Work Sharing Constructs

• Are contained in some parallel context.
  - Parallel regions and scope
    - parallel
    - parallel do (parallel for)
    - parallel sections.

• Work sharing constructs within a parallel region
  - do (for)
  - sections
  - single
  - master
What’s the difference.

- `#pragma omp parallel [clause ()] [clause( )]
- `#pragma omp for
- for(j=0; j<N ; j++) {
- do_some_work(j);
- }
- is the same as !!!!
- `#pragma omp parallel for [clause ()] [clause( )]
- for(j=0; j<N ; j++) {
- do_some_work(j);
- }

What’s the difference [2].

- `#pragma omp parallel [clause()] [clause()]`
- {
  - `num_threads = omp_get_num_threads();`
  - `do_some_work_in_parallel();`
- `#pragma omp for`
- `for(j=0; j<N ; j++) {
  - `do_some_work(j);`
- `}`
- `}`
Example

- `#pragma omp parallel private(j)`
- `{  
  for( j = 0; j<10 ; j++)  
  printf("hello world %d\n",j); 
}`

If there is a team of 4 threads how many print statements would you see? 40
Restrictions on regions sections etc.

- Block nature of code must be maintained.
- Cannot branch in or out of a region, section etc.
  - No pasta based code 😊
- You can have nested parallel regions:
  - Something to be avoided if possible.
  - ALL OpenMP implementations serialize the inner regions
    ▪ The inner parallel region is a team of 1 thread!
  - This is something the standard is addressing now.
- Can have multiple work-sharing constructs in a region.
  - `do`, `section`, `single`, `master`
- All threads must encounter the SAME work-sharing constructs.
Threadprivate

- Makes a copy of a global data structure for each thread, copy comes from master thread.
  - Fortran common blocks, modules
  - C/C++ global and static data structures
  - Copy only done at the beginning of the FIRST parallel region.

- Threadprivate data hangs around across the scope of a parallel region.
  - Different from other entities.
  - Threads really don’t go away they are just inactive in the static extent.
  - New dynamic threads get a copy of the data of the master thread.
Threadprivate [2]

- The OpenMP directive must be applied AFTER the data is declared
  
  ➤ And everywhere the data is declared!!

- common /cm/ data, data2
- !$omp threadprivate(/cm/)

- !$omp parallel
- code
- !$omp end parallel
copyin(list)

• The copyin clause is used at the beginning of a region to put data from the master thread into the threadprivate data structures.
  ➢ This provides a mechanism to re-synchronize data between parallel regions.

• Basically only use threadprivate and copyin functionality when it is absolutely necessary.
The parallel sections

- A lot like parallel for
  - A combined parallel region and work sharing construct.

- `#pragma omp parallel sections`
  - `{`
  - `#pragma omp section`
  - `{ block of code 1}`
  - `#pragma omp section`
  - `{block of code 2}`
  - `#pragma omp section`
  - `{block of code 3}`
  - `}`
The parallel sections

• #pragma omp parallel sections
• {
  • #pragma omp section
  • { malloc A; generate_A}
  • #pragma omp section
  • { malloc B; generate_B}
  • #pragma omp section
  • { malloc C; generate_C}
• }
Single and Master directives

• The single directive allows a block of code to be executed by only one thread!!
  ➢ Implied barrier at the end of
  ➢ Any thread can do the job
    • Implementation dependent

• The Master directive
  ➢ Very similar to the single directive
  ➢ ONLY the master thread can execute the code block

• A section is executed by a single thread
  ➢ Any thread
  ➢ Multiple concurrent sections can execute.
  ➢ A parallel sections with only one section is the same as single.
Example of single

- `#pragma omp parallel shared(input_data)`
- `{`
- `  code … code … code`
- `#pragma omp single`
- `{`
- `  scanf("%d",&input_data);`
- `}` /* implied barrier here */
- `code`
- `}`
The Runtime Library Calls

• Get the number of threads
  ➢ int omp_get_num_threads(void);
    - integer function omp_get_num_threads()

• Set the number of threads
  ➢ void omp_set_num_threads(int);
    - subroutine omp_set_num_threads(integer)

• Get the maximum number of threads
  ➢ int omp_get_max_threads(void);
    - integer function omp_get_max_threads()

• Get the number of processors available
  ➢ int omp_get_num_procs(void);
    - integer function omp_get_num_procs()
Other Runtime Calls

