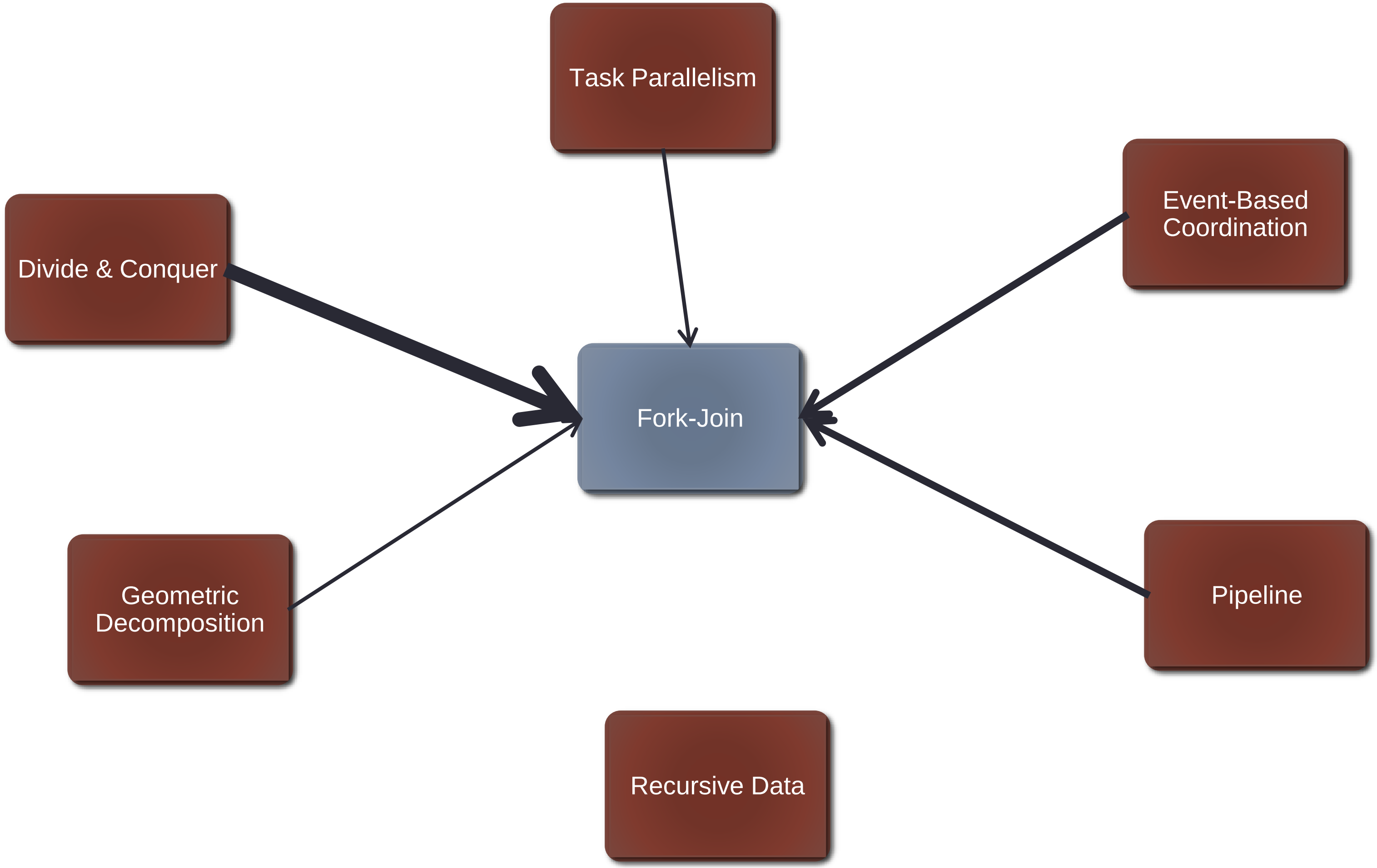
Fork-Join: The Context

- Applicable where the algorithm imposes an irregular or dynamic control structure
- Tasks are created dynamically (*forked*) and terminated (*joined* with the forking task) as the program continues to execute
- In some cases, the forking pattern would be very regular. In these cases, loop parallelism (discussed in a later lecture) would be a better choice
  - Fork-Join is more generally applicable
  - Loop parallelism can be thought of as a special case of Fork-Join
- A good match, for example, with the divide & conquer pattern discussed previously

Fork-Join: The Forces

- Algorithms often imply relationships between tasks, with the relationships arise dynamically. It can be useful to have the relationship between the UEs closely match the relationship between the tasks
- A one-to-one mapping between UEs and tasks is usually natural
  - but this must be balanced against the number of UEs that a system can handle
- UE creation and destruction are expensive operations. It may be desirable to structure the program so as to restrict the number of forks and joins.

Relationship to Parallel Algorithm Strategy



Fork-Join: The Solution

- Two Possible Solutions:
  - Direct task/UE mapping
  - Indirect task/UE mapping

- With Fork-Join the UEs are usually (but don't have to be) threads
- In both cases, a fork results in an extra thread (or several extra threads) being assigned to the problem and a join results in the removal of threads from working on the problem

Direct Mapping

- The simplest case
  - …and a common one
- Map each task to a single UE
- As new tasks are created, new UEs are created
- There is almost always a synchronisation point where the parent (forking) UE waits for the forked tasks to complete and the forked UE to re-join

Indirect Mapping

- Use a thread pool
- Create threads at the start
  - usually with same number of UEs as PEs
- Cheaper than thread creation/destruction
- Forking corresponds to taking a thread from the thread pool and joining returns it to the thread pool
- A bit like a low-level implementation of the Master-Worker pattern which will be discussed in more detail later

Fork-Join: OpenMP, Java, MPI

- The Fork-Join pattern is the standard programming model in OpenMP
  - OpenMP programs start as a single thread and on reaching a parallel construct, a team of threads is forked
  - At the end of the parallel region, the threads rejoin their parent
  - In the case of loops, you get the special case of loop parallelism
- The Fork-Join pattern is also the standard implementation model for Java threads
  - Java also provides classes/interfaces to help manage Fork-Join in java.util.concurrent
- Fork-Join can be implemented with MPI, but it's not such a natural fit
  - In this case, indirect mapping / process pools are often used

Fork Join in OpenMP

- Using non iterative loop directives
- Parallel sections

```
#pragma omp parallel sections
{
   #pragma omp section
   {
      ......
   }
   #pragma omp section
   {
      ......
   }
}
```

Fork - one thread executes code in here

Fork - one thread executes code in here

Join - all threads block here

Fork Join in OpenMP

- Tasks

```
#pragma omp parallel
{
  #pragma omp task
      some task
  #pragma omp task
      some task
  #pragma omp task
      some task
  #pragma omp taskwait
}
```

Fork – enqueue a task to be executed by a thread at some point

Fork – enqueue a task to be executed by a thread at some point

Fork – enqueue a task to be executed by a thread at some point

Join – wait for all tasks to complete

- Tasks run at scheduling points (such as implicit/explicit barriers)
- This can be more flexible than sections but also the synchronisation using *taskwait* can be more complex

Other languages too…..

Conclusions

- Fork-Join implementation strategy is suitable for irregular or dynamic control structures
  - Tasks are created (forked) and terminated (joined) dynamically