More work sharing

OpenMP loop scheduling

The default

The default behavior for a for loop is to divide the iterations by the number of threads and assign each thread that number of iterations, in order

What about unequal load?

What happens if the load at each iteration is unequal?

One thread is busy and the others, having finished, just waiting around

nasty progressive function

```
double f (int iters){
  double result = 0.0;

  for (int j = -iters; j<= iters; ++j)
    result += sqrt(atan(j));

  return result;
}
```

Range grows with i. Bigger i, longer time.

atan and sqrt can't really be optimized.

```
 double t1 = omp_get_wtime();
# pragma omp parallel for
num_threads(t_cnt) reduction(+:
result)
   for(int i=0; i<iters; ++i)
      result  += f(i);
   double t2 = omp_get_wtime();
```

Work is unevenly distributed, threads at the end do much more work.

schedule

Can adapt the "schedule" of thread to work in a couple of ways:

- static
- dynamic
- guided
- auto

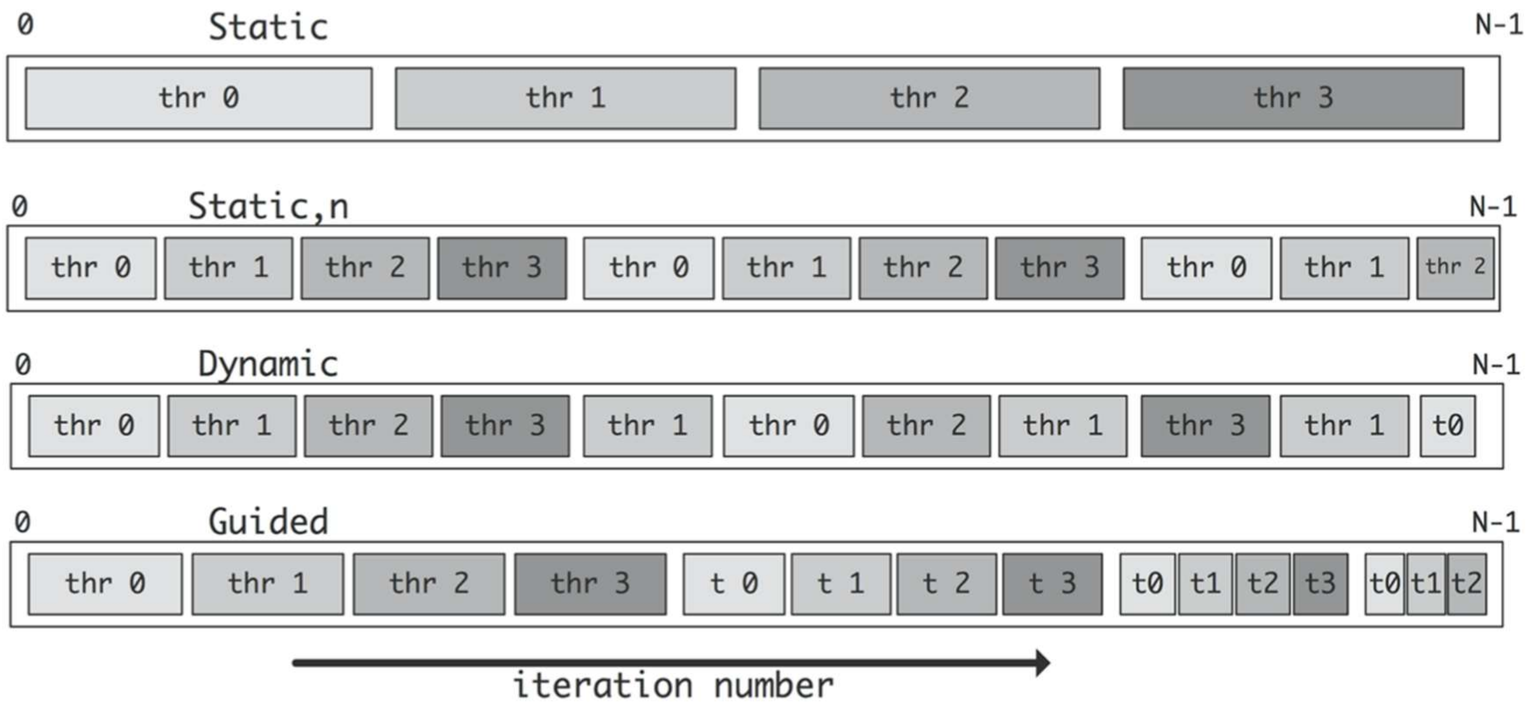


Figure 16.3: Illustration of the scheduling strategies of loop iterations

static

- takes a chunk size (default 1)
- Loop iterations are divided into equal sized pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iteration are evenly (if possible) divided contiguously among the threads

dynamic

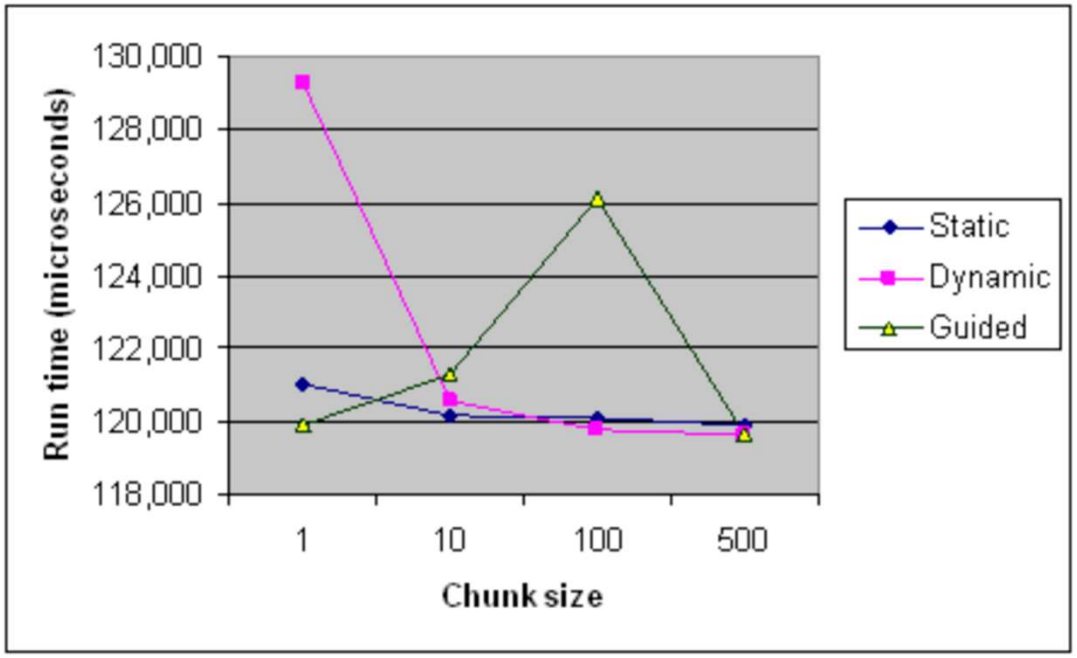
- chunk size defaults to 1
- Loop iterations are divided into pieces of size chunk and then dynamically assigned to threads. When a thread finishes one chunk, it is dynamically assigned another.
- must be careful with this, lot of overhead implied here

guided

- For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations (dynamic) divided by the number of threads, decreasing to 1. For a chunk size with value
- $k (k > 1)$, the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations
- even more overhead

It's complicated

- See, e.g., https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling



For the progressive problem

What do you get?

By default, worksharing for loops end with an implicit barrier

- *nowait*: If specified, threads do not synchronize at the end of the parallel loop
- *ordered*: specifies that the iteration of the loop must be executed as they would be in serial program.
- *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 iteration in all associated loops determines the order of the iterations in the collapsed iteration space.

sections and tasks

Other worksharing forms

Clearly the for worksharing is a great convenience for an OpenMP program. However there are others that are a little more free form:

- sections
- tasks

Sections

If you know how many elements of work you have to do and would like to divide it up, sections is for you

- use the sections pragma to start off the division
- each section pragma is an independent piece of work that can be run by a thread in the team

Tasks

- first appeared in OpenMP 3
- a kind of dynamic, independent scheduling of tasks that get assigned to threads.
- allows for recursion.

dynamic

With sections, you have to know how many you have.

