OpenMP Things to consider

- OpenMP creates a separate data stack for every worker thread to store copies of private variables (master thread uses regular stack)
- Size of these stacks is not defined by OpenMP standards
    - Intel compiler: default stack is 4MB
    - gcc/gfortran: default stack is 2MB
- Behavior of program is unpredictable when the stack space is exceeded
- most compilers will throw seg fault
- To increase stack size use environment var OMP_STACKSIZE,
    - export OMP_STACKSIZE=512M
    - export OMP_STACKSIZE=1GB

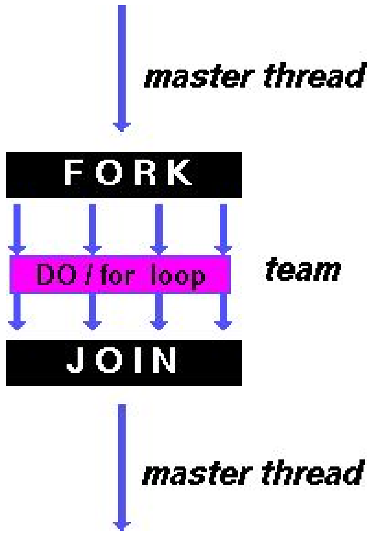
To make sure master thread has large enough stack space use the `ulimit -s` command (unix/linux).

OpenMP: omp_get_wtime

```
double tstart, tend, elapsed

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

      #progma omp parallel
      {...}

tend = omp_get_wtime(); //gettime();

elapsed = tend-tstart;
```

The Worksharing Construct - DO/For

- shares iterations of a loop across the team. Represents a type of **Data parallelism** which focuses on distributing the **data** across different parallel computing nodes (vs. **task parallelism**)



Worksharing Constructs - DO/for Directive

- The DO / for directive specifies that the iterations of the loop **immediately following** it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

Worksharing Construct - DO/for Directive

Fortran

```
!$OMP DO [clause ...]
        SCHEDULE (type [,chunk])
        ORDERED
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        SHARED (list)
        REDUCTION (operator | intrinsic : list)
        COLLAPSE (n)

  do_loop

!$OMP END DO  [ NOWAIT ]
```

