Loop Worksharing Constructs

Sequential Code

```
for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

OpenMP Parallel Region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)  { a[i] = a[i] + b[i];}
}
```

OpenMP Parallel Region and
a Worksharing for Construct

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

(OpenMP makes the loop control index on a parallel loop private to a thread)

Software to examine cores

XRG (on Mac)

Menumeters (on top menu)

OpenMP

Internal Control Variables: ICV

omp_set_num_threads: number of threads that are to be used in parallel region

omp_set_dynamic: sets number of threads dynamically: true or false

omp_set_nested: parallel regions can be nested inside of parallel regions. If a parallel thread encounters a parallel region, then it spawns a new team of threads, unless omp_set_nested is false.

omp_set_max_active_levels: number of parallel nesting levels

Environmental Variables:

OMP_NUM_THREADS 8
OMP_DYNAMIC TRUE l FALSE

Synchronization: Barriers

Barrier : all threads have to wait at a fixed location; automatic for end of parallel region

Mutual exclusion (mutex): only one thread or one thread at a time

Constructs: Critical, Atomic, Barrier

```
#pragma omp parallel
{
  int id = omp_get_thread();
  A[id] = big_calculation(id);

#pragma omp barrier     //<—need barrier to ensure all elements of A are calculated
                        //before going on.

B[id]= big_calculation2(id, A);
}
```

NOTE: implied barrier in each for loop

Mutual Exclusion

```
float res;
#pragma omp parallel

{
   float B; int i, id, nthrds;

   id = omp_get_thread_num();
   nthrds = omp_get_num_threads();

   for (i=id; i<niters; i+=nthrds)
    {
      B= big_job(i);
     #pragma omp critical  //only one thread at a time  (serializing the code)
     res + = consume(B);
    }
}
```

Schedule

#pragma omp for schedule(static[,chunk]) Deal out blocks of loop to threads; you know that the load is relatively balanced

#pragma omp for schedule(dynamic[,chunk]) each thread grabs chunk and the grabs more when done; when the load is imbalanced.

#pragma omp for schedule(auto) lets the compiler figure it out

Can save time by putting the parallel pragma together with for

#pragma omp parallel for

instead of
#pragma omp parallel
 {#pragma omp for
…
}

Working with Loops

Want all loop iterations independent

```
int i,j, A[max];

j=5
for (i=0;i <max; I++)
{
    j+=2;                                   loop carried dependency: i=0; j=7
                                                                     i=1; j=9
   A[i]=big(j);
}
```

REWRITE

```
int i,j, A[max];
#pragma omp parallel for
for (i=0; i<max; i++)
 {
   int j=5+2*(i+1);                  no loop carried dependency; run in any order
   A[i]= big(j);
}
```

An Example: Average

```
double avg = 0.0;
double A[max];
int i;
for (i=0; i<max; i++)
 { avg += A[i];
}
avg= avg/max
```

Known as a **reduction**. OpenMP reduction (op:list) makes a local copy of avg; then combines with global value

```
double avg = 0.0;
double A[max];
int i;
#pragma omp parallel for reduction(+:avg)    <— —
for (i=0; i<max; i++)
 { avg += A[i];
}
avg= avg/max
```

Reduction operators: + * / — min max

Convert The Pi Calculation to OpenMP: myPinC.cc

Review MyPinC.cc (in TextEditor)

Simple Pi

```
#include <omp.h>
static long num+staps = 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;
}
```

Pi Calculation

Examine myPiOpenMP.cc

#pragma omp for: nested loops

```
for (int i=0; i< N; i++)              //loop 1
    {
      for (int j = 0; j< M; j++)      //loop 2
           {  … do stuff with i,j
           }
     }
```

Choices:

If you use **#pragma omp for** in front of first loop; only parallelizes first loop; also if you put this command in front of both loops, OMP does not do 2nd loop.

You could put **#pragma omp for** just in front of second loop, which is pretty good.

Alternatively, you can use **#pragma omp for collapse(2)** in front of first loop, this will make it into one loop that is parallelized and the optimum.

Or do it yourself:

```
#pragma omp parallel for
 for (int count =0; count < N*M; count++)
   {int j = count%M;
    int i = count/M;
    …do stuff with i,j
   }
```

Same task

nowait, master, barrier, single

```
#pragma omp for nowait   //  no implicit barrier }; saves time but risky

#pragma omp master       // block done by master thread only
 {  }                    /// no barrier implied
#pragma omp barrier
                         // all threads finish up here
#pragma omp single       //only one thread executes
{ }                      //barrier at end, could have nowait
```

Sections (not used a lot)

Each section: one thread

```
#pragma omp parallel
{
    #pragma omp sections
    {
    #pragma omp section
        x_calculation();
    #pragma omp section
        y_calculation();
    #pragma omp section
        z_calculation();
    }
}
```

<— header

Using sections

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        printf ("section 1 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("section 2 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("section 3 id = %d, \n", omp_get_thread_num());
    }
}
```

Include this code into a program and run several times.
What happens?
/usr/local/bin/g++ fname.cpp -fopenmp -o fname

Solution

```
#include <omp.h>
#include <istream>
using namespace std;

int main()
{
omp_set_num_threads(8);
#pragma omp parallel
 {
#pragma omp sections
  {
    #pragma omp section
    {
        printf ("section 1 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("section 2 id = %d, \n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf ("section 3 id = %d, \n", omp_get_thread_num());
    }
  }
 }
  return 0;
}
```

Lock: lowest level mutual exclusion

omp_init_lock;
omp_set_lock;
omp_unset_lock;
omp_destroy_lock;

omp_test_lock;

Example: histogram: put data in bins—lots of race conditions

```
#pragma omp parallel for
for(i=0;iNBUCKETS; i++){
   omp_init_lock(&hist_locks[i]);    hist[i] = 0:
}
#pragma omp parallel for
for(i=0;iNVALS; i++){
   ival = (int) sample(arr[i]);
   omp_set_lock(&hist_locks[ival]);
      hist[ival]++;
   omp_unset_lock(&hist_locks[ival]);
 }

for(i=0;i<NBUCKETS; i++)
 omp_destroy_lock(&hist_locks[i]);
```

Variable Scope

Variables defined in serial region before parallel region are by default SHARED.

Variables defined inside the parallel region are not shared

You can change this: SHARED PRIVATE FIRST PRIVATE LASTPRIVATE (comes out of parallel region)

Variables declared private in #pragma omp for are not initialized.
FIRSTPRIVATE are initialized to the global value of variable

LASTPRIVATE retains last value out of the parallel region

Homework

Solve Laplace Equation by iteration using finite differences

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

$$\phi(x,0) = 1.0, \phi(x,1) = 0.0, \phi(0,y) = 0.0, \phi(1,y) = 0$$

Given x= i * dx; y= j*dy; and dx=dy, then

$$\phi_{ij} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right)$$