Scientific Computing (Phys 2109/Ast 3100H)
III. High Performance Scientific Computing

Lecture 3: More OpenMP

SciNet HPC Consortium, University of Toronto

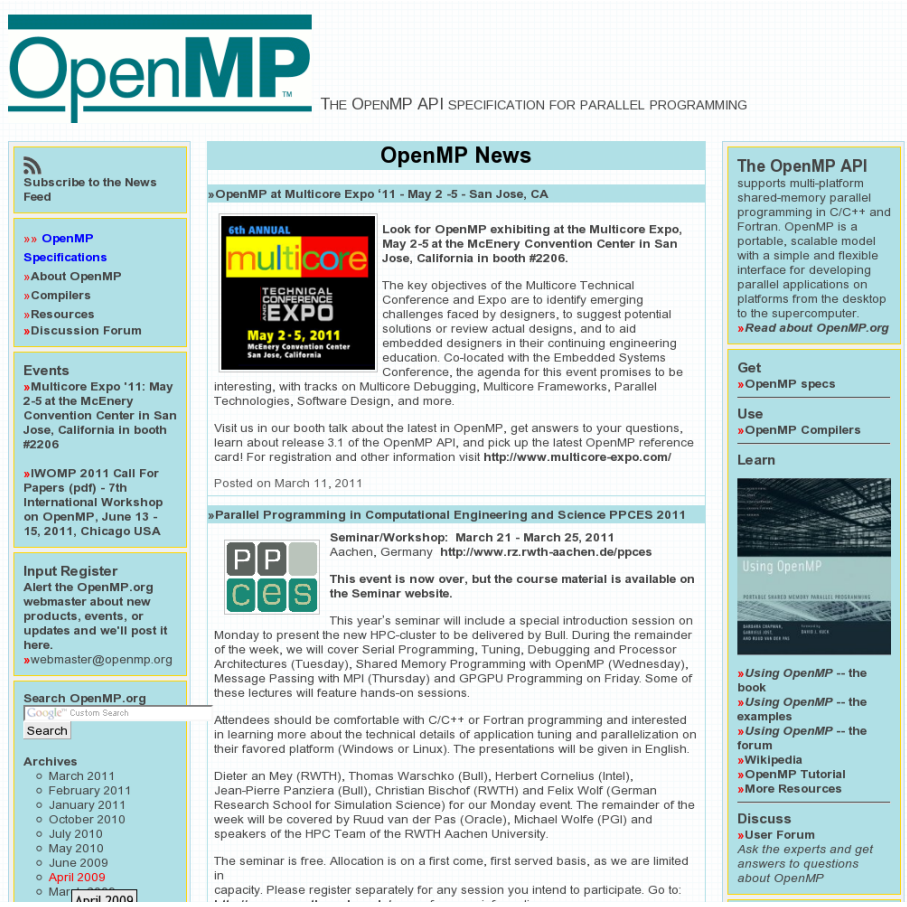
February/March 2012

SciNet compute • calcul CANADA

OpenMP Recap – What was it?

- For shared memory systems.
- Add parallelism to functioning serial code.
- http://openmp.org

- Compiler, run-time environment does a lot of work for us
- Divides up work
- But we have to tell it how to use variables, where to run in parallel, . . .
- Mark parallel regions.
- Works by adding compiler directives to code.
  Invisible to non-openmp compilers.



SciNet compute • calcul CANADA

OpenMP Recap – How do you use it?

In code:

- In C/C++, you add lines starting with `#pragma omp`.
  This starts parallel threads, each executing the subsequent code block.
- These lines are skipped (sometimes with a warning) by compilers that do not support OpenMP.

When compiling:

- To turn on OpenMP support in gcc/g++, add the `-fopenmp` flag to the compilation and link commands.

  (gcc $\geq$ 4.4 supports OpenMP 3.0, gcc 4.7 supports 3.1)

When running:

- The environment variable `OMP_NUM_THREADS` determines how many threads will be started in an OpenMP parallel block.

OpenMP Recap - OpenMP Functions from omp.h

- `void omp_set_num_threads(int nthreads);` set # threads to use next
- `int omp_get_thread_num();` get current thread number
- `int omp_get_num_threads();` get current # threads
- `void omp_set_num_threads(int nthreads);` set # threads to use next
- `int omp_get_num_procs();` get # processors available

Example

```
#include <stdio.h>
#include <omp.h>
int main() {
  printf("At start of program\n");
  #pragma omp parallel
  {
    printf("Hello world from thread %d of %d!\n",
        omp_get_thread_num(),
        omp_get_num_threads());
  }
}
```

OpenMP Recap - Variables and the Memory Model

Variables in parallel regions are a bit tricky, because there are different memory 'regions':

- A shared memory space, accessible to all threads
- A private memory for each thread, created when thread starts, and destroyed when it's done.
- (even private memory for each task, later)

Variables that are declared in the serial part have varying defaults for whether they become shared or private in the parallel region.

- Declare variables locally whenever possible!

  Automatically thread private, avoids many bugs.
- For all other variables used in the parallel regions, specify there access (shared, etc).
- Use `default(none)` to avoid bugs.

OpenMP Recap – Work Sharing in Loops

- We don't generally want tasks to do exactly the same thing.
- Want to partition a problem into pieces, each thread works on a piece.
- Most scientific programming full of work-heavy loops.
- OpenMP has a worksharing construct: omp for.

