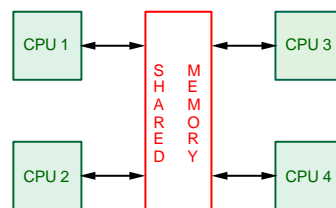


Parallelization on shared memory machines

RGP I - week 4 part II
OpenMP

Shared memory architectures

- ◆ All CPUs share the same memory space
- ◆ Semi-automatic parallelization is possible



- ◆ Example

- ◆

```
for (int n=0;n<100;++n)
  work(n);
```

- ◆ Have a thread (lightweight process) on each CPU do 1/4 of the iterations
- ◆ OpenMP is the standard for such semi-automatic parallelization

OpenMP standard

- ◆ Home page: <http://www.openmp.org>
 - ◆ Contains the specification of the standard including many examples
- ◆ We will look at the C/C++ standard
- ◆ Semi-automatic parallelization using directives
 - ◆ A directive is written as a line before the statement or block of statements:

```
#pragma omp directive
```
- ◆ Some auxiliary function calls

A first parallel example

- ◆ A simple loop is parallelized
 - ◆ Possible only if there are no dependencies
- ◆

```
#pragma omp parallel  
{ // each parallel region starts with this directive  
#pragma omp for  
  for (i=1; i<n; i++)  
    b[i] = (a[i] + a[i-1]) / 2.0;  
}
```
- ◆ Or the shortcut version for a single loop

```
#pragma omp parallel for  
for (i=1; i<n; i++)  
  b[i] = (a[i] + a[i-1]) / 2.0;
```

Two loops

- ◆ Two loops are parallelized

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
  for (i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

- ◆ The `nowait` directive avoids the implied barrier at the end of the first loop. A thread may start on the second loop before the first is finished.

Loop parallelization schedules

- ◆ Several scheduling strategies can be specified

```
#pragma omp parallel for schedule(schedule)
for (i=1; i<n; i++)
  b[i] = (a[i] + a[i-1]) / 2.0;
```

- ◆ Static scheduling: `schedule(static, chunksize)`
 - ◆ Assigns each thread blocks of size `chunksize` at compile time
 - ◆ Useful if all iterations take the same amount of work and all threads are equally fast
- ◆ Static scheduling: `schedule(dynamic, chunksize)`
 - ◆ Assigns each thread blocks of size `chunksize`
 - ◆ Once a thread finishes a block it gets the next block to be done
 - ◆ Useful if all iterations take the same amount of work but not all threads are equally fast
- ◆ Static scheduling: `schedule(guided, chunksize)`
 - ◆ Assigns blocks of size at least `chunksize`
 - ◆ At first bigger blocks are used, later smaller ones to give optimal performance
 - ◆ Useful if neither thread speed nor work load are equal

Simple parallel regions

- ◆ Split the work

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    ipoints = npoints / np;
    subdomain(x, iam, ipoints);
}
```

- ◆ `shared` specifies which variables are shared between the threads
- ◆ `private` specifies variables of which each thread has its own
- ◆ By default all variables except for loop counters in for loop are shared
- ◆ For more information, and `firstprivate`, `lastprivate`, `copyin` see the OpenMP specification

Auxilliary functions

- ◆ `omp_get_thread_num()` ... returns the number of the current thread
- ◆ `omp_set_num_threads(int)` ... sets the number of threads
- ◆ `omp_get_num_threads()` ... returns the number of threads
- ◆ `omp_get_max_threads()` ... returns the maximum number of threads
- ◆ `omp_get_num_procs()` ... returns the number of processors used
- ◆ `omp_set_dynamic(bool)` ... enables/disables automatic adjustment of the number of threads
- ◆ `omp_get_dynamic()` ... returns if automatic adjustment is allowed
- ◆ All these functions work only with OpenMP. To make the code portable use the following trick to e.g. enforce four threads if OpenMP is used:

```
#ifdef _OPENMP
omp_set_dynamic(false);
omp_set_num_threads(4);
#endif
```

Critical sections

- ◆ Some parts of code may be critical
 - ◆ Only one thread may enter it at any time
 - ◆ Example: assigning a new task
- ◆

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
  #pragma omp critical ( xaxis )
  x_next = dequeue(x);
  work(x_next);
  #pragma omp critical ( yaxis )
  y_next = dequeue(y);
  work(y_next);
}
```
- ◆ Different critical sections may be distinguished by names
 - ◆ Each section with a certain name may be entered only by one thread at a time.
 - ◆ More than one section may have the same name: only one thread at a time may be in any section with the given name

Performing atomic updates

- ◆ We need to store results of calculations
 - ◆ No two threads should try to update the same location simultaneously
- ◆ Solution 1: make the writing **critical**: only one thread will ever write terribly slow, no speedup!


```
#pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++) {
  #pragma omp critical
  x[index[i]] += work1(i);
  y[i] += work2(i);
}
```
- ◆ Solution 2: make the writing **atomic**: no two threads will ever have the same value of `index[i]` simultaneously, much faster


```
#pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++) {
  #pragma omp atomic
  x[index[i]] += work1(i);
  y[i] += work2(i);
}
```

Reductions

- ◆ Loops like the following may appear in an integration code

```
for (i=0; i<n; i++)
    sum += f(a+i*delta);
```

- ◆ This is one way to parallelize, using features we know

```
#pragma omp parallel shared (sum) private(partial)
{
    partial=0.;
    #pragma omp for
    for (i=0; i<n; i++)
        partial += f(a+i*delta);

    #pragma omp atomic
    sum += partial;
}
```

- ◆ Or better, automatically using the reduction clause

```
#pragma omp parallel for reduction(+: sum)
for (i=0; i<n; i++)
    sum += f(a+i*delta);
```

Parallelizing macro tasks

- ◆ Consider three functions that can be executed simultaneously:

```
#pragma omp parallel sections
{
    #pragma omp section
    xaxis();
    #pragma omp section
    yaxis();
    #pragma omp section
    zaxis();
}
```

Statements executed only by a single thread

- ◆ Only a single thread should ever print

```
#pragma omp parallel
{
    #pragma omp single
    std::cout << "Beginning work1.\n";

    work1();
    #pragma omp single
    std::cout << "Finishing work1.\n";
    #pragma omp single nowait
    std::cout << "Finished work1 and beginning work2.\n";
    work2();
}
```

Keeping the same order

- ◆ Consider a loop

```
for (i=lb; i<ub; i+=st)
    work(i);
```

- ◆ This function works but prints the number in arbitrary order

```
void work(int k)
{
    #pragma omp critical
    std::cout << k;
}
```

- ◆ The ordered pragma ensures to get the same order as in sequential execution

```
void work(int k)
{
    #pragma omp ordered
    std::cout << k;
}
```

OpenMP capable compilers

- ◆ The compiler of choice is “guidec / guidec++” from Kuck and Associates (KAI)
It is installed on “Stardust”
- ◆ The recently released Intel C++ Compiler for Linux has OpenMP support built in. It’s free for non commercial use.
- ◆ Most Fortran compilers on supercomputing platforms support OpenMP