西北工業大學

Parallel Computing

Exercise

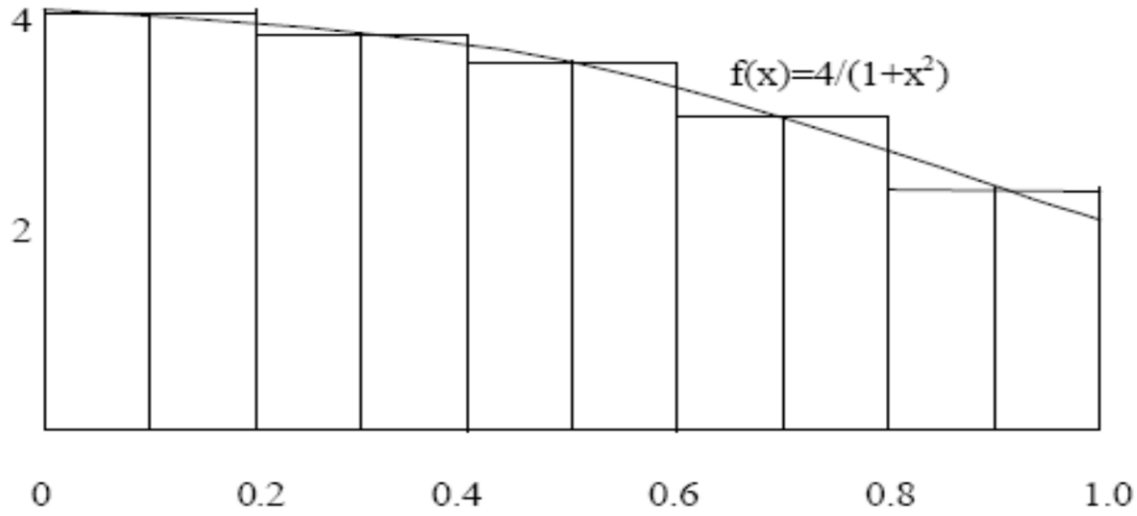
Outline

- Exercises 1 : Serial PI Program
- Exercise 2: first OpenMP PI program
- Exercise 3: Pi with a loop and a reduction
- Exercise 4: Using a critical section to remove impact of false sharing
- Exercise 5: Using tasks to calculate Pi

Exercises 1 : A recurring example

Numerical integration

$$Pi = \int_0^1 \frac{4}{1+x^2}dx \approx \frac{1}{N}\sum_{i=1}^{N} f\left(\frac{i}{N} - \frac{1}{2N}\right) = \frac{1}{N}\sum_{i=1}^{N} f\left(\frac{i-0.5}{N}\right)$$



Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i;
   double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Hot spot

Exercise 2

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines

✓ int omp_get_num_threads(); → **Number of threads in the team**

✓ int omp_get_thread_num(); → **Thread ID or rank**

✓ double omp_get_wtime(); → **Time in Seconds since a fixed point in the past**

Algorithm strategy:The SPMD Pattern

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
   int i, nthreads; double pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)  nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
          }
   }
  for(i=0, pi=0.0;i<nthreads;i++)
}     pi += sum[i] * step;
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

