Durham University
Department of Computer Science

Core I—Introduction to HPC

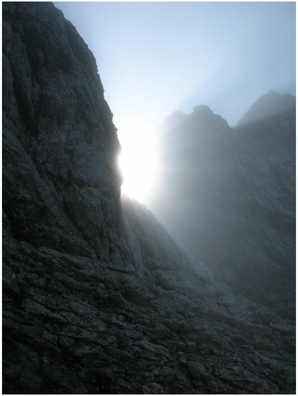
Session VI: OpenMP (2/2)

Dr. Weinzierl

Michaelmas term 2019

Outline

Durham University
Department of Computer Science



Reduction
Loop scheduling
Tasks

Thread communication—repeated

Durham University
Department of Computer Science

We distinguish two different communication types:

- Communication through the join
- Communication inside the BSP part (not “academic” BSP)

> Critical section: Part or code that is ran by at most one thread at a time.

> Reduction: Join variant, where all the threads reduce a value into one single value.

- $\Rightarrow$ Reduction maps a vector of $(x_0, x_1, x_2, x_3), \ldots, x_{p-1}$ onto one value x, i.e. we have a all-to-one data flow
- $\Rightarrow$ Inter-thread communication realised by data exchange through shared memory
- $\Rightarrow$ Fork can be read as one-to-all information propagation (done implicitly by shared memory)

Collective operations

Durham University
Department of Computer Science

Collective operation: A collective operation is an operation that involves multiple cores/threads.

- Any synchronisation is a collective operation
- BSP/OpenMP implicitly synchronises threads, i.e. we have used synchronisation
- Synchronisation however does not compute any value

```
double result = 0.0;
#pragma omp parallel
{
  double myResult = 0.0;
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    myResult += a[i] * b[i];
  }
  #pragma omp barrier
}
```

- The above fragment synchronises all threads
- This type is a special type of a *collective operation*
- Barriers are implicitly inserted by BSP programming model

Collective operations

Durham University
Department of Computer Science

> Collective operation: A collective operation is an operation that involves multiple cores/threads.

- Challenge: Synchronisation does not compute any value

```
double result = 0.0;
#pragma omp parallel
{
  double myResult = 0.0;
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    myResult += a[i] * b[i];
  }
  #pragma omp critical
  result += myResult;
}
```

- The above fragment mimics an all-to-one operation (all threads aggregate their data in the master's result variable)
- This type is called *reduction* which is a special type of a *collective operation*
- OpenMP provides a special clause for this

Reduction


```
double result = 0;
#pragma omp parallel for reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

```
double result = 0;
#pragma omp parallel for private(result) reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

```
double result = 0;
#pragma omp parallel for firstprivate(result) reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

- Which variant is correct?

Reduction

Durham University
Department of Computer Science

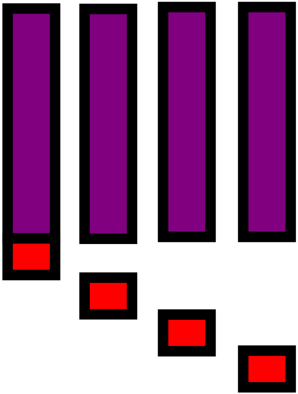
```
double result = 0;
#pragma omp parallel for reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

```
double result = 0;
#pragma omp parallel for private(result) reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

```
double result = 0;
#pragma omp parallel for firstprivate(result) reduction(+:result)
for( int i=0; i<size; i++ ) {
  result += a[i] * b[i];
}
```

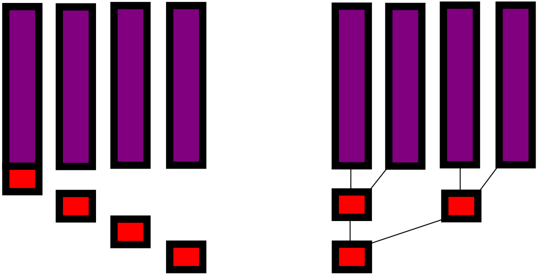
- Which variant is correct?
- Reduction keyword is parameterised with (commutative) operation (+,-,*,&,|,. . .)
- Reduction works solely for scalars.
- Keyword makes scalar private first but then merges the threads' scalars.
- Keyword initialises private copy with identity w.r.t. the operator.

Performance study