Example

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
  #pragma omp parallel default(none) shared(n,x,y,a,z)
  #pragma omp for
  for (int i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}
```

OpenMP Recap – Work Sharing in Loops

- We don't generally want tasks to do exactly the same thing.
- Want to partition a problem into pieces, each thread works on a piece.
- Most scientific programming full of work-heavy loops.
- OpenMP has a worksharing construct: omp for.

Example

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
  #pragma omp parallel for default(none) shared(n,x,y,a,z)
  for (int i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}
```

Conclusion Recap

So far, we know how to:

- Start a threaded parallel region:
  `#pragma omp parallel`
- Deal with basic variable scope:
  `default() shared() private()`
- Parallelize a loop:
  `#pragma omp for`
- In a parallel region, have only one process do something:
  `#pragma omp single`

This is (almost) enough to be able to OpenMP-parallelize the diffusion code.

OpenMP version of the diffusion code

1. Temperature evolution

```
 for  (i=1; i<totpoints+1; i++) {
  temperature[new][i] = temperature[old][i] + dt*kappa/(dx*dx) *
    (temperature[old][i+1] - 2.*temperature[old][i] +
    temperature[old][i-1]);
}
```

2. Compute theoretical

```
for  (i=0; i<totpoints+2; i++)
  theory[i] = a*exp(-(x[i]*x[i]) / (2.*sigma*sigma));
```

3. Compute error

```
error = 0.;

for  (i=1;i<totpoints+1;i++)
  error += (theory[i] - temperature[new][i])
    *(theory[i] - temperature[new][i]);
```

OpenMP version of the diffusion code

1. Temperature evolution

```
#pragma omp parallel for default(shared) private(i)
 for  (i=1; i<totpoints+1; i++) {
  temperature[new][i] = temperature[old][i] + dt*kappa/(dx*dx) *
    (temperature[old][i+1] - 2.*temperature[old][i] +
    temperature[old][i-1]);
}
```

2. Compute theoretical

```
for  (i=0; i<totpoints+2; i++)
  theory[i] = a*exp(-(x[i]*x[i]) / (2.*sigma*sigma));
```

3. Compute error

```
error = 0.;

for  (i=1;i<totpoints+1;i++)
  error += (theory[i] - temperature[new][i])
    *(theory[i] - temperature[new][i]);
```

OpenMP version of the diffusion code

1. Temperature evolution

```
#pragma omp parallel for default(shared) private(i)
 for  (i=1; i<totpoints+1; i++) {
  temperature[new][i] = temperature[old][i] + dt*kappa/(dx*dx) *
    (temperature[old][i+1] - 2.*temperature[old][i] +
    temperature[old][i-1]);
}
```

2. Compute theoretical

```
#pragma omp parallel for default(shared) private(i)
for  (i=0; i<totpoints+2; i++)
  theory[i] = a*exp(-(x[i]*x[i]) / (2.*sigma*sigma));
```

3. Compute error

```
error = 0.;

for  (i=1;i<totpoints+1;i++)
  error += (theory[i] - temperature[new][i])
    *(theory[i] - temperature[new][i]);
```

OpenMP version of the diffusion code

1. Temperature evolution

```
#pragma omp parallel for default(shared) private(i)
 for  (i=1; i<totpoints+1; i++) {
  temperature[new][i] = temperature[old][i] + dt*kappa/(dx*dx) *
    (temperature[old][i+1] - 2.*temperature[old][i] +
    temperature[old][i-1]);
}
```

2. Compute theoretical

```
#pragma omp parallel for default(shared) private(i)
for  (i=0; i<totpoints+2; i++)
  theory[i] = a*exp(-(x[i]*x[i]) / (2.*sigma*sigma));
```

3. Compute error

```
error = 0.;
#pragma omp parallel for default(shared) private(i) ???(error)
for  (i=1;i<totpoints+1;i++)
  error += (theory[i] - temperature[new][i])
    *(theory[i] - temperature[new][i]);
```

Remainder of this lecture:

- Data dependences: Reductions.
- Getting performance: load balancing, memory access, hybrid
- Non-loop/non-array constructs

Dot Product

- Dot product of two vectors
- Start from a serial implementation, then will add with OpenMP
- git clone /scinet/course/sc3/lc3
- code in ndot.c
- `$ source setup`
- `$ make ndot`
- `$ ./ndot`
- Tells time, answer, correct answer.

$$n = \vec{x} \cdot \vec{y} = \sum_i x_i y_i$$

```
$ source setup
$ make ndot
$ ./ndot
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 4.9254e-02 seconds.
```

SciNet compute • calcul CANADA

Dot Product - serial

```
#include <stdio.h>
#include "pca_utils.h"
double ndot(int n, double *x, double *y){
  double tot = 0;
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
int main() {
  int n=1e7;
  double *x = vector(n), *y = vector(n);
  for (int i=0; i<n; i++)
    x[i] = y[i] = i;
  double nn=n-1;
  double ans=nn*(nn+1)*(2*nn+1)/6.0;
  pca_time tt;
  tick(&tt);
  double dot=ndot(n,x,y);
  printf("Dot product is %14.4e (vs %14.4e) for n=%d. Took %12.4e secs.\n",
    dot, ans, n, tocksilent(&tt));
}
```

Dot Product - serial

```
#include <stdio.h>
#include "pca_utils.h"
double ndot(int n, double *x, double *y){
  double tot = 0;
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
int main() {
  int n=1e7;
  double *x = vector(n), *y = vector(n);
  for (int i=0; i<n; i++)
    x[i] = y[i] = i;
  double nn=n-1;
  double ans=nn*(nn+1)*(2*nn+1)/6.0;
  pca_time tt;
  tick(&tt);
  double dot=ndot(n,x,y);
  printf("Dot product is %14.4e (vs %14.4e) for n=%d. Took %12.4e secs.\n",
    dot, ans, n, tocksilent(&tt));
}
```

```
$ make ndot
$ ./ndot
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 4.9254e-02 secs.
```

Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We need the sum from everybody.
- We could make `tot` shared, then all threads can add to it.

```
double ndot(int n, double *x, double *y) {
  double tot = 0;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We need the sum from everybody.
- We could make `tot` shared, then all threads can add to it.