Worksharing Construct - DO/for Directive

C/C++

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

  for_loop
```

Worksharing Construct - DO/for Directive

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team.

- **STATIC** Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

- **DYNAMIC** Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

Worksharing Construct - DO/for Directive Schedule Clause (cont)

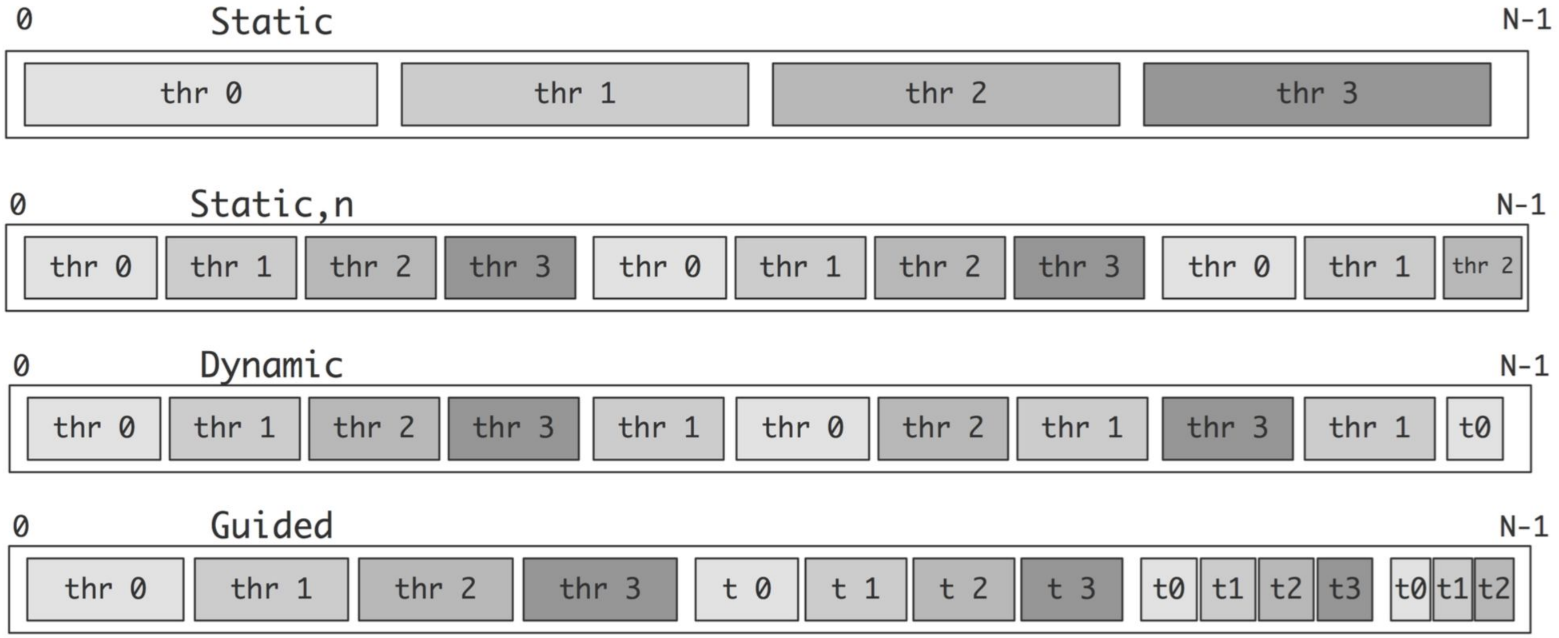
- **GUIDED** Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. (Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.)
  - The size of the initial block is proportional to:
    - number_of_iterations / number_of_threads
  - Subsequent blocks are proportional to
    - number_of_iterations_remaining / number_of_threads
  - The chunk parameter defines the minimum block size.
    - The default chunk size is 1.

Worksharing Construct - DO/for Directive Schedule Clause (cont)

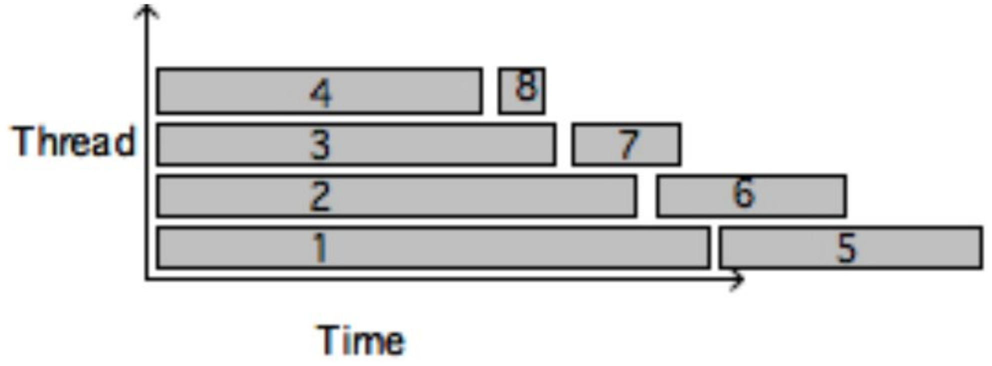
- **RUNTIME** The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

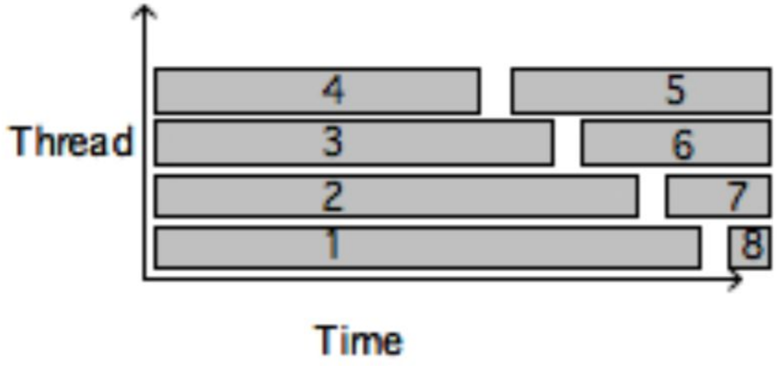
- **AUTO** The scheduling decision is delegated to the compiler and/or runtime system.

Worksharing Construct - DO/for Directive Schedule Clause Visual



Worksharing Construct - DO/for Directive Schedule Clause Visual, Static vs Dynamic





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 - DO/for Directive

NO WAIT/nowait : If set, threads do not sync up at the end of the loop.

ORDERED : Specifies that the iterations of the loop will be executed in serial.

Worksharing Construct - DO/for Directive

COLLAPSE : Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Worksharing Construct - DO/for Directive

Restrictions:

- The DO loop can not be a DO WHILE loop, or a loop without loop control.
  - the loop index must be an integer and the loop control parameters/expressions must be the same for all threads.
- Do not write your code to depend upon which thread executes a particular iteration.
- It is illegal to branch out of a loop associated with a DO/for directive.
- The chunk size must be specified as a loop invariant integer expression.
  - there is no synchronization during its evaluation by different threads.
- ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.

Worksharing Construct - DO/for Directive Example (C)

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{
      int i, chunk;
      float a[N], b[N], c[N];

      /* Some initializations */
      for (i=0; i < N; i++)
      {
            a[i] = b[i] = i * 1.0;
      }
      chunk = CHUNKSIZE;
```

```
#pragma omp parallel shared(a,b,c,chunk)
private(i)
      {

#pragma omp for schedule(dynamic,chunk)
nowait
            for (i=0; i < N; i++)
            {
                  c[i] = a[i] + b[i];
            }


      }  /* end of parallel section */
}
```

Worksharing Construct - DO/for Directive Example(Fortran)

```
      PROGRAM VEC_ADD_DO

       INTEGER N, CHUNKSIZE, CHUNK, I
       PARAMETER (N=1000)
       PARAMETER (CHUNKSIZE=100)
       REAL A(N), B(N), C(N)

!      Some initializations
       DO I = 1, N
         A(I) = I * 1.0
         B(I) = A(I)
       ENDDO
       CHUNK = CHUNKSIZE
```

```
!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE
(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

      END
```

Worksharing Construct - Reduction a brief (very brief) intro

```
#pragma omp parallel for reduction(+:s)
 for (i=0; i<N; i++)
    s += f(i);
```

- 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.

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 - in class

Calculate pi.

Compute π by numerical integration. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using Riemann sums.

Let $f(x) = \sqrt{1 - x^2}$ be the function that describes the quarter circle for x = 0 to 1.

Then we can compute:

$$\pi/4 = \sum_{i=0}^{N-1} \Delta x f(x_i) \quad \text{where} \quad x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Exercise 3 - Code

```c
#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;
```

```c
  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 - 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 3 - 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?