MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 9 - OpenMP & MPI

Graham Riley

Today's Lecture

- Dividing work to be done in parallel between threads in Java (as you are doing in the labs) is rather awkward, especially if you want to do it more than once in a program
- Many programs requiring parallelism are not written in Java anyway – but Fortran or C or C++!
- There are two standard parallel programming notations already in widespread use:
  - OpenMP = Open MultiProcessing
  - MPI = Message Passing Interface

OpenMP

- Used to create parallel programs for shared memory computers
- Collection of compiler directives and library functions
- http://www.openmp.org
- Based on fork-join model (i.e. very similar to the way we wrote Java earlier)
- Goals (besides parallel performance):
  - Sequential equivalence
  - Incremental parallelism

Parallel Regions

- Having identified part of a sequential program which would benefit, annotate it so that multiple threads execute in this *parallel region* and join up again at the end
- Start the region with a pragma (in C) or a comment (in Fortran)

```
#pragma omp parallel
```

Then the following code block is parallel

Fortran Aside

Fortran is not block-structured so it uses:

```
!$OMP PARALLEL
…
…
…
!$OMP END PARALLEL
```

This issue crops up with other features – and is dealt with similarly. We will stick with C.

Shared and Private Variables

- As you would expect, variables declared before the parallel region are **shared** (by default – can be controlled in the pragma), whereas those declared within it are **private** (local to thread)
- Following example shows this, and how to specify the number of threads to be used (one way)
- Also note that the code block following a pragma must be a structured block – which means it has a single point of entry (at top) and a single point of exit (at the bottom)
- There is an implicit barrier at the end

Example

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int j = 42 ; // shared variable
#pragma omp parallel num_threads(3)
   {
      int c ; // private to each thread
      c = omp_get_thread_num() ;
      printf("c = %d, j = %d\n", c, j) ;
    }
}
```

Output from Example

c = 0, j = 42

c = 2, j = 42

c = 1, j = 42

Note: the order of the threads' outputs is not defined!

Worksharing

- Typically don't want all threads to do the same thing. Instead want them to share work between them.
- The most commonly used form of this is loop-splitting:
  - Find a time consuming loop
  - Restructure, if necessary, to make iterations independent of each other
  - Split parts of the 'iteration space' to different threads
    - confusingly, this is known as (loop) **scheduling** – there are many different ways of doing it

Loop-splitting Example

```
double answer, res ;
answer = 0.0 ;
for (j = 0 ; j < N ; j++) {
      res = big_comp(j) ;  // big_comp takes a lot of time
      combine(answer, res) ;  // combine fairly quick
       }
```

To decouple loop iterations, one way is to make res an array (or declare it locally or private), and then we can compute all the res values in parallel

Split-loop Version

```
double answer, res[N] ; // N hash-defined earlier
answer = 0.0 ;
#pragma omp parallel for
  { // If we don't specify num_threads, a default number is used (from the
environment variable OMP_NUM_THREADS)
    // loop scheduling scheme also defaults (to a block schedule)
        for (j = 0 ; j < N ; j++) { // Special case: j is NOT shared!
           res[j] = big_comp(j) ;
        }
  }
  for (j = 0 ; j < N ; j++) combine(answer, res[j]) ;
```

Reduction Operations

- A common case is to use the result of each iteration of a loop in a binary associative operator – e.g.

  \+ * – & | max min

  (last two not in C)
- The order of application does not matter
- Initial value to use is implied by the operator
- OpenMP has a **reduction** clause

Reduction Example

```
double x, pi, step, sum = 0.0 ;
step  = 1.0/(double) num_steps ;
#pragma omp parallel for private(x) reduction(+:sum)
   for (j = 0 ; j < num_steps ; j++)
      {  x = (j + 0.5) * step ;
         sum += 4.0/(1 + x * x) ;
       }
   pi = step * sum ;
```

This computes pi by integrating $4/(1+x^2)$ between 0 and 1 using the "midpoint" (or rectangle) rule.

Synchronization Constructs

- **flush** – used to ensure memory consistency between threads!
- **critical** – indicates a critical section (for mutual exclusion) – can be named to provide multiple "locks"
- **barrier** – can be added explicitly
- These are all done using pragmas – example follows
- Also locks – done by library routines
  - omp_init_lock(&L), omp_set_lock(&L), omp_unset_lock(&L)

```
#pragma omp parallel for private(res)
for (j = 0 ; j < N ; j++) {
    res = big_comp(j) ;
#pragma omp critical
    combine(answer, res) ;
  }
```

This is how to do earlier example if combine isn't short – but wouldn't be safely done in parallel.

Loop Scheduling

Users can affect the scheduling of loop-iterations to threads (i.e. which iterations to give to a thread) and whether to do this statically or dynamically – e.g.