• Determine if in a parallel region
• Determine and set the dynamic thread extent
  ➢ Default is number of threads is equal to number of processors and static extent.
• Determine and set whether nested parallelism is enabled or disabled
  ➢ Remember all implementations are disabled at this point.
• Determine and set nested lock mechanisms
PI Program: an example

```c
/*--- Plain serial code ------------*/
#include <stdio.h>
#include <math.h>
#include "CWtime.h"
int main (int argc, char *argv[])
{
    int num_steps = 100000000;
    int i;
    double PI = (double)0.0;
    double step, x, pi, sum = (double)0.0;
    double t0, t1, t, w0, w1, w;

    PI = ((double)2.0)*acos((double)0.0));
    step = (double)1.0/(double) num_steps;
```
PI Program: an example

```c
w0 = Wall_Time();
t0 = CPU_Time();

for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x);
}

pi = step * sum;
```
PI Program: an example

```c
int main()
{
    double PI, pi, x;
    double t1, w1, t0, w0;
    int num_steps;

    t0 = CPU_Time();
    w0 = Wall_Time();
    t1 = CPU_Time();
    w1 = Wall_Time();
    t = t1 - t0; w = w1 - w0;
    x = fabs(PI - pi);
    printf("PI = %.15f\n", PI);
    printf("pi = %.15f, diff = %.6e\n", pi, x);
    printf("Number of steps: %d\n", num_steps);
    printf("Number of threads: %d\n", num_threads);
    printf("CPU Time : %.2f \n", t);
    printf("Wall Time : %.2f \n", w);
}
```
OpenMP PI Program: Parallel for with a reduction

/*------- parallel for with a reduction -----------------*/
#include <stdio.h>
#include <math.h>
#include "CWtime.h"
#include <omp.h>
int main (int argc, char *argv[])
{
    int num_steps = 100000000;
    int i, num_threads;
    double PI = (double)0.0;
    double step, x, pi, sum = (double)0.0;
    double t0, t1, t, w0, w1, w;

OpenMP adds 2 to 4 lines of code
OpenMP PI Program: Parallel for with a reduction

\[ PI = ((\text{double}2.0) \times (\text{acos}((\text{double}0.0)))); \]
\[ \text{step} = (\text{double}1.0)/(\text{double} \text{ num\_steps}); \]

```c
#pragma omp parallel shared(num_threads)
{
    #pragma omp single
    {
        num_threads = omp_get_num_threads();
    }
}
```

OpenMP adds 2 to 4 lines of code
OpenMP PI Program: Parallel for with a reduction

w0 = Wall_Time();
t0 = CPU_Time();

#pragma omp parallel for reduction(+:sum) private(x)
for (i=1; i<= num_steps; i++){
    x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x);
}
pi = step * sum;

OpenMP adds 2 to 4 lines of code
OpenMP PI Program: Parallel for with a reduction

```c
    t = t1-t0; w = w1-w0;
    x = fabs(PI - pi);
    printf("PI = %.15f\n",PI);
    printf("pi = %.15f, diff = %.6e\n",pi,x);
    printf("Number of steps: %d   Number of threads: %d\n",num_steps,num_threads);
    printf("CPU Time : %.2f \n",t);
    printf("Wall Time : %.2f \n",w);
}
```

OpenMP adds 2 to 4 lines of code
OpenMP PI Program: Parallel Region example (SPMD Program)

/*- SPMD parallel region --------------*/
sums = (double *)malloc
    (num_threads*sizeof(double));
#pragma omp parallel
    { double xp;  int id;
        id = omp_get_thread_num();
        sums[id] = (double)0.0;
        for (i=(id+1);i<= num_steps; i+=num_threads){
            xp = (i-0.5)*step;
            sums[id] += 4.0/(1.0+xp*xp);
        } } pi = (double)0.0;
for(i=0;i<num_threads;i++)
    pi += step * sums[i];
OpenMP PI Program: Work sharing construct

#pragma omp parallel
{
    double xp;  int id;  id = omp_get_thread_num();
    sums[(id*(1+padsize))] = (double)0.0;
#pragma omp for
    for (i=(id+1);i<= num_steps; i++){
        xp = (i-0.5)*step;
        sums[(id*(1+padsize))] += 4.0/(1.0+xp*xp);
    }
}
pi = (double)0.0;
for(i=0;i<num_threads;i++)
    pi += step * sums[i*(1+padsize)];
OpenMP PI Program: A private clause and a critical section

pi = (double)0.0;
#pragma omp parallel private(i,x,sum)
{
    int id;
    id = omp_get_thread_num();
    sum=(double)0.0;
    for (i=(id+1);i<= num_steps; i+=num_threads){
        x = (i-0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
#pragma omp critical
    pi += sum*step;
}

Note: We didn’t need to create an array to hold local sums or clutter the code with explicit declarations of “x” and “sum”.
## Timings

<table>
<thead>
<tr>
<th>Piparfor</th>
<th>Pispmd</th>
<th>Pispmdpad</th>
<th>Piwork</th>
<th>picrit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.08</td>
<td>3.08</td>
<td>3.08</td>
<td>3.12</td>
</tr>
<tr>
<td>2</td>
<td>1.54</td>
<td>10.24</td>
<td>1.54</td>
<td>1.56</td>
</tr>
<tr>
<td>3</td>
<td>1.03</td>
<td>5.91</td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td>4</td>
<td>0.77</td>
<td>8.44</td>
<td>0.77</td>
<td>0.78</td>
</tr>
</tbody>
</table>
Synchronization

• Many forms

  ➢ Implied at the end of work sharing constructs
    ▪ single
    ▪ master
    ▪ section
    ▪ do/for
      ♦ Removed with a nowait clause

  ➢ Critical sections
    ▪ Can be named!!
      ♦ Must not have the name of a routine or global variable
    ▪ Why would you name them?
      ♦ Different critical sections are orthogonal!
        ♣ This means that a critical section “hawkeye” can execute
          while critical section “cyclone” is executing!!!
Synchronization [2]

- Critical sections must be in the same routine.
  - May encompass multiple statements and function calls within the critical section.
- The “atomic” directive is the single statement version of the critical section.
- Mutex locks are available as well.
  - These are more flexible than critical sections
  - Can be placed anywhere in the code.
  - The execution order determines what happens.
    - First obtain the lock
    - Do section critical work
    - Release the lock
  - We will learn more about these in the Pthreads model.
Synchronization [3]

• Explicit barriers
  ➢ `#pragma omp barrier`
  ➢ `!$omp barrier`
    ▪ Used within a parallel region to force a barrier among all threads.

• The last sync operation is “ordered”

• This forces a section of code in a parallel loop to be executed in the sequential extent of the loop.
  ➢ Computations in the parallel loop are parallelizable.
  ➢ Each iteration waits for the completion of the previous iteration’s ORDERED instructions.
  ➢ Often used to force sequential looking output!
OpenMP Summary Pages

- `#pragma omp directive-name [clause]`
- **Parallel region** “omp parallel [clause]”
  - Work-sharing clauses (inside a parallel region)
    - `#pragma omp for [clause]`
      - `for(...)`
    - `#pragma omp sections [clause]`
      - `{`
        - `[#pragma omp section block of code ]`
      - `}`
    - `#pragma omp single block of code`
OpenMP Summary Pages (2)

- Combined parallel and work-sharing directives
  - `#pragma omp parallel for [clause]`
  - `#pragma omp parallel sections [clause]
    {
      [#pragma omp section
       block of code ]
    }
  `
OpenMP Summary Pages (3)

- **Synchronization directives**
  - `#pragma omp master`
  - Block of code
  - `#pragma omp critical [(name)]`
  - `#pragma omp barrier`
  - `#pragma omp atomic`
  - Line of code
  - `#pragma omp flush[(list)]`
  - `#pragma omp ordered`
  - Block of code

- **Data specific**
  - `#pragma omp threadprivate (list)`
OpenMP Summary Pages (4)

- shared (list)
- private (list)
- firstprivate (list)
- lastprivate (list)
- reduction (operator|intrinsic : list)
- copyin (list)
- if (expression)
- schedule(type[,chunk])
- nowait
OpenMP Summary Pages (5)

- **reduction** (operator|intrinsic : list)

- Operators in C / C++
  - +
  - *
  - -
  - &
  - |
  - ^
  - &&
  - ||
OpenMP Summary Pages (6)

• `schedule(type[,chunk])`

  - **type**
    - `static`
    - `dynamic`
    - `guided`
    - `runtime`
      - `OMP_SCHEDULE`

  - **chunk**
    - Either a static chunk size or a minimum chunk size
    - A chunk of iterations in the loop!
OpenMP Summary Pages (7)

• **Library routines**
  - `void omp_set_num_threads(int)`
  - `int omp_get_num_threads(void)`
  - `int omp_get_max_threads(void)`
  - `int omp_get_thread_num(void)`
  - `int omp_get_num_procs(void)`
  - `int omp_in_parallel(void)`
  - `void omp_set_nested(int)`
  - `int omp_get_nested(void)`
OpenMP Summary Pages (8)

- **Environment Variables**
  - `OMP_SCHEDULE`
    - “type, chunk”
  - `OMP_NUM_THREADS`
    - An integer value
  - `OMP_NESTED`
    - True or false 😊
    - Nested parallelism turned on.
• **Questions should be asked in class**
  - If you have a question; someone else may also!
  - If you don’t understand something
    - Ask a question
  - If you don’t understand something in the lecture
    - Stop the lecture by asking me to explain it again
  - If I ask ‘‘Does that make sense?’’
    - If so fine.
    - If not ask a question!!!!!!!!!!!!!!!
  - You are paying me to educate you!!!!!
    - The TA is not paid to read you the lecture notes!
    - I am not paid to read you the lecture notes!
    - You have to make an effort.
Are there any Questions??
Homework #3

- Basically using OpenMP you need to parallelize
  - Your ddot matrix multiply routine
  - Your daxpy matrix multiply routine
  - Your blocked matrix multiply routine
  - Generation of your 3 analytical matrices

- Optional
  - Probably a good idea if your gen_C routine is $O(N^3)$!
  - Two ways to think about this especially if all three routines are $O(N^2)$!

- Options
  - Parallelize each generation routine (boring)
    - Not enough work to parallelize!
  - Parallelize the three calls to the generation routines
    - Using sections 😊