```
double ndot(int n, double *x, double *y) {
  double tot = 0;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_race
Dot product is 1.1290e+20
(vs 3.3333e+20) for n=10000000.
Took 5.2628e-02 secs.
```

Not only is the answer wrong, it was slower to compute!

Race Condition - why it's wrong

- Classical parallel bug.
- Multiple writers to some shared resource.
- Can be very subtle, and only appear intermittently.
- Your program can have a bug but not display any symptoms for small runs!
- Primarily a problem with shared memory.

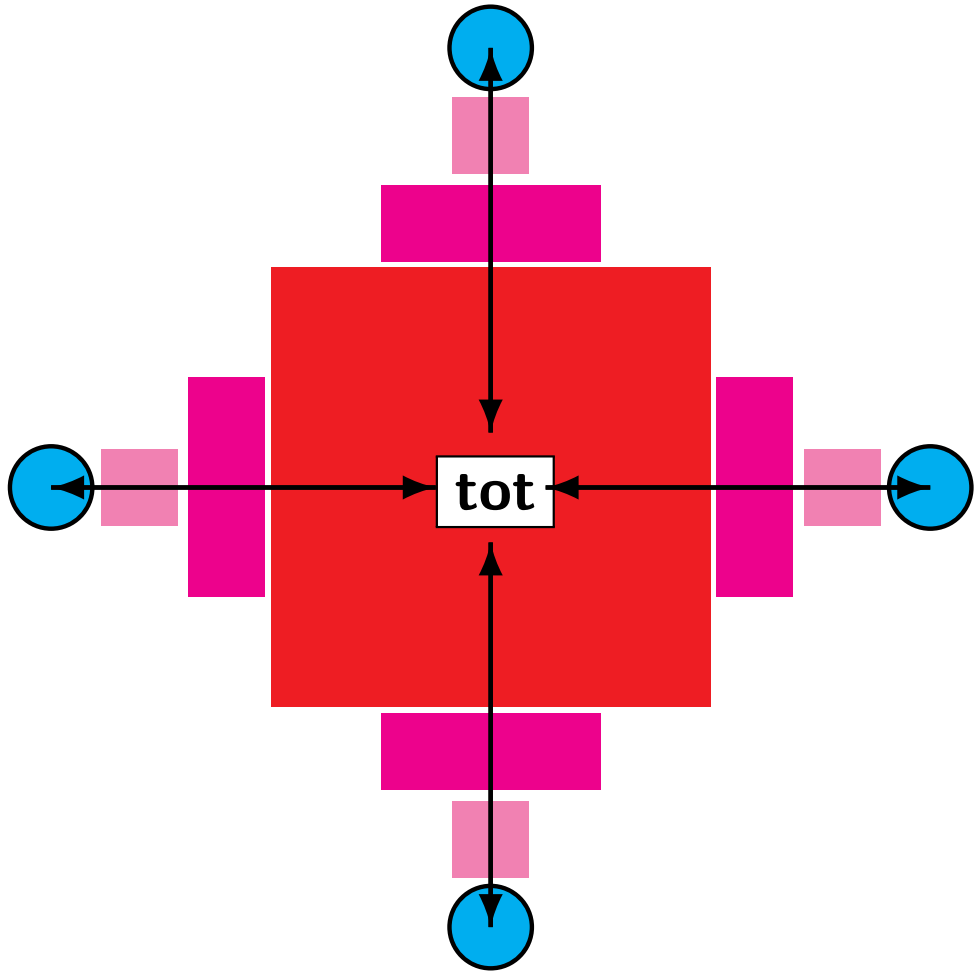
`tot` = 0

Thread 0: add 1	**Thread 1: add 2**
read `tot`(=0) into register	
reg = reg+1	read `tot`(=0) into register
store reg(=1) into `tot`	reg=reg+2
	store reg(=2) into `tot`

`tot` = 2

Race Condition - why it's slow

- Multiple cores repeatedly trying to read, access, store same variable in memory.
- Not (such) a problem for constants (read only); but a big problem for writing.
- Sections of arrays – better.



OpenMP critical construct

- Defines a critical region.
- Only one thread can be operating within this region at a time.
- Keeps modifications to shared resources saffe.
- `#pragma omp critical`

```
double ndot(int n,double*x,double*y){
  double tot = 0;
  #pragma omp parallel for \
  default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    #pragma omp critical
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_critical
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_critical
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 5.1377e+00 secs.
```

Correct, but 100x slower than serial version!

OpenMP atomic construct

- Most hardware has support for atomic instructions (indivisible so cannot get interrupted)
- Small subset, but load/add/stor usually one.
- Not as general as critical
- Much lower overhead.
- `#pragma omp atomic`