```
#pragma omp parallel for schedule(dynamic,10)
```

indicates that the iteration space should be broken into chunks of size = 10 iterations, and each thread should take one chunk at a time and come back for more when it has finished

A static allocation divides the chunks evenly between the threads at the start – this is a mistake if iterations vary considerably in required computation …

Static Loop Scheduling

- There are three commonly used static schedules:
  - **Block schedule** – divide the iterations into contiguous chunks of (near) equal size – default in OpenMP
  - **Cyclic schedule** – give one iteration to each thread in turn, returning to the first when you run out of threads – useful when the work per iteration changes in an orderly fashion
  - **Block-cyclic schedule** – give a fixed size chunk of iterations to each thread in turn, returning to the first when you run out of threads – useful for using the cache in a 'friendly' way

MPI

- MPI can be used to program a wide range of architectures
  - Distributed memory machines – i.e. clusters
  - MPPs
  - Shared memory machines
- Library for C & Fortran (& others)
- Most widely used API for parallel programming

Messages, Processes, Communicators

- Messages are identified by (rank of) sending process, receiving process, and integer tag
- These are defined with a communication context – to avoid confusion (so same identifier can be reused in library code, for example, in a different context)
- Processes are grouped – initially within a single group, but can split later
- Communication context plus process group forms a **communicator** (e.g. MPI_COMM_WORLD)

MPI "Hello World"

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char ** argv) {
  int num_procs ; // number of processes
  int ID ; // will be in range 0 <= ID < num_procs
  if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(-1) ;
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs) ; // ask for value
  MPI_Comm_rank(MPI_COMM_WORLD, &ID) ; // ask which one am I !
  printf("Hello  World from process %i of %i\n", ID, num_procs) ;
  MPI_Finalize() ;
}
```

‘Blocking’ Send and Recv

- int MPI_Send(buff, count, MPI_type, dest, tag, COMM) ;
- int MPI_Recv(buff, count, MPI_type, source, tag, COMM, &stat) ;
  - buff is a pointer to a buffer
  - count is #items of type MPI_type
  - MPI_type – e.g. MPI_DOUBLE, MPI_INT, ….
  - dest – the rank of the receiving process
  - source – the rank of the sending process (could be MPI_ANY_SOURCE in Recv)
  - tag – integer. In Recv could be MPI_ANY_TAG.
  - COMM – the MPI communicator (e.g. MPI_COMM_WORLD)
  - stat – a structure holding status information
- Send can also be non-blocking – implementation dependent – you can also allocate an explicit buffer to make it non-blocking

Collective Operations

- Besides various send and receive operations there are collective operations, e.g.
  - MPI_Barrier – barrier synchronisation
  - MPI_Bcast – broadcast
  - MPI_Scatter – broadcast with patterns of data movement
  - MPI_Gather – inverse of ‘scatter’
  - MPI_Reduce – a reduction operation
- Using such operations, many programs make little use of more primitive Send and Recv operations

Other MPI Features

- Asynchronous – i.e. non-blocking – communication MPI_Isend and MPI_Irecv
- With these, you need MPI_Wait and/or MPI_Test
- Persistent communication
- Modes of communication
  - Standard
  - Synchronous
  - Buffered
- MPI also supports “one-sided” messaging
  - MPI_Put(), MPI_Get()

Next Lecture

- Using locks is inherently pessimistic: to be safe, anything which could cause a problem if executed in parallel is made to execute sequentially. You pay a high price for what might actually be very rare.
- So why not be optimistic? Speculate – i.e. do things which might not be right, and then check if you need to throw them away (hoping this happens only occasionally)