Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- OpenMP parallel for
- Nested loops
- Loop dependencies

Questions?

Scheduling Loops

Example

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

In practice, how are iterations assigned to threads?

Thread	Iterations
0	0, 1, 2, …, n/t – 1
1	n/t, n/t + 1, …, 2n/t
…	…
t-1	n(t-1)/t, …, n-1

Default partitioning.

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t+1, 2n/t+1, \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \ldots$

Cyclic partitioning.

Example

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

- i.e., f(i) calls the sin function *i* times.
- assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).
- How are iterations assigned to threads? What's the best way of doing it?

Results

#Threads	1	2 (default scheduling)	2 (cyclic scheduling)
Runtime	3.67	2.76	1.84
Speedup	1	1.33	1.99

The Schedule Clause

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

Questions?

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

The Static Schedule Type

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

The Static Schedule Type

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

Thread 0 : 0, 1, 6, 7

Thread 1 : 2, 3, 8, 9

Thread 2 : 4, 5, 10, 11

The Static Schedule Type

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

Thread 0 : 0, 1, 2, 3

Thread 1 : 4, 5, 6, 7

Thread 2 : 8, 9, 10, 11

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.
- Better load balancing, but higher overhead to schedule the chunks (can be tuned through the chunksize)

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- The chunks have size *num_iterations/num_threads,* where *num_iterations* is the number of unassigned iterations
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.
- Smaller chunks towards the end to avoid stragglers

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
- E.g.: `$ export OMP_SCHEDULE="static,1"`

- Another way to specify the schedule type is to set it with the function omp_set_schedule(omp_sched_t kind, int chunk_size);

How to select a schedule option

- **static**: If iterations are homogeneous
- **dynamic/guided**: If execution cost varies
- If in doubt, set:

```
#pragma omp parallel for schedule( runtime )
for(. . .
```

- There is no guarantee it will select the most performing schedule option.
- Measure/Try different options (the best choice might be different depending on the input to the program)

Exercise (@home)

4. Create a C program for visualizing the thread iteration assignment performed by a `parallel for` directive, for different `schedule` schemes. Your program should have a per-thread vector that accumulates the loop control variable values assigned to each thread. Use this program, and appropriate `schedule` settings, to experiment with multiple schemes, without recompiling your program. This is a sample of the output you should be able to get for a loop of 100 iterations:

```
$ export OMP_SCHEDULE="static,5"
$ ./solution
Thread 0 : 0 1 2 3 4 40 41 42 43 44 80 81 82 83 84
Thread 1 : 5 6 7 8 9 45 46 47 48 49 85 86 87 88 89
Thread 2 : 10 11 12 13 14 50 51 52 53 54 90 91 92 93 94
Thread 3 : 15 16 17 18 19 55 56 57 58 59 95 96 97 98 99
Thread 4 : 20 21 22 23 24 60 61 62 63 64
Thread 5 : 25 26 27 28 29 65 66 67 68 69
Thread 6 : 30 31 32 33 34 70 71 72 73 74
Thread 7 : 35 36 37 38 39 75 76 77 78 79
```

Questions?

OPIS

OPIS

OPIS Code: B76Z7JSO

Instructions:

More on Mutual Exclusion

Named Critical Sections

- OpenMP provides the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously (i.e., it is like acting on two different locks)
- However, these must be set at compile time
- What if we want to have multiple locks/critical sections but we do not know how many at compile time? (e.g., a linked list with a lock for each node)

Locks in OpenMP

```
omp_lock_t writelock;
omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
    omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

critical, atomic, or locks?

1. In general, the atomic directive has the potential to be the fastest method of obtaining mutual exclusion.
2. However, the OpenMP specification allows the atomic directive to enforce mutual exclusion across all atomic directives in the program. i.e., the following might be executed in a mutually exclusive way (depends on the implementation)

```
# pragma omp atomic
x++;
```

```
# pragma omp atomic
y++;
```

3. The use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

Some Caveats

1. You shouldn't mix the different types of mutual exclusion for a single critical section.

```
#  pragma omp atomic
   x += f(y);
```

```
#  pragma omp critical
   x = g(x);
```

2. There is no guarantee of fairness in mutual exclusion constructs.
3. It can be dangerous to "nest" mutual exclusion constructs.

Nested critical sections

```
int main() {

#  pragma omp critical
y = f(x);
…
}

double f(double x) {

#  pragma omp critical
z = g(x); /* z is shared */
…
return z;
}
```

Solution

1. The nested critical sections will go into deadlock.
2. In this example, we can solve the problem by using named critical sections.

```
int main() {

#  pragma omp critical(one)
y = f(x);
…
}

double f(double x) {

#  pragma omp critical (two)
z = g(x); /* z is shared */
…
return z;
}
```

Exercises

How to take timings

- Use the GET_TIME macro we have seen in the last exercise session (you can find the code on Github)
- Use omp_get_wtime

Exercise

lec15 - Matrix Multiplication

Synchronization Constructs

Master/Single Directives

- **master**, **single**: both force the execution of the following structured block by a single thread. There is a significant difference: single implies a *barrier* on exit from the block. There are other differences (e.g., with master, the block is guaranteed to be executed by the master thread).

```
int examined = 0;
int prevReported = 0;
#pragma omp for shared( examined, prevReported )
  for( int i = 0 ; i < N ; i++ )
   {
      // some processing

      // update the counter
#pragma omp atomic
      examined++;

      // use the master to output an update every 1000 newly ↩
          finished iterations
#pragma omp master
      {
         int temp = examined;
         if( temp - prevReported >= 1000)
           {
             prevReported = temp;
             printf("Examined %.2lf%%\n", temp * 1.0 / N );
           }
      }
   }
```

Barrier Directive

- **barrier**: blocks until all team threads reach that point.

```
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i*i;

        // Print intermediate results.
        #pragma omp master
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);

        // Wait.
        #pragma omp barrier

        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

The section/sections directives

- How can we send different task in parallel?

```
#pragma omp parallel
switch (omp_get_thread_num())
    {
    case 0: {
          //concurrent block 0
          }
    break;
    case 1: {
          //concurrent block 1
          }
    break;
    }
}
```

The section/sections directives

- The individual work items are contained in blocks decorated by section directives:

```
#pragma omp parallel sections
{
#pragma omp section
  {
    // concurrent block 0
  }
 ...
#pragma omp section
  {
    // concurrent block M-1
  }
}
```

- omp parallel sections directive combines the omp parallel and omp sections directives.
- There is an implicit barrier at the end of a sections construct unless a nowait clause is specified.

Synchronization Constructs

- **ordered**: used inside a parallel for, to ensure that a block will be executed as if in sequential order.

```
double data[ N ];
#pragma omp parallel shared( data, N )
{

#pragma omp for ordered schedule( static, 1 )
   for(int i = 0; i < N; i++)
      {
         // process the data

         // print the results in order
#pragma omp ordered
         cout << data[i];
      }
}
```

ordered clause is required

Questions?

Exercises

Exercise

lec15 - Pi Calculation

False sharing

False sharing

- **False sharing** : sharing cache lines without actually sharing data.
- How to fix it:
  - Pad the data
  - Change the mapping of data to threads/cores
  - Use private/local variables

Padding the data

- Original

```
double x[N];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];
#pragma omp parallel for schedule(static, 1)
  for( int i = 0; i < N; i++ )
     x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

- Can kill cache effectiveness.
- Wastes memory.

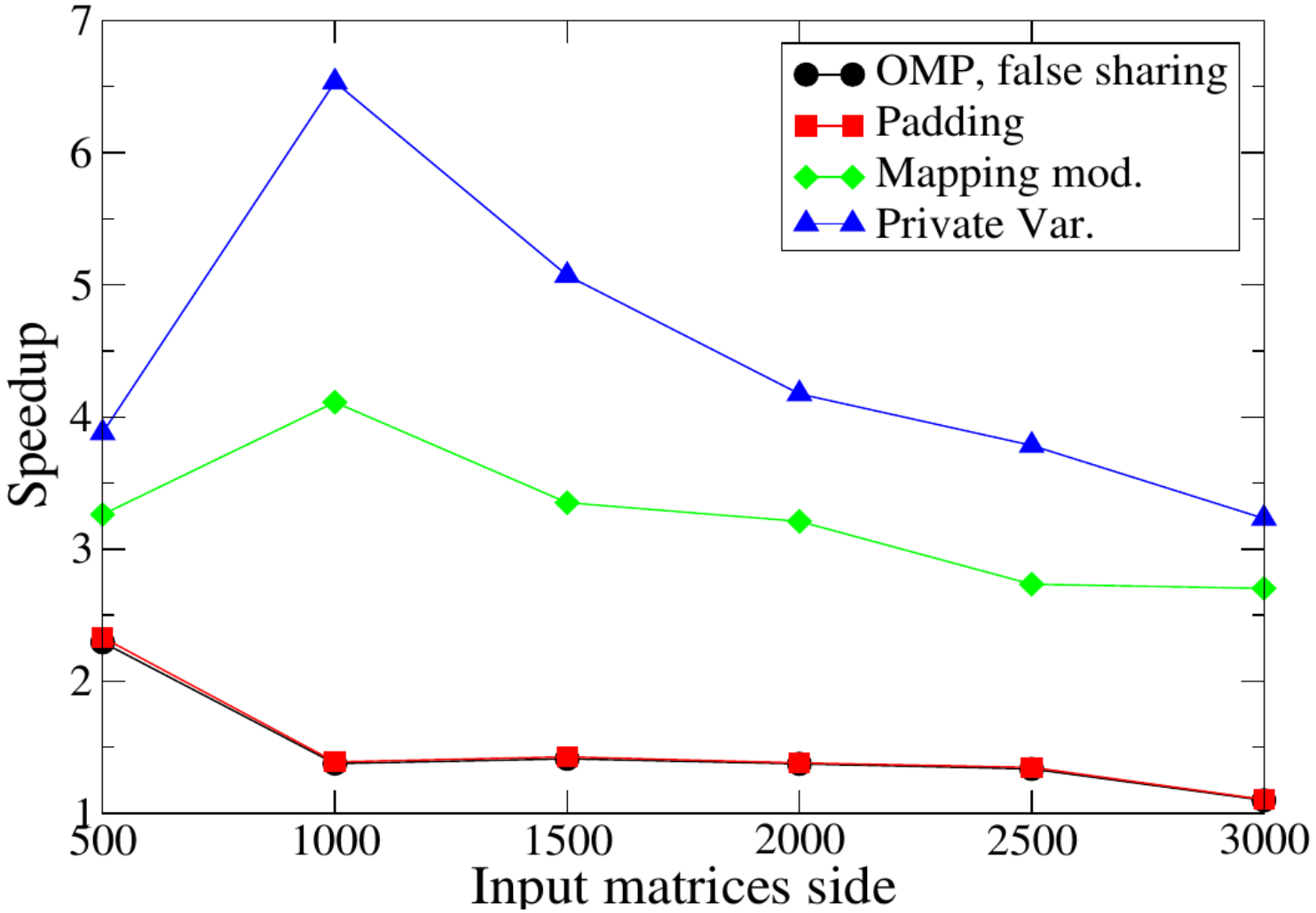
Data mapping change

```
double x[N];
#pragma omp parallel for schedule(static, 8)
  for( int i = 0; i < N; i++ )
      x[ i ] = someFunc( x [ i ] );
```

Using private variables

```
// assuming that N is a multiple of 8
double x[N];
#pragma omp parallel for schedule(static, 1)
   for( int i = 0; i < N; i += 8 )
   {
      double temp[ 8 ];
      for(int j = 0; j < 8; j++)
         temp[ j ] = someFunc( x [ i + j] );
      memcpy( x + i, temp, 8 * sizeof( double ));
   }
```

Impact of false sharing on matrix multiplication



Questions?

OpenMP + MPI

- MPI defines 4 levels of thread safety:
  - MPI_THREAD_SINGLE: One thread exists in program
  - MPI_THREAD_FUNNELED: only the master thread can make MPI calls. Master is one that calls MPI_Init_thread()
  - MPI_THREAD_SERIALIZED: Multithreaded, but only one thread can make MPI calls at a time
  - MPI_THREAD_MULTIPLE: Multithreaded and any thread can make MPI calls at any time

- Safest (easiest) to use **MPI_THREAD_FUNNELED**
  - Fits nicely with most OpenMP models
  - Expensive loops parallelized with OpenMP
  - Communication and MPI calls between loops

Exercises

Exercise

lec15 - Histogram Calculation