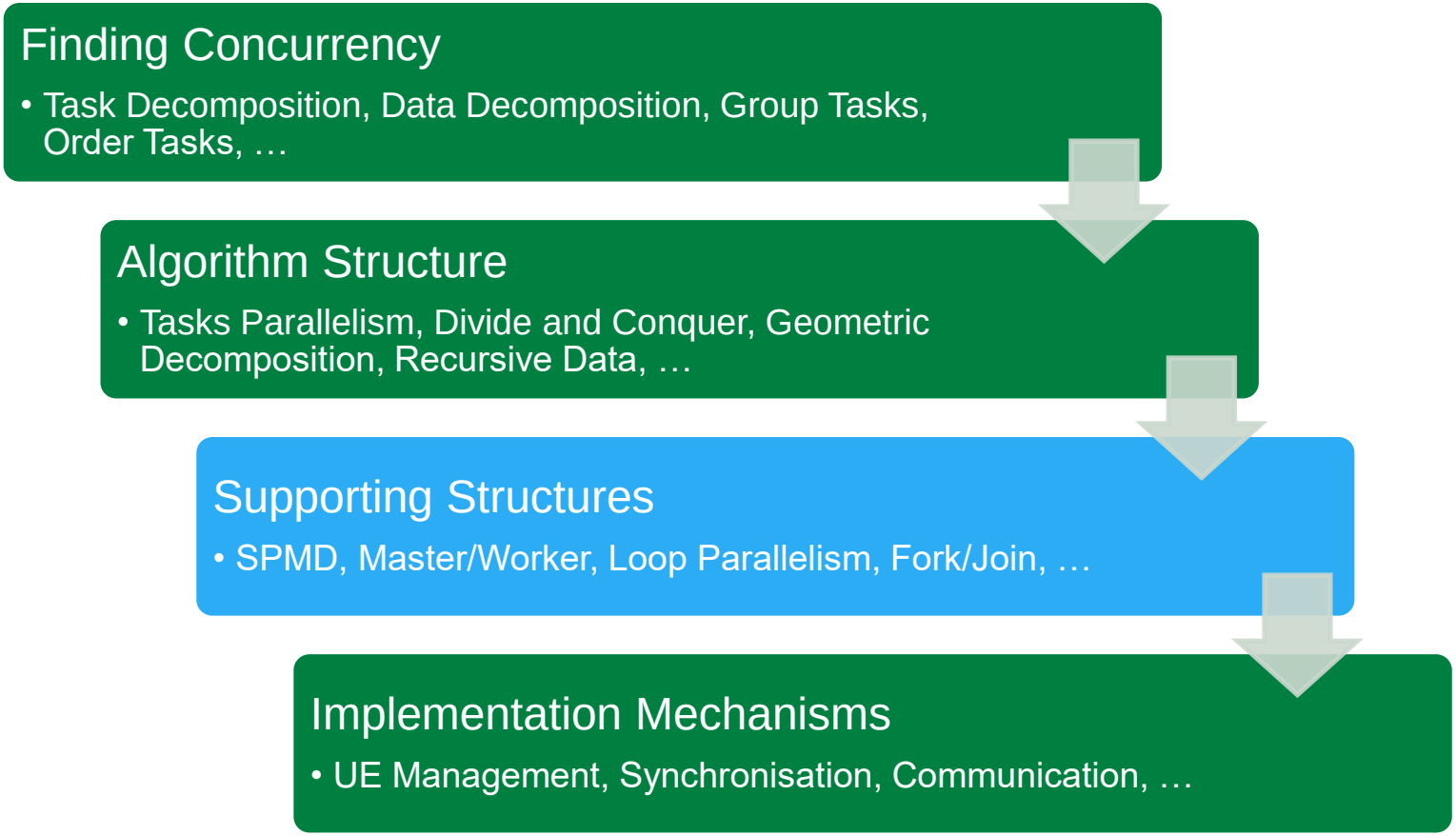
Parallel Design Patterns-L10

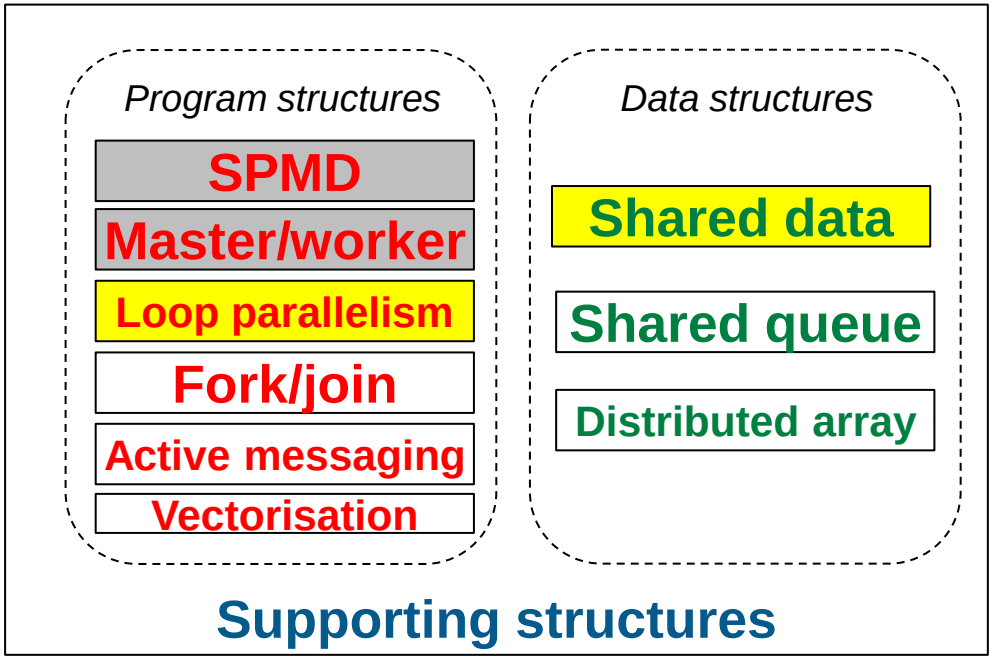
Loop Parallelism

Shared Data

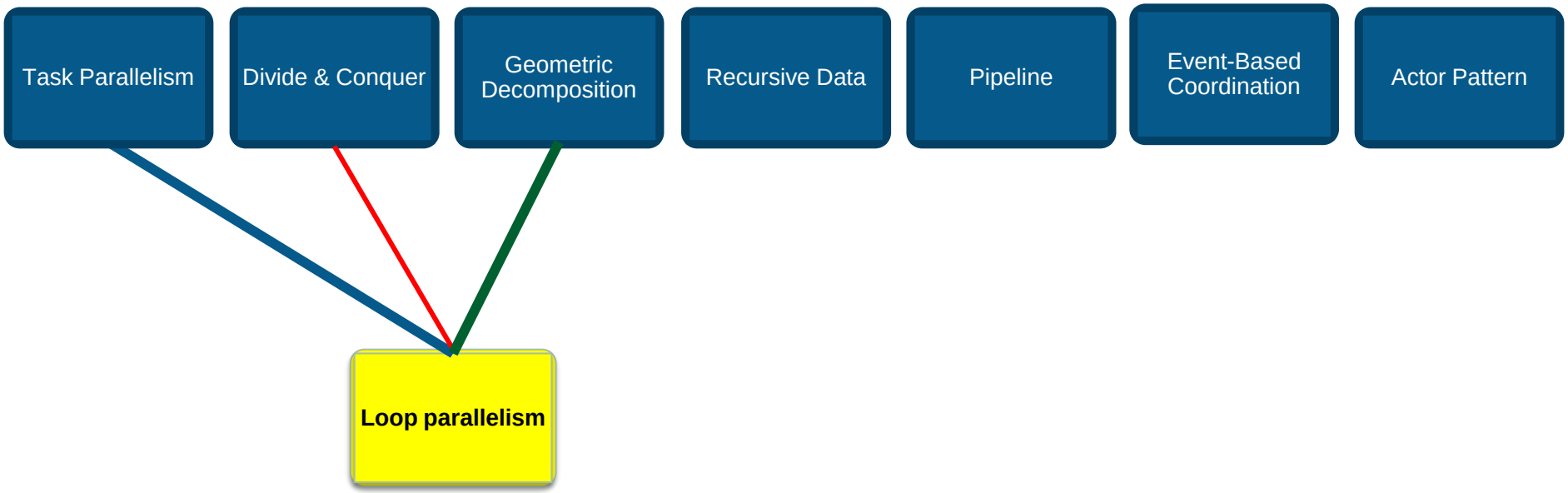
Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13



Supporting structures is also known as implementation strategy and we will use these terms interchangeably


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

1. Find the bottlenecks
2. Eliminate loop-carried dependencies
3. Parallelise the loops
4. Optimise the loop schedule
   - Optimising the mapping of iterations to UEs

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.

Profiling tools

```
  Samp% |      Samp | Group
        |           |  Function
        |           |   Caller

 100.0% | 11,956.0 | Total
|---------------------------------------
|  69.4% |  8,301.0 | USER
||--------------------------------------
||  69.4% |  8,300.0 | main
|||-------------------------------------
3||  20.5% |  2,453.0 | main:serial.c:line.61
3||  20.5% |  2,446.0 | main:serial.c:line.68
3||  13.9% |  1,663.0 | main:serial.c:line.60
3||  10.0% |  1,195.0 | main:serial.c:line.69
3||   2.2% |    265.0 | main:serial.c:line.59
3||   2.2% |    263.0 | main:serial.c:line.67
|||=====================================
||=======================================
|  19.8% |  2,366.0 | ETC
||--------------------------------------
||  19.7% |  2,350.0 | __ieee754_pow_sse2
3|        |          |  __ieee754_pow_sse2
||=======================================
|  10.8% |  1,289.0 | MATH
||--------------------------------------
||  10.8% |  1,289.0 | pow
3|        |          |  pow
|=======================================
```

CrayPat installed on ARCHER2 (details at www.archer2.ac.uk)

...	Source	CPU Time: Total	CPU Time: Self
48	for (i=1;i<=ny;i++) {		
49	bnorm=bnorm+pow(u_k[i+(j*mem_size_y)]*4-u_k[(i-1)+(j*mem_size_y)]-		
50	u_k[(i+1)+(j*mem_size_y)]-u_k[i+((j-1)*mem_size_y)]-u_k[i+((j+1)*mem_size_y)], 2);		
51	}		
52	}		
53	bnorm=sqrt(bnorm);		
54			
55	gettimeofday(&start_time, NULL);		
56	for (k=0;k<MAX_ITERATIONS;k++) {		
57	// Calculates the current residual norm		
58	for (j=1;j<=nx;j++) {	0.0%	0.040s
59	for (i=1;i<=ny;i++) {	2.0%	2.588s
60	rnorm=rnorm+pow(u_k[i+(j*mem_size_y)]*4-u_k[(i-1)+(j*mem_size_y)]-	50.2%	65.924s
61	u_k[(i+1)+(j*mem_size_y)]-u_k[i+((j-1)*mem_size_y)]-u_k[i+((j+1)*mem_size_y)], 2);	15.6%	20.480s
62	}		
63	}		
64			
65	// Do the Jacobi iteration		
66	for (j=1;j<=nx;j++) {	0.0%	0.064s
67	for (i=1;i<=ny;i++) {	2.1%	2.764s
68	u_kp1[i+(j*mem_size_y)]=0.25 * (u_k[(i-1)+(j*mem_size_y)]+u_k[(i+1)+(j*mem_size_y)]+	19.2%	25.189s
69	u_k[i+((j-1)*mem_size_y)]+u_k[i+((j+1)*mem_size_y)]);	10.9%	14.330s
70	}		
71	}		
72			
73	// Swap data structures round for the next iteration		
74	temp=u_kp1;		
75	u_kp1=u_k;		
76	u_k=temp;		
77			
78	norm=sqrt(rnorm)/bnorm;		
79	rnorm=0.0;		
80			
81	if (norm < convergence_accuracy) break;		
82	if (max_its > 0 && k >= max_its) break;		
83			
84	if (k % REPORT_NORM_PERIOD == 0) printf("Iteration= %d Relative Norm=%e\n", k, norm);		
85	}		
86	gettimeofday(&end_time, NULL);		

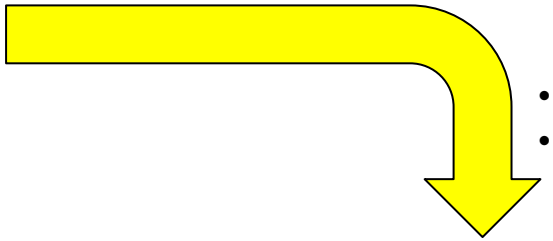
Intel's Vtune which is very popular and freely available for all architectures.

- Installed on Cirrus

Eliminating Loop-Carried Dependencies

- Loop iterations must be nearly independent
  - Remove dependencies where possible:
- Temporal dependencies
  - Replace iterative series with closed forms

```
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);
}
```

```
int a=0;
for (int i=0;i<N;i++) {
  a+=i;
}
```

- Spatial dependencies
  - Replicate, update and reduce
  - Explicit synchronisation to protect shared data, details in *Shared Data* pattern (later in the lecture)

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

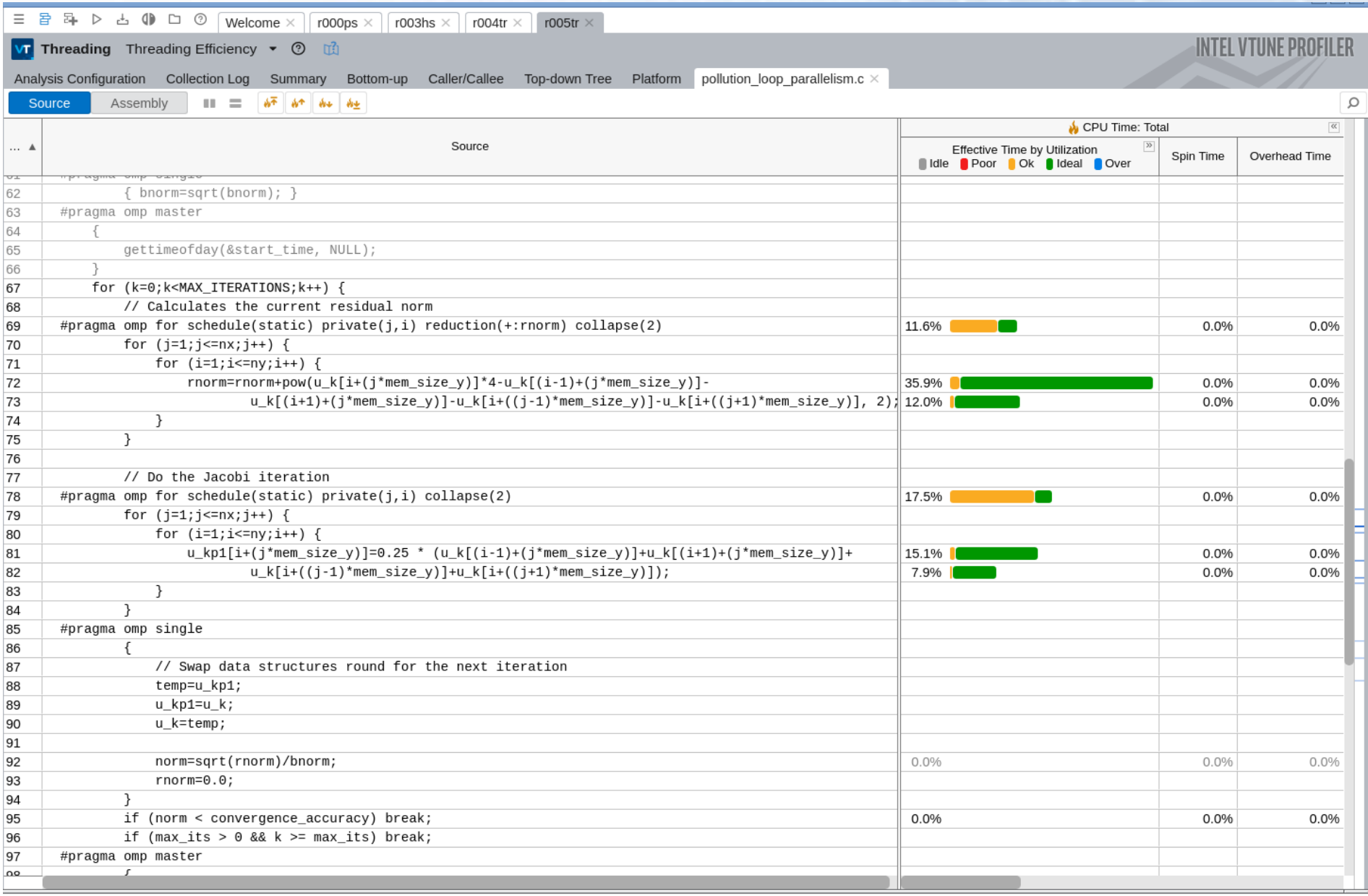
```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

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

    return 0;
}
```

Code example from http://en.wikipedia.org/wiki/OpenMP

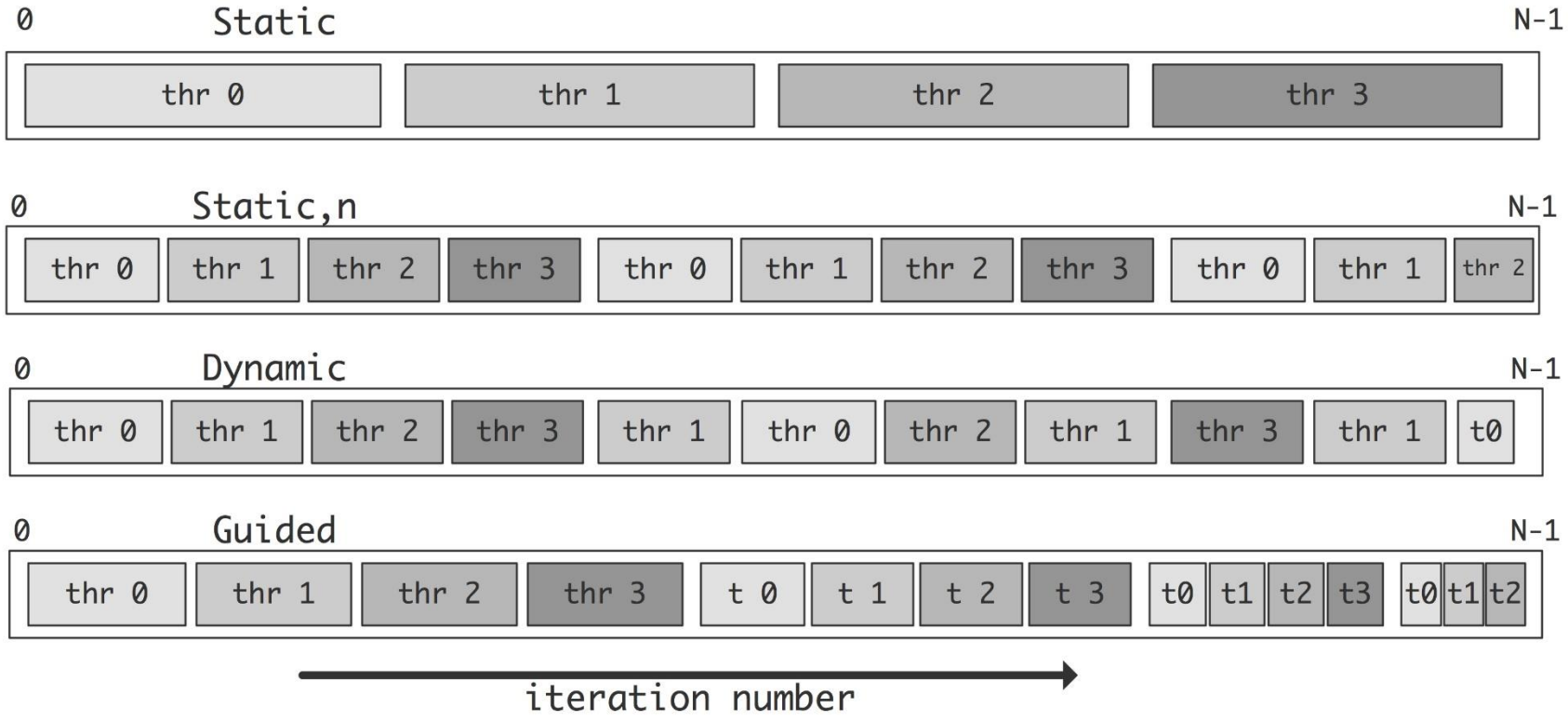
View via profiling



Optimising the loop schedule

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

- Again, this can be added incrementally
  - Often based on experimentation with the code