```
double result = 0.0;
#pragma omp parallel
{
  double myResult = 0.0;
  #pragma omp for
  for( int i=0; i<size; i++ ) {
    myResult += a[i] * b[i];
  }
  #pragma omp critical
  result += myResult;
}
```

Performance study

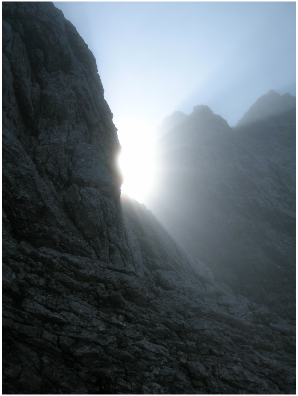


Concept of building block

- Content
  - Introduce term collective
  - Introduce reduction syntax
  - Study (potential) impact of reduction feature
- Expected Learning Outcomes
  - The student *knows* reductions in in OpenMP and its variants
  - The student can *identify* reductions in given codes
  - The student can *program* with reductions

Outline

Durham University
Department of Computer Science



Reduction
Loop scheduling
Tasks

Remaining agenda

Durham University
Department of Computer Science

Starting point:

- Data analysis allows us to identify candidates for data parallelism/BSP
- Concurrency analysis plus speedup laws determine potential real-world speedup
- OpenMP allows us to annotate serial code with BSP logic

Open questions:

- How is work technically split?
- How is work assigned to compute cores?
- What speedup can be expected in practice?

Scheduling: Assign work (loop fragments) to threads.

Pinning: Assign thread to core.

Technical remarks

Durham University
Department of Computer Science

On threads:

- A thread is a logically independent application part, i.e. it has its own call stack (local variables, local function history, . . .)
- All threads share one common heap (pointers are replicated but not the area they are pointing two)
- OpenMP literally starts new threads when we hit `parallel for` $\Rightarrow$ overhead
- OpenMP hides the scheduling from user code

On cores:

- Unix cores can host more than one thread though more than two (hyperthreading) becomes inefficient
- Unix OS may reassign threads from one core to the other $\Rightarrow$ overhead
- Unix OS can restrict cores-to-thread mapping (task affinity)
- Unix OS can be forced to keep cores-to-thread mapping (pinning)

Grain size

Durham University
Department of Computer Science

Grain size: Minimal size of piece of work (loop range, e.g.).

- Concurrency is a theoretical metric, i.e. machine concurrency might/should be smaller
- Multithreading environment thus wrap up multiple parallel tasks into one job
- Grain size specifies how many tasks may be fused

Technical mapping of tasks

- Each thread has a queue of tasks (jobs to do)
- Each job has at least grain size
- Each thread processes tasks of its queues (dequeue)
- When all task queues are empty, BSP joins

Static scheduling

Durham University
Department of Computer Science

Definition:

1. Cut problem into pieces (constrained by prescribed grain size)
2. Distribute work chunks among queues
3. Disable any work stealing

In OpenMP:

- Default behaviour of `parallel for`
- Trivial grain size of 1 if not specified differently
- Problem is divided into `OMP_NUM_THREADS` chunks ⇒ at most one task per queue

```
#pragma omp parallel for schedule(static,14)
...
```

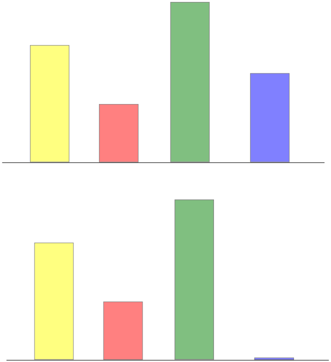
Properties:

- Overhead
- Balancing
- Flexibility w.r.t. inhomogeneous computations

Work stealing

Durham University
Department of Computer Science

Work stealing: When one thread runs out of work (work queue become empty), it tries to grab (steal) work packages from other threads.



Dynamic scheduling

Durham University
Department of Computer Science

Definition:

1. Cut problem into pieces of prescribed grain size
2. Distribute all work chunks among queues
3. Enable work stealing

In OpenMP:

- To be explicitly enabled in `parallel for`
- Trivial grain size of 1 if not specified differently
- Set of chunks is divided among `OMP_NUM_THREADS` queues first

```
#pragma omp parallel for schedule(dynamic)
...
#pragma omp parallel for schedule(dynamic,14)
...
```

Properties:

- Overhead
- Balancing
- Flexibility w.r.t. inhomogeneous computations

Guided scheduling

Durham University
Department of Computer Science

Definition:

1. Cut problem into chunks of size N/p constrained by grain size and with $p =$OMP_NUM_THREADS
2. Cut remaining tasks into pieces of $(N/(2p))$ constrained by grain size
3. Continue cut process iteratively
4. Distribute all work chunks among queues; biggest jobs first
5. Enable work stealing

In OpenMP:

- To be explicitly enabled in `parallel for`
- Trivial grain size of 1 if not specified differently
- Set of chunks is divided among `OMP_NUM_THREADS` queues first

```
#pragma omp parallel for schedule(guided)
...
#pragma omp parallel for schedule(guided,14)
...
```

Properties:

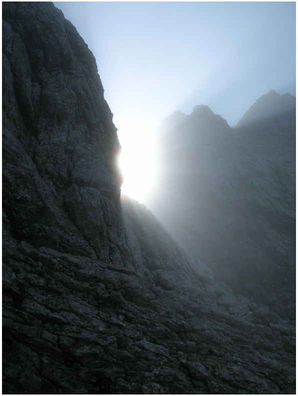
- Overhead
- Balancing
- Flexibility w.r.t. inhomogeneous computations

Concept of building block: Loop scheduling

Durham University
Department of Computer Science

- Content
  - Introduce terminology
  - Discuss work stealing
  - Study OpenMP's scheduling mechanisms
  - Study OpenMP's two variants of dynamic scheduling
  - Conditional parallelisation
- Expected Learning Outcomes
  - The student **knows** technical terms tied to scheduling
  - The student can **explain** how work stealing conceptually works
  - The student can **identify problems** arising from poor scheduling/too small work packages
  - The student can **use** proper scheduling in OpenMP applications

Outline



Reduction
Loop scheduling
Tasks

If not all threads shall do the same

Durham University
Department of Computer Science

A manual task implementation:

```
#pragma omp parallel for schedule(static:1)
for (int i=0; i<2; i++) {
  if (i==0) {
    foo();
  }
  else {
    bar();
  }
}
```

Shortcomings:

- Syntactic overhead
- If `bar` depends at one point on data from `foo`, code deadlocks if ran serial
- Not a real task system, as two tasks are synchronised at end of loop

Tasks in OpenMP 3.0

Durham University
Department of Computer Science

- Introduce new task keyword:

```
#pragma omp task
```

- Parallel regions sets up queues:

```
#pragma omp parallel
```

- All tasks that are parallel to each other befill this queue
- Before OpenMP 3.0 there used to be a

```
#pragma omp section
```

  command. The standard does not specify, whether sections can be stolen.
- Tasks may spawn additional tasks

Task example

Durham University
Department of Computer Science

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      printf( "Task A" );
      printf( "we stop Task A now" );
      #pragma omp task
      {
        printf( "Task A.1" );
      }
      #pragma omp task
      {
        printf( "Task A.2" );
      }
      #pragma omp taskwait
      printf( "resume Task A" );
    }
    #pragma omp task
    {
      printf( "Task B" ); }
    #pragma omp taskwait
    #pragma omp task
    printf( "Task A and B now have finished" );
  }
}
```

Task communication

- Tasks may communicate through shared memory
- Critical sections remain available

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
      foo();
    #pragma omp task
      bar();
  }
}
```

Observations:

- OpenMP pragmas are ignored if we compile without OpenMP

⇒ Code still deadlocks if `bar` depends on

⇒ Should work with `OMP_NUM_THREADS=1`

- There is still an implicit join where `omp parallel` terminates

⇒ Task paradigm is embedded into fork-join model

Task example

Durham University
Department of Computer Science

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      printf( "Task A" );
      printf( "we stop Task A now" );
      #pragma omp task
      {
        printf( "Task A.1" );
      }
      #pragma omp task
      {
        printf( "Task A.2" );
      }
      #pragma omp taskwait
      printf( "resume Task A" );
    }
    #pragma omp task
    {
      printf( "Task B" ); }
    #pragma omp taskwait
    #pragma omp task
    printf( "Task A and B now have finished" );
  }
}
```

Can you draw the task dependency graph? It is the “inverse” of the spawn graph!

Concept of building block

Durham University
Department of Computer Science

- Content
  - Introduce task parallelism
  - Discuss task features compared to “real” tasking systems
- Expected Learning Outcomes
  - The student *can analyse* OpenMP’s task concept and upcoming features
  - The student *can write* a task-based OpenMP code