```
double ndot(int n,double*x,double*y){
  double tot = 0;
  #pragma omp parallel for \
  default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    #pragma omp atomic
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_atomic
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_atomic
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 8.5156e-01 secs.
```

Correct, and better – only 16x slower than serial.

How should we fix the slowdown?

- Local sums.
- Each processor sums its local values ($\mathbf{10^7/P}$ additions).
- And sums those to `tot` (only **P** additions with critical or atomic)

How should we fix the slowdown?

- Local sums.
- Each processor sums its local values ($\mathbf{10^7/P}$ additions).
- And sums those to `tot` (only $\mathbf{P}$ additions with critical or atomic)

$$\begin{aligned} \mathbf{n} &= \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} \\ &= \sum_{\mathbf{i}} \mathbf{x_i}\, \mathbf{y_i} \\ &= \sum_{\mathbf{p}} \left(\sum_{\mathbf{i}} \mathbf{x_i}\, \mathbf{y_i} \right) \end{aligned}$$

How should we fix the slowdown?

- Local sums.
- Each processor sums its local values ($\mathbf{10^7/P}$ additions).
- And sums those to `tot` (only **P** additions with critical or atomic)

$$\begin{aligned} \mathbf{n} &= \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} \\ &= \sum_{\mathbf{i}} \mathbf{x_i}\, \mathbf{y_i} \\ &= \sum_{\mathbf{p}} \left(\sum_{\mathbf{i}} \mathbf{x_i}\, \mathbf{y_i} \right) \end{aligned}$$

HANDS-ON: Try it!

Parallelize ndot with partial sums.

As a starting point, take omp_ndot_local.c (a copy of omp_ndot_race.c)

Local variables solution

```
double tot = 0;
#pragma omp parallel shared(x,y,n,tot)
{
  int mytot = 0;
  #pragma omp for
  for (int i=0; i<n; i++)
    mytot += x[i]*y[i];
  #pragma omp atomic
  tot += mytot;
}
```

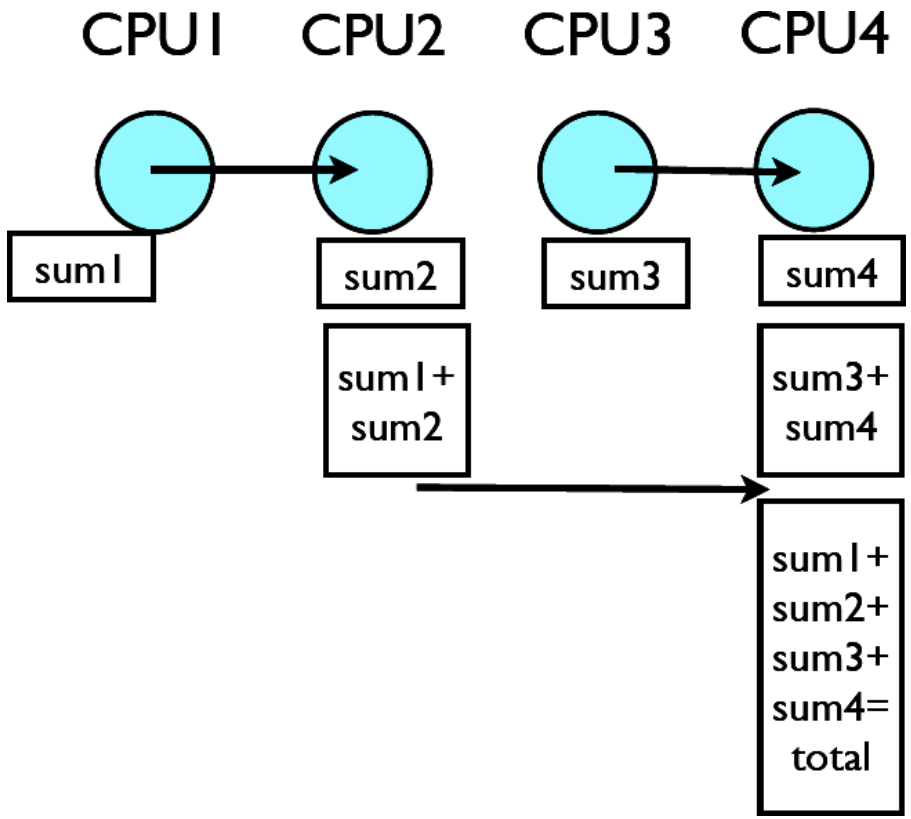
```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.7902-02 seconds.
```

Now we're talking!

2.77x faster.

OpenMP Reduction Operations

- This is such a common operation, this is something built into OpenMP to handle it.
- "Reduction" variables - like shared or private.
- Can support several types of operations: - + * ...
- omp_ndot_reduction.c



OpenMP Reduction Operations

```
double tot = 0;
#pragma omp parallel for \
shared(x,y,n) reduction(+:tot)
for (int i=0; i<n; i++)
  tot += x[i]*y[i];
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.8928e-02 seconds.
```

Same speed, simpler code!

OpenMP Reduction Operations

```
double tot = 0;
#pragma omp parallel for \
shared(x,y,n) reduction(+:tot)
for (int i=0; i<n; i++)
  tot += x[i]*y[i];
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.8928e-02 seconds.
```

Same speed, simpler code!

Reduction operators:

`+ - *`	arithmatic
`&& ||`	logical: and, or
`& | ^`	bitwise and, or, xor
`min max`	extremal values (in C in openmp 3.1)

Performance

- We threw in 8 cores, got a factor of 3 speedup. Why?
- Often we are limited not by CPU power but by how quickly we can feed CPUs.
- For this problem, we had $\mathbf{10^7}$-long vectors, with 2 numbers of 8 bytes long flowing through in 0.036 seconds.
- Combined bandwidth from main memory was 4.3 GB/s. Not far off of what we could hope for on this architecture.
- One of the keys to good OpenMP performance is using data when we have it in cache. Complicated functions: easy. Low work-per-element (dot product, FFT): hard.

A bit more on variables

- We had:
  #pragma omp ... shared(), private(), and reduction.
- Want private variable to get value from the serial part?
  Use firstprivate():

```
#include <stdio.h>
int main() {
  int n = 0;
  #pragma omp parallel firstprivate(n)
  {
    #pragma omp for
    for (int i=0;i<100;i++)
      n++;
    printf("My n=%\n",n);
  }
}
```

A bit more on variables

- Private variables are destroyed after parallel region.
- What if you want the result of a private variable to be preserved?
  Use lastprivate:

```
#include <stdio.h>
int main() {
  int n;
  #pragma omp parallel for lastprivate(n)
  for (int i=0;i<100;i++)
    if (i>70) n=i;
  printf("Last n was %\",n);
}
```

Load Balancing in OpenMP

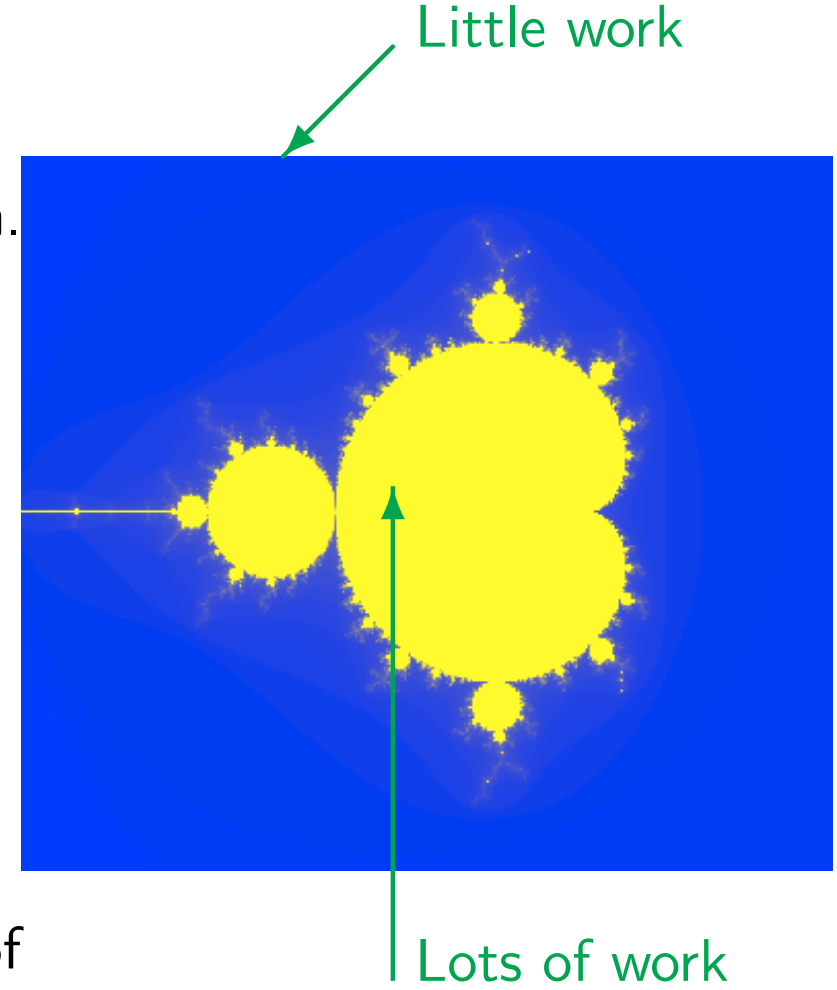
- So far every iteration of the loop had the same amount of work.
- This is not always the case.
- Sometimes cannot predict beforehand how unbalanced the problem is.

OpenMP has work sharing construct that allow you do statically or dynamically balance the load.

Example - Mandelbrot Set

- Mandelbrot set example of non-balanced problem.
- Defined as complex points **a** where $|\mathbf{b}_\infty|$ finite, with $\mathbf{b_0} = \mathbf{0}$ and $\mathbf{b_{n+1}} = \mathbf{b_n^2} + \mathbf{a}$. If $|\mathbf{b_n}| > \mathbf{2}$, point diverges.
- Calculation:
  - pick some **nmax**
  - iterate for each point **a**, see if crosses 2.
  - Plot **n** or **nmax** as colour.

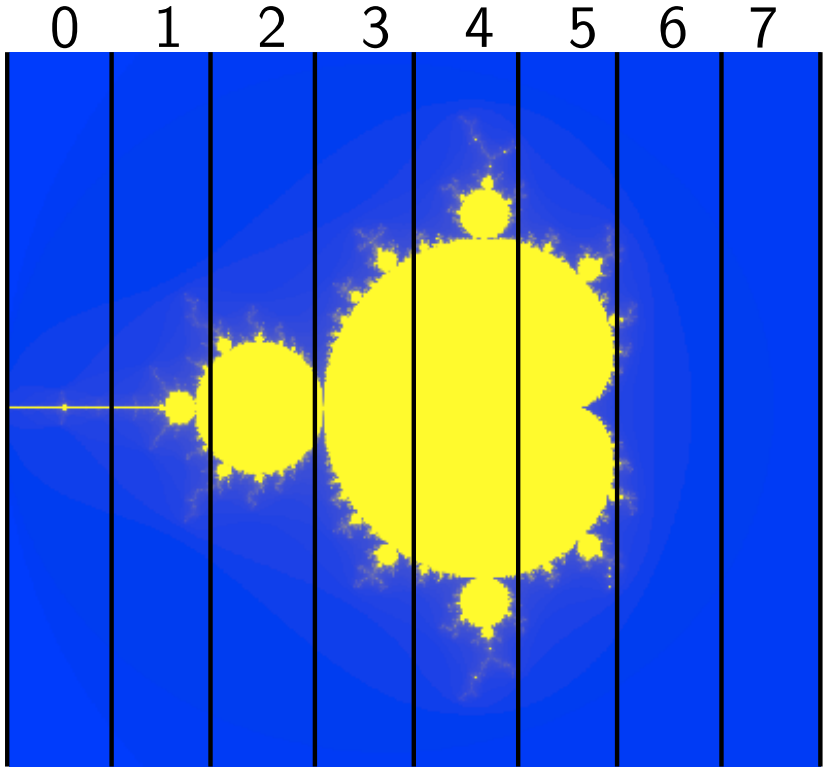
  Outside of set, points diverge quickly (2-3 steps). Inside, we have to do lots of work (1000s steps).
- make mandel; ./mandel



SciNet compute • calcul CANADA

First OpenMP Mandelbrot Set

- Default work sharing breaks up **N** iterations into $\sum$ **N/nthreads** contiguous chunks and assigns them to threads.
- But now threads 7,6,5 will be done and sitting idle while threads 3 and 4 work alone...
- Inefficient use of resources.

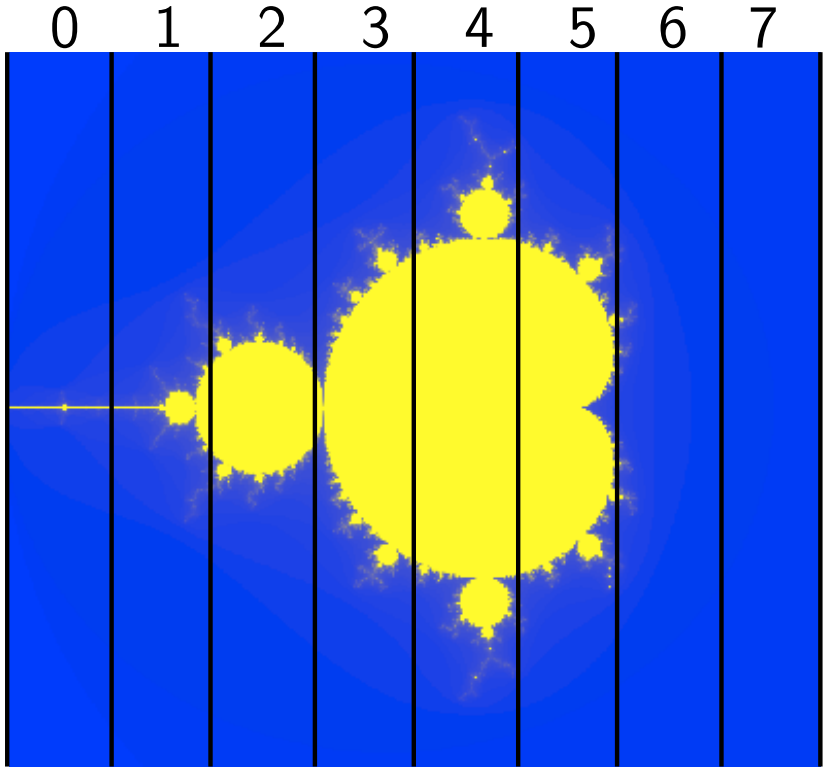


800x800 pix; N/nthreads 100x800

First OpenMP Mandelbrot Set

- Default work sharing breaks up **N** iterations into $\sum$ **N/nthreads** contiguous chunks and assigns them to threads.
- But now threads 7,6,5 will be done and sitting idle while threads 3 and 4 work alone...
- Inefficient use of resources.

Serial	0.63s
Nthreads=8	0.29s
Speedup	2.2x
Efficiency	27%

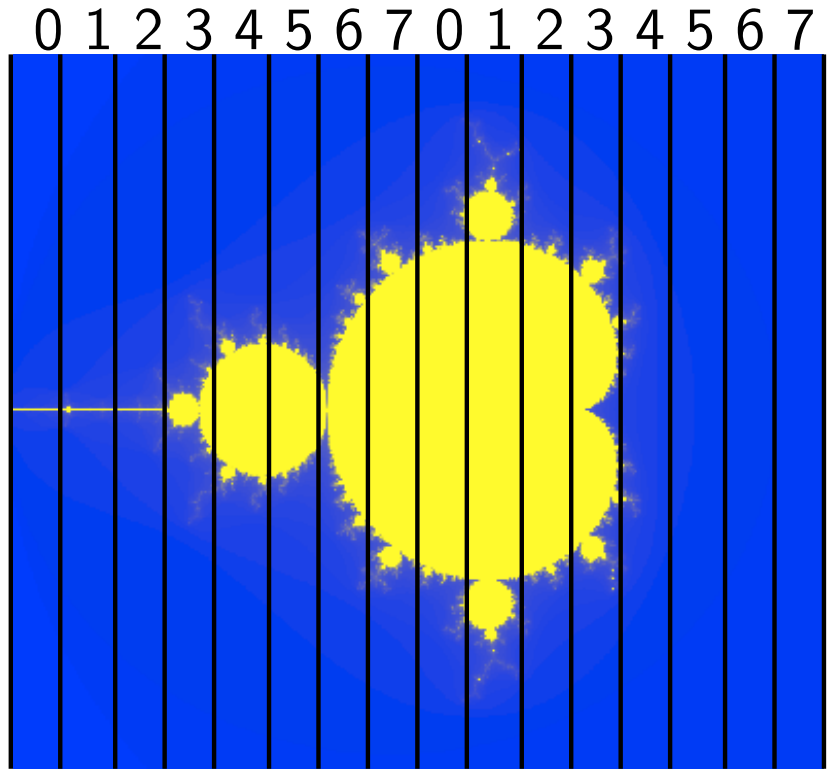


800x800 pix; N/nthreads 100x800

SciNet compute • calcul CANADA

Second Try OpenMP Mandelbrot Set

- Can change the chunk size to be different from **N/nthreads**:
  `#pragma omp for schedule(static,50)`
- In this case, more columns; work better distributed.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:



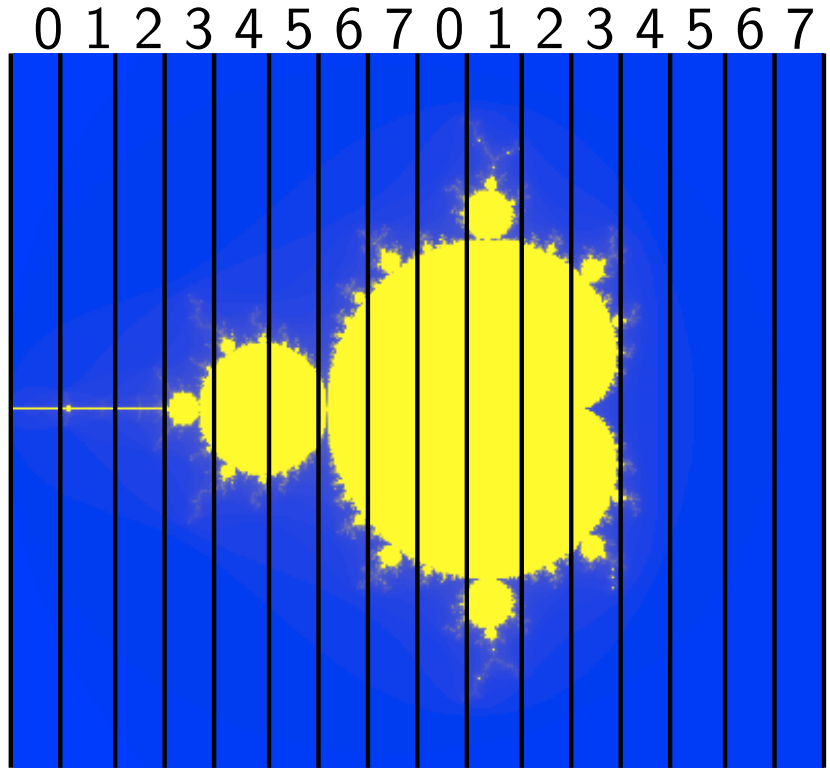
800x800 pix; each thread: 50x800

SciNet compute • calcul CANADA

Second Try OpenMP Mandelbrot Set

- Can change the chunk size to be different from **N/nthreads**:
  `#pragma omp for schedule(static,50)`
- In this case, more columns; work better distributed.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:

Serial	0.63s
Nthreads=8	0.15s
Speedup	4.2x
Efficiency	52%



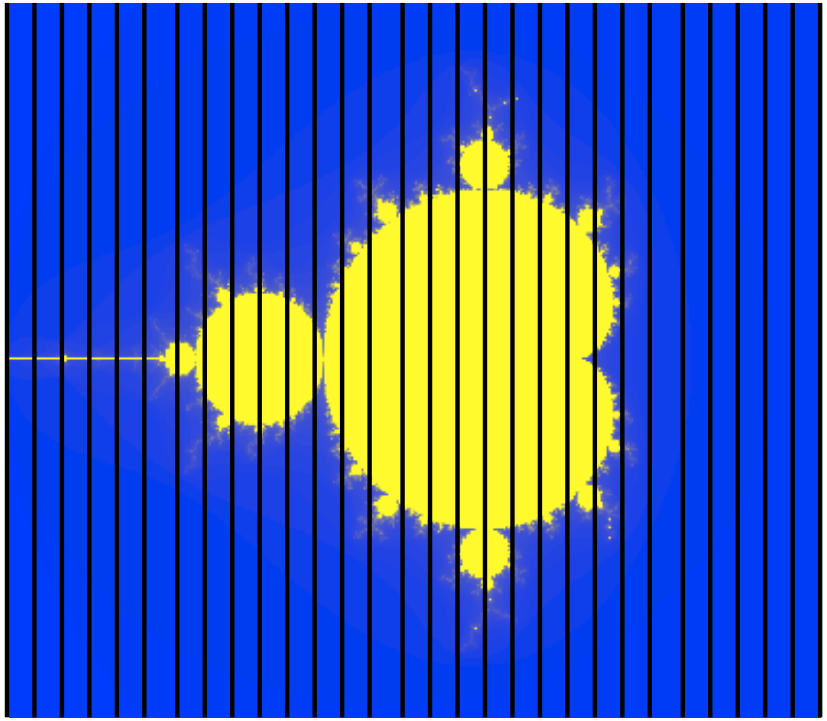
800x800 pix; each thread: 50x800

SciNet compute • calcul CANADA

Third Try: Schedule dynamic

- Break up into many pieces and hand them to threads when they are ready:

```
#pragma omp for schedule(dynamic)
```

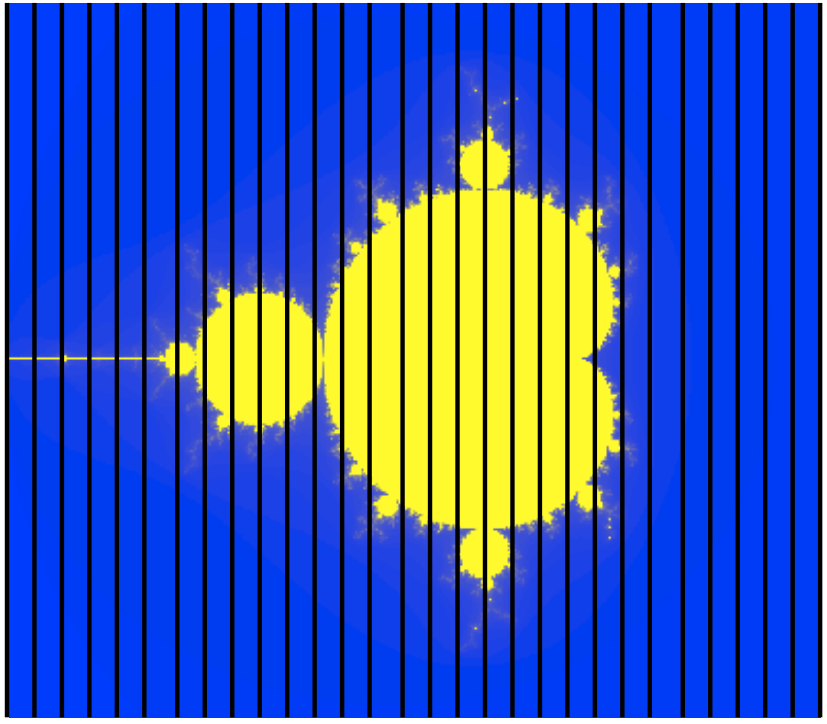


- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

Third Try: Schedule dynamic

- Break up into many pieces and hand them to threads when they are ready:

```
#pragma omp for schedule(dynamic)
```



- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

Serial	0.63s
Nthreads=8	0.10s
Speedup	6.3x
Efficiency	79%

Tuning

- schedule(static) (default) or schedule(dynamic) are good starting points.
- To get best performance in badly imbalanced problems, may have to play with chunk size; depends on your problem and on hardware.

(static,4)	(dynamic,16)
0.084s	0.099s
7/6x	6.4x
95%	79%

Two level loops

In scientific code, we usually have nested loopes were all the work is.

Almost without exception, want the pragma on the outside-most loop.

Why?

```
#pragma omp for schedule(static,4)
for (int i=0;i<npix;i++)
  for (int j=0;j<npix;j++){
    double x=i/(double)npix;
    double y=j/(double)npix;
    double complex a=x+I*y;
    mymap[i][j]=f(a,maxiter);
  }
```

Style Points

- If a variable is a private temporary variable inside a parallel region, try declaring it inside the region.
  Makes parallel region easier to specify, and can prevent bugs.
- OpenMP supports reduction and initialization clauses. These are never necessary to use, but are convenient and can streamline code.
- You have seen how to find out how many threads exist, etc. However, in none of our examples did we use that info.
  If you think you need to know how many threads you have, you may well be doing something wrong (with some notable exceptions such as complex reduction). Using locally declared variables, and critical regions most likely will do everything you need.

A Few More Directives

- #pragma omp ordered - execute the loop in the order it would have run serially. Useful if you want ordered output in a parallel region. Never useful for performance.
- #pragma omp master - a block that only the master thread (thread 0) executes. Usually, #pragma omp single is better.
- #pragma omp sections - execute a list of things in parallel. In OpenMP 3, task directive (later in lecture) is more powerful
- #pragma omp for collapse(n): nested loops scheduled as one big loop.

Memory Access

- Memory access is important for serial programs, but can become particularly important in OpenMP
- There is typically a limited bandwidth to main memory. If it has to be shared 2, 4, or 8 ways, it becomes especially critical to access it sensibly.
- Note on shared variables in OpenMP: If you aren't changing them, the compiler can copy the shared variable to local cache and no performance hit. Modifying shared variables is expensive - we have already seen this with the dot product.

Conditional OpenMP

- There is always overhead associated with starting threads, splitting work, etc. Also, some jobs parallelize better than others.
- Sometimes, overhead takes longer than 1 thread would need to do a job - e.g. very small matrix multiplies.
- OpenMP supports conditional parallelization. Add if(condition) to parallel region beginning. So, for small tasks, overhead low, while large tasks remain parallel.

Conditional OpenMP in Action

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  #pragma omp parallel if (n>10)
  #pragma omp single
    printf("have %d
    threads with n=%d\n",
    omp_get_num_threads(),n);
}
```

```
$ ./conditional_if 12
have 8 threads with n=12
$ ./conditional_if 9
have 1 threads with n=9
$
```

First, pull an integer from the command line. Check to see if it's bigger than a number (in this case, 10). If so, start a parallel region. Otherwise, execute serially.

Controlling # of Threads

- Sometimes you might want more or fewer threads. May even want to change while running.
- Example - TCS cluster. