HUST-USYD Summer School on Parallel Programming Practice – Lecture 7

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

THE UNIVERSITY OF SYDNEY

Outline

- Review of Homework 4
- Support general data structures beyond for loops
  - Sections
  - Tasks
- Summary
  - Parallel algorithm design for shared-memory machines
  - OpenMP programming
- Parallel algorithm design for distributed-memory machines
- Homework 5

Work-Sharing Construct: Section Directive

- So far we have learnt OpenMP directive to parallelize for loops
- To parallelize code rather than for loops we may use sections directive:

```
#pragma omp sections [clause ...]  newline
{
  #pragma omp section
  structured_block;
  #pragma omp section
  structured_block;
  #pragma omp section
  structured_block;
}
```

- In the sections construct, each section is executed by one thread
- By default, there is a barrier at the end of the omp sections
- Use nowait clause to turn off the barrier
- The sections directive must be inside a parallel region

Work-Sharing Construct: Section Directive

– E.g., a sequential code:

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
z = zita(x, y);
```

Functions alpha and beta can be executed in parallel

Functions gamma and delta can be executed in parallel

– The OpenMP code with sections directive:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      v = alpha();
      #pragma omp section
      w = beta();
   }
   #pragma omp sections
   {
      #pragma omp section
      x = gamma(v, w);
      #pragma omp section
     y = delta();
   }
  z = zita(x, y);
}
```

Structures beyond for Loops

– Consider a simple linked list traversal:

```
node *p = head;
while (p)
{
   process(p);
   p = ->next;
}
```

– Note the OpenMP loop work-sharing construct only works with loops for which the number of loop iterations can be represented by a closed form expression at compiler time
  – while loops are not supported
– In practice, however, sometimes we do need to parallelize while loops (and other general data structures)
  – How?

Structures beyond for Loops

– One solution is to count the number of nodes in the list, copy pointer to each node into an array, and then use OpenMP for construct

```
p = head; count = 0;
while (p){
   parr[count] = p;
   p = ->next;
   count++;
}
#pragma omp parallel
{
  #pragma omp for schedule(static, 1)
  For {int i=0; i<count; i++)
     process(parr[i]);
}
```

Copy pointer to each node into an array

Count the number of node in the list

Process nodes in parallel with a for loop

– Require multiple passes over the data
– Can we do it more neatly?
  – Use OpenMP task construct

OpenMP firstprivate Clause

- Before discussing OpenMP task construct, we introduce one more OpenMP data environment clause firstprivate
- When using private(list) clause, a local copy of list in the private clause is made to each thread, but the value is not initialized
- E.g.,

```
int tmp = 0;
#pragma omp parallel for private(tmp)
for (int i=0; i<n; i++)
   tmp += i;
printf("%d\n", tmp);
```

tmp was not initialized

tmp is 0 here

OpenMP firstprivate Clause

- Using firstprivate clause private copy will be initialized from the shared variable
- E.g.,

```
int tmp = 0;
#pragma omp parallel for firstprivate(tmp)
for (int i=0; i<n; i++)
   tmp += i;
printf("%d\n", tmp);
```

Each thread gets its own copy of tmp with an initialized value of 0

OpenMP Task Construct

- #pragma omp task
- Tasks are independent units of work
- Tasks are composed of
  - code to execute
  - data environment
  - internal control variables (ICV)
- Threads perform the work of each task
- The runtime system decides when tasks are executed
- The basic idea is to set up a task queue
  - When a thread encounters a task directive, it packages a new instance of a task and then continue
  - Some thread executes the task at some later time

OpenMP Task Construct

– Use OpenMP task construct for the linked list traversal problem:

```
#pragma omp parallel
{
  #pragma omp single
  {
    p = head;
    while (p) {
      #pragma omp task firstprivate(p)
        process(p);
      p = p->next;
    }
  }
}
```

1. Create a team of threads

2. One thread executes the single construct

Other threads wait at the implied barrier at the end of single construct

3. The single thread creates a task with a different value p each time

single用于单独创建一个task，task用多少线程与single无关

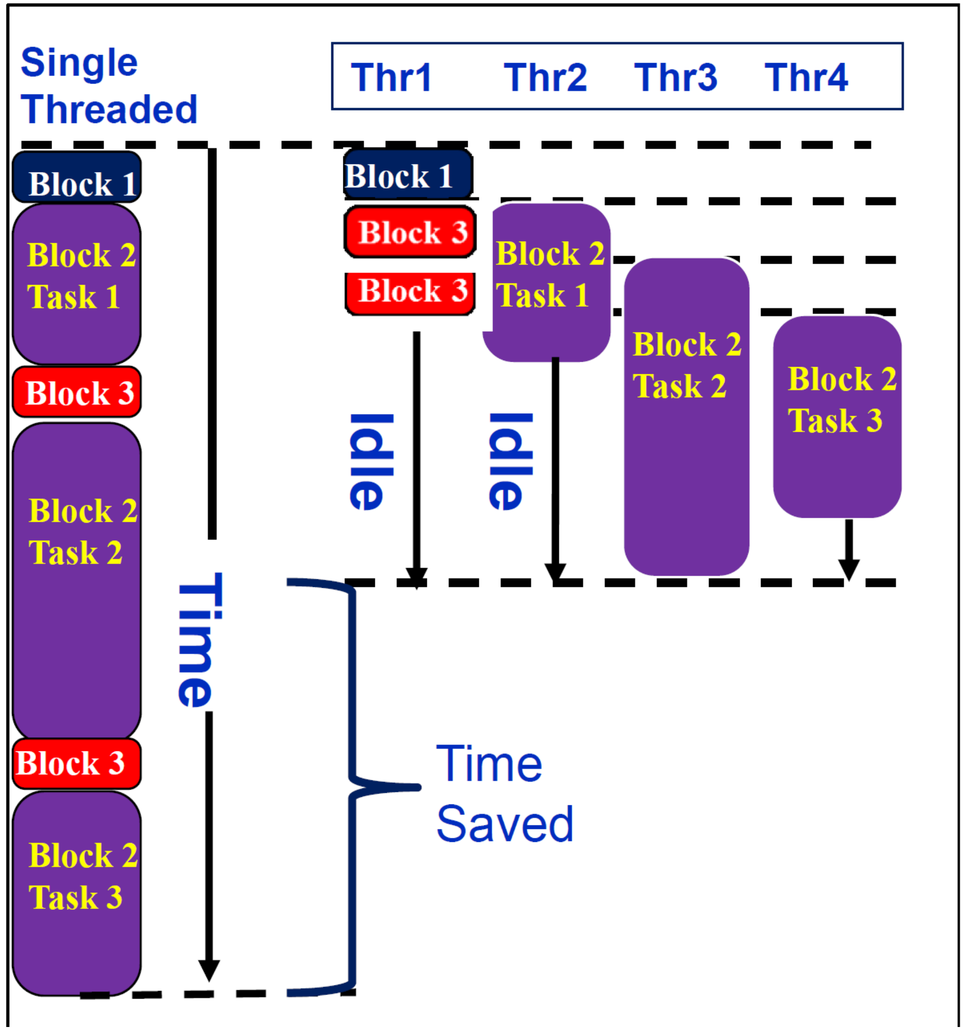
task直接进任务队列

4. Threads waiting at the barrier execute tasks

Execution moves beyond the barrier once all tasks are completed

OpenMP Task Construct

```
#pragma omp parallel
{
  #pragma omp single
  {   //block 1
     node * p = head;
     while (p) { // block 2
     #pragma omp task
       process(p);
     p = p->next;   //block 3
     }
  }
}
```



Summary

- In a shared memory machine
  - Global data are shared by all threads
  - Data don’t need to be moved around when used by different threads
    - Task partitioning and assignment is relatively simpler than that for distributed memory machines
  - Threads coordination must be done explicitly by synchronization on shared variables
    - Must applied properly
    - very important to minimize synchronization overheads
  - Locality is very important
    - Optimize the performance on single processor, or core
    - Increase computational intensity (memory hierarchy)

Summary

- OpenMP:
  - A directive-based Application Programming Interface (API) for developing parallel programs on shared memory architectures
  - Only a small API that hides cumbersome threading calls with simpler directives
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than explicitly create concurrently-executing threads
  - Provide some synchronization constructs

Summary

- OpenMP directives, clauses and functions:
  - To create a team of threads
    - #pragma omp parallel
  - To share work between threads:
    - #pragma omp for
    - #pragma omp single
    - #pragma omp sections
    - #pragma omp task
  - To prevent conflicts (prevent data races)
    - #pragma omp critical
    - #pragma omp atomic
    - #pragma omp barrier
  - Data environment clauses
    - private (list)
    - firstprivate (list)
    - shared (list)
    - reduction(op:list)

Summary

- OpenMP directives, clauses and functions:
  - Loop schedule clauses
    - schedule(static, chunk)
    - schedule(dynamic, chunk)
    - collapse(n)
  - Disable implied barrier
    - nowait
  - Set/get environment variables functions
    - omp_set_num_threads()
    - omp_get_num_threads()
    - omp_get_thread_num()
    - omp_get_procs()
  - Get clock time function
    - omp_get_wtime()
  - … more can be found in openmp.org

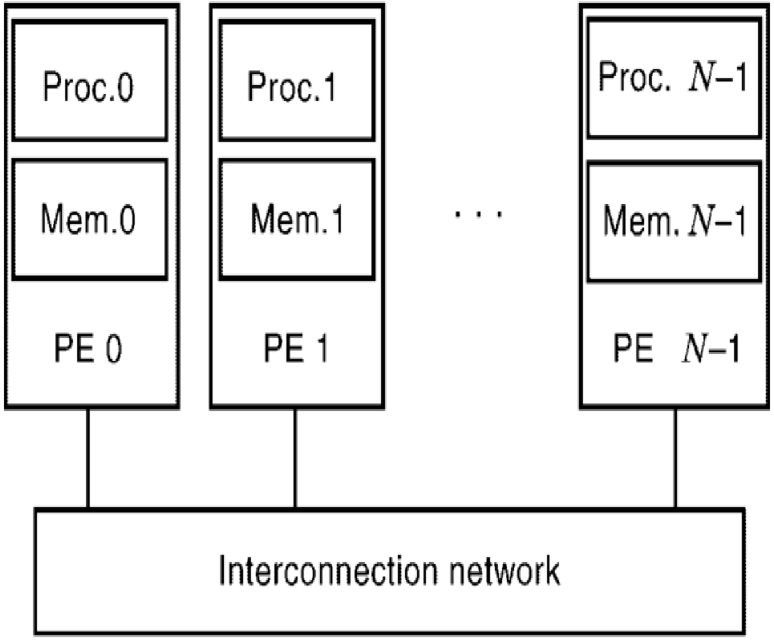
Summary

– It should be noted that OpenMP WILL NOT
  - parallelize automatically
  - guarantee speedup
  - provide freedom from data races

– To write an OpenMP program, normally include the following steps:
  – Parallel algorithm design
  – Write a sequential program accordingly
  – Optimize sequential program
  – Add necessary omp directives, clauses, functions
  – Test and tuning for performance
– The above steps may be repeated
– Also try other possible algorithms

Distributed-Memory Platform

- A distributed-memory platform comprises of a set of processing elements (or computing nodes)
- Each processing element has its own (exclusive) memory which cannot be accessed by other processing elements (no shared variables between processes)
- Processing elements communicate explicitly using (variants of) send and receive primitives
- Popular libraries such as MPI and VPM provide such primitives for process communication



Distributed-Memory Platform

- In parallel programming for distributed-memory platform
  - Data must be partitioned, distributed and kept local to each process explicitly
  - Decomposition of data determines assignment of work
  - For blocking send and receive (processes are blocked till data received) a send/receive pair serves as a synchronization point. Thus synchronization is implicit
  - A novice might feel it is much harder to write a parallel program for distributed-memory platform than for shared-memory platform
- Nowadays it is very common for a computer cluster to have multiple computing nodes, each being a multicore processor
- We need to use shared-memory programming within a multicore node of a cluster, and message passing between the nodes

Distributed-Memory Platform

- Since data need to be distributed across the processors, in the algorithm design and implementation we need seriously consider
  - How data is partitioned and distributed
  - How tasks (or work) are assigned which is also related to data partitioning
  - How to minimize communication overheads
  - Data locality (to reduce communication overheads)
  - Of course, how to balance the workloads

- For shared memory machines, the task assignment is relatively simple because global data are not specifically owned by any thread
  - Give us great flexibility for task assignment

Homework 5

- Write an OpenMP program for Gaussian elimination with partial pivoting
- Also need to optimize the performance on single core by using loop unrolling with unrolling factor = 4

THE UNIVERSITY OF SYDNEY