REVIEW: Worksharing Construct - DO/for Directive Schedule Clause In a Nutshell

- schedule(static) n iterations divided in blocks of n/p.
- schedule(static,m) iterations divided in blocks of m (‘chunk size’), assigned cyclically.
- schedule(dynamic) single iterations, assigned whenever a thread is idle
- schedule(dynamic,m) blocks of m iterations, assigned whenever a thread is idle
- schedule(guided) decreasing size blocks
- schedule(auto) leave it up to compiler/runtime
- schedule(runtime) using environment variable OMP_SCHEDULE

Worksharing Construct - Reduction a brief (very brief) intro

```
int array[8] = { 1, 1, 1, 1, 1, 1, 1, 1};
int sum = 0, i;


#pragma omp parallel for reduction(+:sum)
for (i = 0; i < 8; i++)
{
      sum += array[i];
}

printf("total %d\n", sum);
```

- Reductions are atomic operations
- Can be solved by private variable per thread
- reduction clause is shorthand for all that

An operation acting on shared memory is **atomic** if it completes in a single step relative to other threads. When an atomic store is performed on a shared variable, no other thread can observe the modification half-complete. When an atomic load is performed on a shared variable, it reads the entire value as it appeared at a single moment in time.

Exercise 3 - Code

Take 20 minutes and enhance this code with OpenMP.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main(int argc,char **arg) {

  int nsteps=1000000000; // that's one
                              billion
  double tstart,tend,elapsed,
  pi,quarterpi,h;

  int i;

  tstart = omp_get_wtime(); //gettime();

  quarterpi = 0.; h = 1./nsteps;
```

```
  for (i=0; i<nsteps; i++)
  {
    double x = i*h, y = sqrt(1-x*x);
    quarterpi += h*y;
  }

  pi = 4*quarterpi;
  tend = omp_get_wtime(); //gettime();
  elapsed = tend-tstart;

  printf("Computed pi=%e in %6.3f seconds\n", pi,
elapsed);

  return 0;

}
```

Exercise 3 - Enhancements?

- Use the omp parallel for construct to parallelize the loop.

- Are you seeing any odd behavior? Or weird discrepancles? See if reduction clause can fix this.
  - More on the reduction clause soon!

- Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.

Exercise 3a - Make it parallel, and a schedule

- add ‘adaptive integration’: where needed, the program refines the step size. This means that the iterations no longer take a predictable amount of time.

```
for (i=0; i<nsteps; i++) {
      double x = i*h,x2 = (i+1)*h,

      y = sqrt(1-x*x),y2 = sqrt(1-x2*x2),

      slope = (y-y2)/h;

      if (slope>15) slope = 15;

      int samples = 1+(int)slope,

      is;
```

```
      for (is=0; is<samples; is++)
      {
            double hs = h/samples,

            xs = x+ is*hs,

            ys = sqrt(1-xs*xs);

            quarterpi += hs*ys;

            nsamples++;
      }
}

pi = 4*quarterpi;
```

Exercise 3a - Make it parallel, and add the schedule clause

- Use the omp parallel for construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the reduction clause to fix this.

- Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.

- Start by using schedule(static,n). Experiment with values for n. When can you get a better speedup? Explain this.

- Since this code is somewhat dynamic, try schedule(dynamic). This will actually give a fairly bad result. Why? Use schedule(dynamic,n) instead, and experiment with values for n.

- Finally, use schedule(guided), where OpenMP uses a heuristic. What results does that give?

Ordered Iterations Clause

Let’s recall our OpenMP HelloWorld program.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int nthreads, tid;

/* Fork a team of threads giving them their own copies of
variables */
   #pragma omp parallel private(nthreads, tid)
   {
```

```
/* Obtain thread number */
      tid = omp_get_thread_num();

      printf("Hello World from thread = %d\n",
         tid);
/* Only master thread does this */
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }

   }  /* All threads join master thread and disband */
}
```

Did all the threads execute their code at the same time?

Ordered Iterations Clause

What if we added a loop? Would all the loops execute in their proper order?

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int nthreads, tid;

/* Fork a team of threads giving them their own copies of
variables */
   #pragma omp parallel private(nthreads, tid)
   {
/* Obtain thread number */
      tid = omp_get_thread_num();
```

```
      #pragma omp parallel for
      for (int i=0; i<5)
         printf("Hello World %d from thread = %d\n",
            i, tid);
/* Only master thread does this */
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }

   }  /* All threads join master thread and disband */
}
```

And they do not… Iterations in a parallel loop that are ran in parallel do not execute in lockstep.

Ordered Iterations Clause

The ordered clause coupled with the `ordered` directive can force execution in the right order:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int nthreads, tid;

/* Fork a team of threads giving them their own copies of
variables */
   #pragma omp parallel private(nthreads, tid)
   {
/* Obtain thread number */
      tid = omp_get_thread_num();
```

```
      #pragma omp parallel for ordered
      {
         for (int i=0; i<5)
            printf("Hello World %d from thread = %d\n",
               i, tid);
       }
       #pragma omp ordered
/* Only master thread does this */
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }

   }  /* All threads join master thread and disband */
}
```

NOTE: Each iteration can only encounter *one* `ordered` directive

Nowait Clause

Recall that during a for clause, threads wait until all threads are finished before continuing. The nowait clause allows code to continue with the next line of code in a parallel region.

```
…
   #pragma omp parallel private(nthreads, tid)
   {
/* Obtain thread number */
      tid = omp_get_thread_num();
      #pragma omp parallel for nowait
      {
         for (int i=0; i<1000)
            printf("Hello World %d from thread = %d\n",
               i, tid);
      }
```

```
   #pragma omp parallel for
   {
      for (int i=0; i<1000)
         printf("Goodbye Cruel World %d from thread =
            %d\n", i, tid);
    }

   }
}
```

Any thread that completes its task in the first for loop, may now continue to the second for loop. NOTE: This requires both loops to have the same schedule.

Section Construct

- When doing large blocks of independent computations, split the work into sections.
    - Parallel Loops are an example of independent work units which are numbered
    - If you have a set number of work units, use the section construct
    - Section constructs may contain any number of nested section constructs
    - They need to be coded in such a way that any thread on the current team of threads can execute any section or multitude of sections

```
#pragma omp section
 {
    #pragma omp section
    {
        // one calculation
    }
    #pragma omp section
    {
        // another calculation
    }
}
```

Sections

An example. Suppose we have `y = f(x) + g(x)`

```
…
   double y1,y2;
   #pragma omp sections
   {
      #pragma omp section
         y1 = f(x);
      #pragma omp section
         y2 = g(x);
   }
   y = y1+y2;
```

Given the above code snippet, how can we add a reduction clause for better efficiency?

Single Directive

Specifies that only a single thread should execute this section of the program.

```
#pragma omp parallel num_threads(2)
{
  #pragma omp single
  // Only a single thread can read the input.
  printf_s("read input\n");

  // Multiple threads in the team compute the
  // results.
  printf_s("compute results\n");

  #pragma omp single
  // Only a single thread can write the output.
  printf_s("write output\n");
}
```

- May or may not be the master thread
- Synchronizes through the implicit barrier
  - it’s a **work sharing construct**, there is an implicit barrier after it, which guarantees that all threads have the correct value in their local memory

Master Directive

Specifies that only the master thread should execute a section of the program.

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

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

    // Wait.
    #pragma omp barrier

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

- The master directive supports no OpenMP clauses.
- Does not synchronize through the implicit barrier

Exercise 4 - Homework, Matrix Multiplication Revisited

Write a program that performs matrix multiplication.

- Create two double dimensioned arrays, populate them with random numbers using a single thread
- Create a set of nested loops that multiplies the two arrays (your matrices) together in parallel and using a simple reduction clause (we will be covering reduction in depth)
- Add a worksharing clause, you may need to experiment with this.
- add a time function, and record start time and end time of your loop and how long the loop took to process.
- run this for a 10x10, 100x100, and 1000x1000

How are your running times now compared to when we first did this assignment (Exercise 2)?