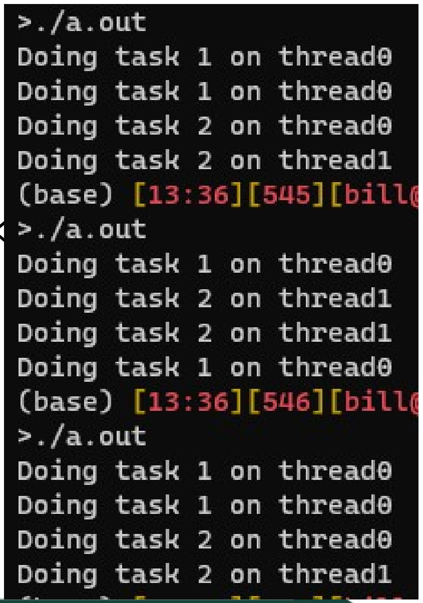
With tasks, you are "generating" a dynamic list of tasks that get thread scheduled.

```
int main (){

#pragma omp parallel num_threads(2) default(none) shared(cout)
  {
#pragma omp task
    {
      stringstream s;
      s << "Doing task 1 on thread" << omp_get_thread_num()  << endl;
      cout << s.str();
    }

#pragma omp task
    {
      stringstream s;
      s << "Doing task 2 on thread" << omp_get_thread_num() <
      cout << s.str();
    }

  } // of parallel
}
```



```
>./a.out
Doing task 1 on thread0
Doing task 1 on thread0
Doing task 2 on thread0
Doing task 2 on thread1
(base) [13:36][545][bill@
>./a.out
Doing task 1 on thread0
Doing task 2 on thread1
Doing task 2 on thread1
Doing task 1 on thread0
(base) [13:36][546][bill@
>./a.out
Doing task 1 on thread0
Doing task 1 on thread0
Doing task 2 on thread0
Doing task 2 on thread1
```

two weirdness-es

1. 4 outputs, only two tasks: why?
2. output order, task assignment to thread changes each time

2)

Scheduling of thread to task is up to the implementation

Output order is always screwed up

4 outputs?

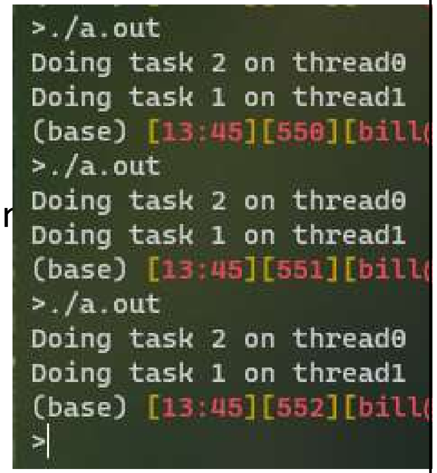
The way this is coded, each thread encounters the parallel section so each thread schedules two tasks.

Probably not what we wanted.

```
int main (){

#pragma omp parallel num_threads(2) default(none) shared(cout)
  {
#pragma omp single
    {
#pragma omp task
      {
    stringstream s;
    s << "Doing task 1 on thread" << omp_get_thread_num()  << endl;
    cout << s.str();
      }

#pragma omp task
      {
    stringstream s;
    s << "Doing task 2 on thread" << omp_get_thread_num() <<er
    cout << s.str();
      }
    } // of single
  } // of parallel
}
```



```
>./a.out
Doing task 2 on thread0
Doing task 1 on thread1
(base) [13:45][550][bill(
>./a.out
Doing task 2 on thread0
Doing task 1 on thread1
(base) [13:45][551][bill(
>./a.out
Doing task 2 on thread0
Doing task 1 on thread1
(base) [13:45][552][bill(
>|
```