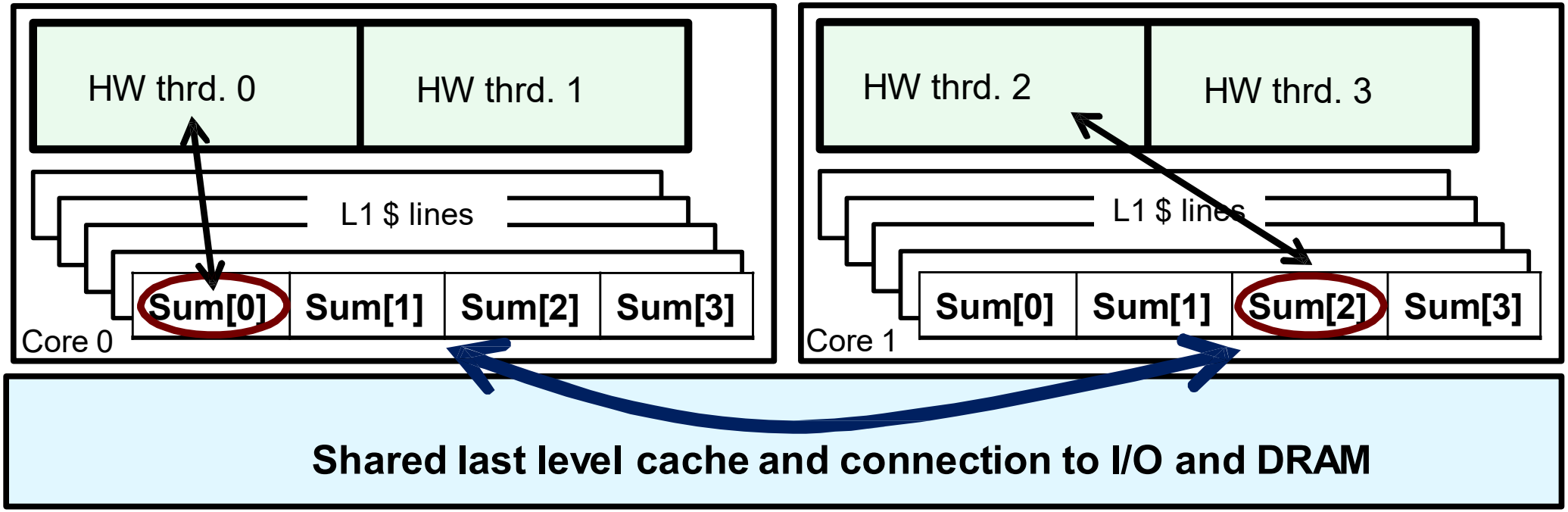
Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)  nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id] += 4.0/(1.0+x*x);
            }
   }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

threads	1st SPMD
1	1.86
2	2.53
3	2.57
4	2.38

Why such poor scaling? False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "flash back and forth" between threads … This is called **"false sharing"**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD      8            // assume 64 byte L1 cache line
#define NUM_THREADS 2 size
void main ()
{     int i, nthreads;
      double pi, sum[NUM_THREADS][PAD];
      step = 1.0/(double) num_steps; omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)     nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id][0] += 4.0/(1.0+x*x);
           }
      }
      for(i=0, pi=0.0;i<nthreads;i++)
                        pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

Results*: pi program padded accumulator

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD    8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id][0] += 4.0/(1.0+x*x);
            }
   }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	2.53	1.01
3	2.57	0.69
4	2.38	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Do we really need to pad our arrays?

- Padding arrays requires deep knowledge of the cache architecture.
- Move to a machine with different sized cache lines and your software performance falls apart.
- There has got to be a better way to deal with false sharing.

Exercise 3

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
  {    int i;
       double x, pi, sum = 0.0;
       step = 1.0/(double) num_steps;
        #pragma omp parallel
        { double x;
          #pragma omp for reduction(+:sum)
               for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
               }
        }
          pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum.
Note … the loop indix is local to a thread by default.

Loops (cont.)

- Made `schedule(runtime)` more useful
  - can get/set it with library routines

```
omp_set_schedule()
omp_get_schedule()
```

  - allow implementations to implement their own schedule kinds
- Added a new schedule kind **AUTO** which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed C++ Random access iterators as loop control variables in parallel loops

Exercise 4

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
  double   pi;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
      int i, id,nthrds;
      double x, sum;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      if (id == 0)   nthreads = nthrds;
      for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
      }
      #pragma omp critical
            pi += sum * step;
  }
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{           double  pi;       step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
            int i, id,nthrds;    double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)  nthreads = nthrds;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        #pragma omp critical
                         pi += 4.0/(1.0+x*x);
             }
    }
              pi *= step;
    }
```

Be careful where you put a critical section

What would happen if you put the critical section inside the loop?

Example: Using an atomic to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 2
 void main ()
{           double  pi;    double step;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
           #pragma omp parallel
          {
            int i, id,nthrds;  double x, sum;
            id =  omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)        nthreads = nthrds;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps;
                       i=i+nthreads){ x = (i+0.5)*step;
                       sum += 4.0/(1.0+x*x);
           }
            sum = sum*step;
           #pragma atomic
                 pi += sum ;
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi so updates don’t conflict

Exercise 5

Exercise: Pi with tasks

- Go back to the original pi.c program
  - Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

Program: OpenMP tasks

```
include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
    double x, sum = 0.0,sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
      }
    }
    else{
      iblk = Nfinish-Nstart;
      #pragma omp task shared(sum1)
          sum1 = pi_comp(Nstart,        Nfinish-iblk/2,step);
      #pragma omp task shared(sum2)
           sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
      #pragma omp taskwait
        sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
          sum =
                  pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```