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);
  }
}
```

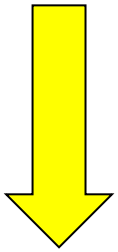
```
for (c=0;c<n1*n2;c++) {
  i=c/n1;
  j=c%n2;
  function_a(i,j);
}
```

Other Loop Optimisations

Stripmining

- Enables the use of vector or SIMD instructions

```
for (i=0;i<n;i++) {
   A[i]=B[i] + C[i]
}
```

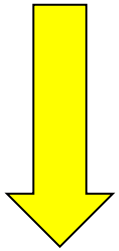


```
for (i=0;i<n;i+=B) {
   for (j=i;j<i+B;j++) {
      A[j]=B[j] + C[j]
   }
}
```

Interchange

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

```
for (j=0;j<n;j++) {
   for (i=0;i<n;i++) {
      A[i,j]=B[i,j]
   }
}
```

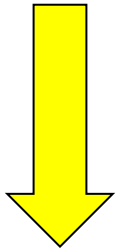


```
for (i=0;i<n;i++) {
   for (j=0;j<n;j++) {
      A[i,j]=B[i,j]
   }
}
```

Other loop optimisations

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

```
for (i=0;i<n;i++) {
   for (j=0;j<n;j++) {
      for (k=0;k<n;k++) {
         C[i,j] += A[i,k] * B[k,j]
      }
   }
}
```

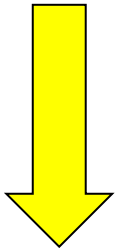


```
for (i=0;i<n;i+=B) {
   for (j=0;j<n;j+=B) {
      for (k=0;k<n;k+=B) {
         for (ii=i;ii<i+B;ii++) {
            for (jj=j;jj<j+B;jj++) {
               for (kk=k;kk<k+B;kk++) {
                  C[ii,jj] += A[ii,kk] * B[kk,jj]
}  }  }  }  }  }
```

Other Loop Optimisations

- Fission
  - Split the loop

```
for (i=0;i<n;i++) {
   for (j=0;j<n;j++) {
      A[i,j] = B[i,j] + C[i,j]
      D[i,j] = B[i,j] * 2
   }
}
```



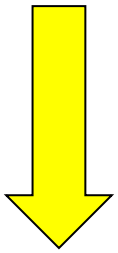
```
for (i=0;i<n;i++) {
   for (j=0;j<n;j++) {
      A[i,j] = B[i,j] + C[i,j]
   }
   for (j=0;j<n;j++) {
      D[i,j] = B[i,j] * 2
   }
}
```

Other Loop Optimisations

Unrolling

- Replicate body to reduce overhead

```
for (i=0;i<n;i++) {
    A[i]=B[i] + C[i]
}
```

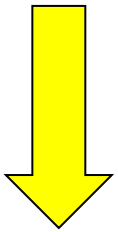


```
for (i=0;i<n;i+=4) {
    A[i]=B[i] + C[i]
    A[i+1]=B[i+1] + C[i+1]
    A[i+2]=B[i+2] + C[i+2]
    A[i+3]=B[i+3] + C[i+3]
}
```

Unroll and jam

- Unroll outer loop, merge copies of inner loop

```
for (i=0;i<n;i++) {
   for (j=0;j<n;j++) {
      A[i]=A[i] + B[j]
   }
}
```

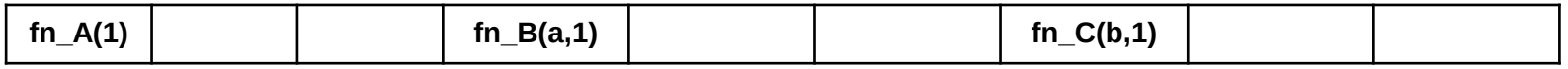


```
for (i=0;i<n;i+=4) {
   for (j=0;j<n;j++) {
      A[i]=A(i) + B(j)
      A[i+1]=A[i+1] + B[j+1]
      A[i+2]=A[i+2] + B[j+2]
      A[i+3]=A[i+3] + B[j+3]
   }
}
```

Software pipelining

```
for (i=0;i<n;i++) {
    a=fn_A(i);
    b=fn_B(a,i);
    c=fn_C(b,i);
}
```

Unpipelined

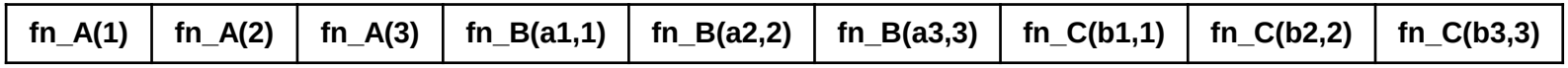


Assuming an instruction can be dispatched every cycle and each takes 3 cycles to complete

```
for (i=0;i<n;i+=3) {
    a1=fn_A(i);
    a2=fn_A(i+1);
    a3=fn_A(i+2);
    b1=fn_B(a1,i);
    b2=fn_B(a2,i+1);
    b3=fn_B(a3,i+2);
    c1=fn_C(b1,i);
    c2=fn_C(b2,i+1);
    c3=fn_C(b3,i+2);
}
```

Trying to maximise the utilisation of the processor's pipeline

Software pipelined



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

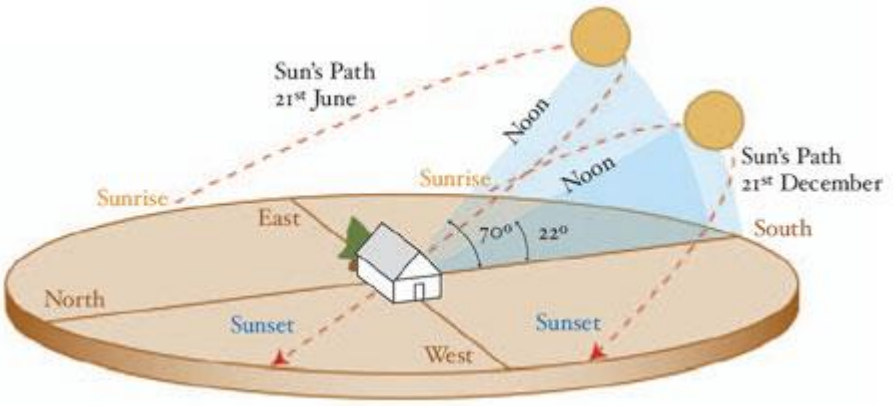
- Synonymous with OpenMP but other technologies possible too such as parallel iterators and *par* loops

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

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

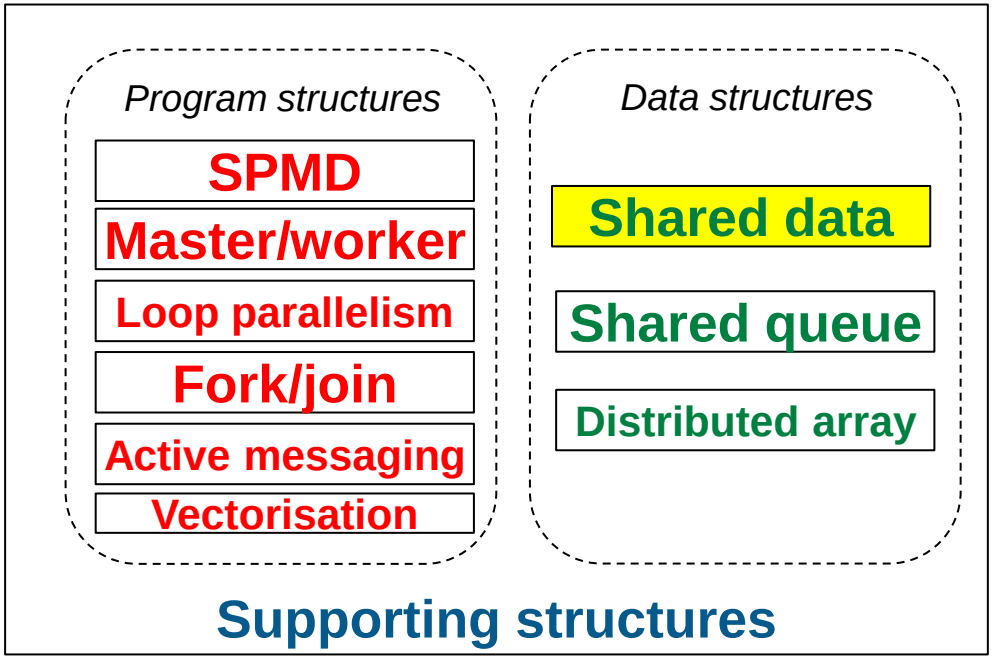
```
do i = 22 to 70
  do j = 1 to num_rays
    ……
  end do
end do
```

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

Shared Data: The Problem

- Shared Data is an *Implementation (Supporting structures) Pattern*



- The Problem: How does one explicitly manage shared data in a set of concurrent tasks?

Shared Data: Context

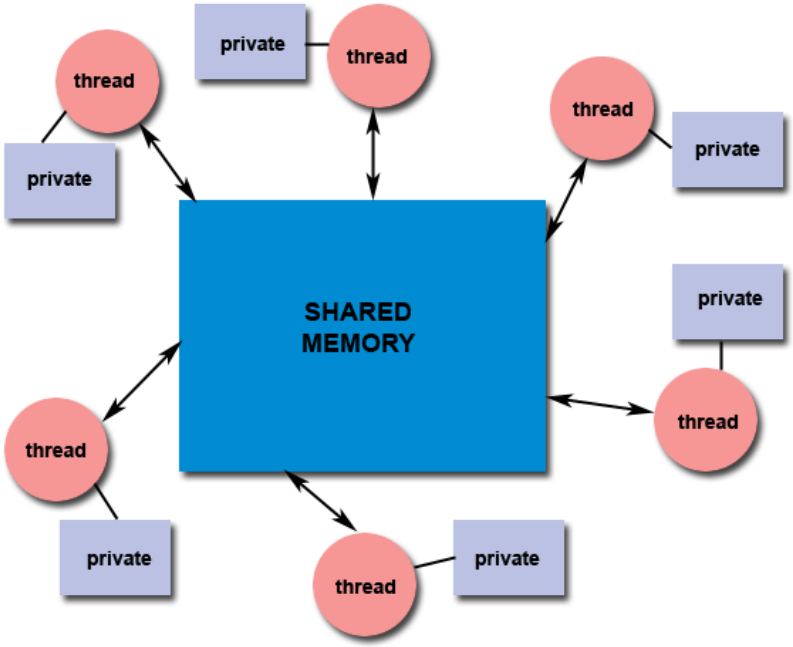
- Some parallel algorithm patterns provide particular means to deal with would-be shared data by *pulling* it out the task
  - Replication & Reduction with Task Parallelism
  - Halo-swapping with Geometric Decomposition

- Sometimes it's not possible to manage the sharing of data in this way, and it's necessary to explicitly incorporate mechanisms to manage shared state into the tasks themselves. This is when it is often useful to apply the *Shared Data* pattern
- Necessary, for example, in task parallelism when dependencies are neither removable or separable.

Shared Data: Context (2)

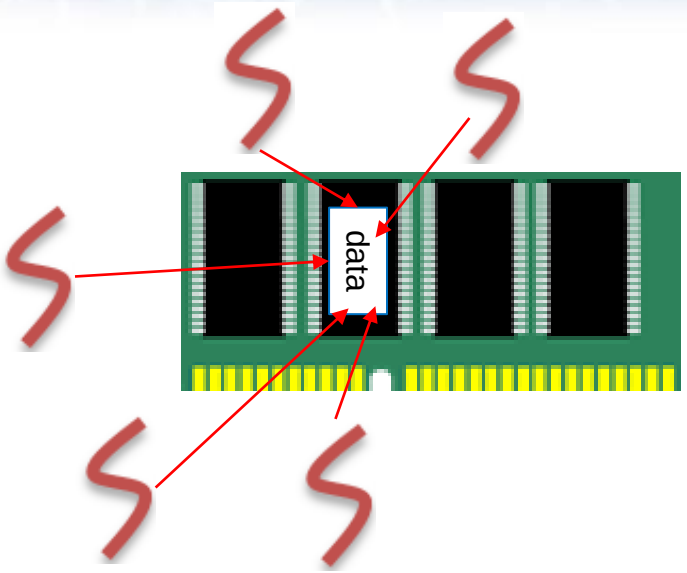
- Common attributes for problems that need the *Shared Data* pattern:
  - At least one data structure is accessed by multiple tasks in the course of the program's execution
  - At least one task modifies the shared data structure, and
  - The tasks potentially need to use the modified value during the concurrent computation

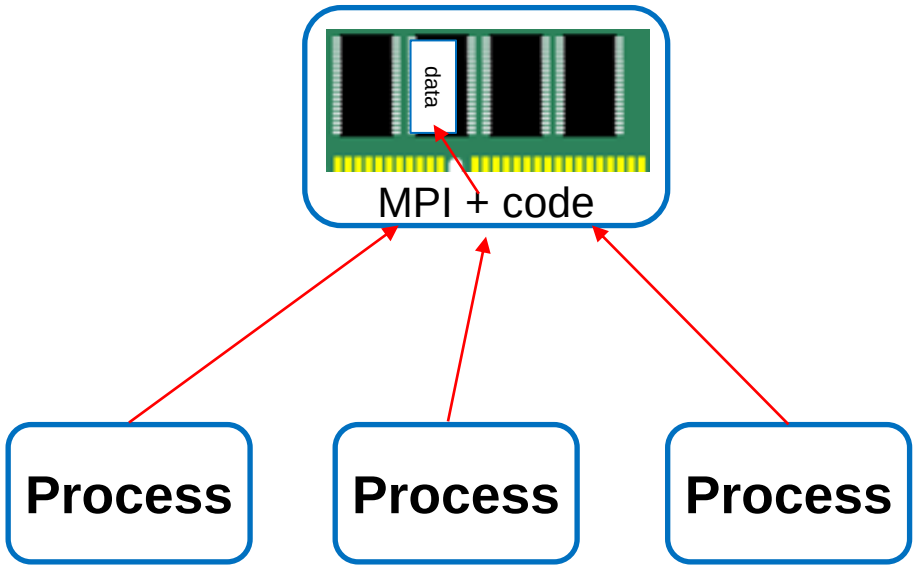
- Most commonly assume this is with shared memory (threaded programming) but can be required with distributed memory too



Shared Data: Forces

- The results of the computation must be correct for *any* ordering of the tasks that could occur during the computation
- Explicitly managing shared data can incur parallel overhead, which must be kept small if the program is to run efficiently





- Techniques for managing shared data can limit the number of tasks that can run concurrently, thereby potentially reducing scalability
- If the constructs used to manage shared data are not easy to understand, the program will be harder to maintain

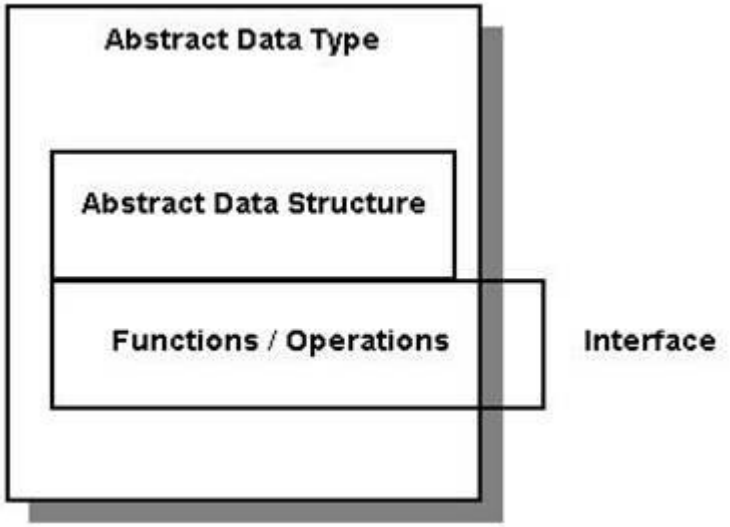
Solution

- Ensure this pattern is needed
  - By revisiting earlier decisions can we find an approach matching one of the algorithm strategy patterns without the need for shared data?

1. Make use of abstract data types (ADTs)
2. Implement appropriate concurrency-control protocol
   - One-at-a-time execution
   - Noninterfering sets of operations
   - Readers/Writers
   - Reducing the size of the critical section
   - Nested locks
   - Application-specific semantic relaxation
3. Review other considerations
   - Memory synchronisation
   - Task scheduling

Using an Abstract Data Type

- Consider the shared data type as an ADT with a *fixed set* of (possibly complex) operations on the data
  - e.g. for a shared queue, you might have *put*, *get*, *remove*, *isEmpty, getSize*



- Each task will typically perform a sequence of these operations, *along with operations on other (non-shared) data*
- Operations should have the property that they each leave the data in a consistent, meaningful state
- Implementation of individual operations should be such that lower-level actions should not be visible to other tasks/UEs

Concurrency Control Protocols

- Once you have defined an ADT and its operations, we need to ensure that the operations provide the same results as if they were executed in serial.

- One-at-a-time execution
  - The simplest approach, ensure operations indeed do execute in serial
  - Uses a Critical Section
    - Provided directly by language, or indirectly through mutex locks, synchronised blocks, OpenMP critical
  - Usually straightforward to implement, but often overly conservative resulting in bottlenecks.

```
function operation1 {
  synchronised {
    ……
  }
}
function operation2 {
  synchronised {
    ……
  }
}
function operation3 {
  synchronised {
    ……
  }
}
```

Concurrency Control Protocols

- Noninterfering sets of operations
  - Analyze the interference between operations, operation A *interferes with* operation B if A writes a variable that B reads or writes.
  - Maintain ***disjoint*** sets of interfering operations, where operations in different sets do not interfere.
  - Within each ***disjoint*** set operations execute one at a time, but operations in different sets can proceed concurrently

```
function operation1 {
  synchronised A {
    ……
  }
}
function operation2 {
  synchronised A {
    ……
  }
}
function operation3 {
  synchronised B {
    ……
  }
}
```

Concurrency Control Protocols

- Readers/Writers
  - If operations cannot be separated out but if some operations modify the data and others only read it then we can go from here.
  - If A is a writer (both modify and read) but B is reader (only read) then A interferes with itself and B, but B interferes with nothing.
  - Therefore if one task is performing A then no other task should be able to execute A or B. But any number of Bs can execute concurrently. *This is the basis for RW locks in pthreads*
  - Introduces some overhead, some thought needed by lock writers

```
function get {
  synchronise read {
    ……
  }
}
function put {
  synchronise write {
    ……
  }
}
function getSize {
  synchronise read {
    ……
  }
}
```

Concurrency Protocols

- Reducing the size of the critical section
  - Don't put the whole operation in a critical section
  - Analyze the operations in more detail, does only one aspect cause interference?
  - Very easy to get wrong, so be careful!
  - Repeated locking and unlocking can be expensive

```
function operation1 {
  synchronised {
    ……
  }
}
function operation2 {
  ……
  synchronised {
    ……
  }
  ……
}
function operation3 {
  ……
  synchronised {
    ……
  }
  ……
  synchronised {
    ……
  }
}
```

Concurrency Protocols

- Nested locks
  - A hybrid of noninterfering operations and reducing the CS size
  - If you have *almost* non-interfering operations, an extra lock can be placed around just the interfering part of the operation
  - If A reads and writes to x and y, and B reads and writes to y then strictly speaking these interfere. However, can place a lock around A's y access to allow for additional concurrency
  - Increased potential for deadlock

```
function operation1 {
  synchronised A {
    ……
    synchronised B {
      ……
    }
  }
}
function operation2 {
  synchronised B {
    ……
  }
}
function operation3 {
  synchronised B {
    ……
    synchronised A {
      ……
    }
  }
}
```

Concurrency Protocols

- Application specific semantic relaxation
  - e.g. partially replicate shared data, and don't keep all of the copies completely in sync
  - In some cases may involve a duplication of work (i.e. a number of tasks searching for an answer based upon the same starting conditions) but this can be more efficient than managing shared data to avoid this.
  - Application logic means that conflict can never happen in reality

```
function operation1 {
   ……
}
function operation2 {
   ……
}
function operation3 {
   ……
}
```

Other considerations

- Memory synchronisation
  - Caching and compiler optimisation can result in unexpected behaviour.
  - I.e. a stale value might be read from a cache or a new value not flushed to memory.
  - In OpenMP there is a flush directive which is invoked by several other directives (such as after a for, critical, single, barrier.)
  - In Java memory is explicitly synchronised when entering and leaving synchronised blocks, when locking and unlocking locks and all variables marked with *volatile*.
  - In C or FORTRAN have the *volatile* keyword too, often needed!
- Task scheduling
  - Will a task be idle, waiting for access to some shared data?
  - If so can we assign tasks to UEs in such a way that minimises this?
  - Or can we assign multiple tasks to UEs such that there is always one that is not waiting and doing some work?

Shared data: Summary

- First consider if you really have to use this
- Make use of Abstract Datatypes
- Carefully consider the appropriate concurrency protocol
  - Trade off between simplicity and performance?
- Can I do other things (such as clever task scheduling) to minimise the impact this will have?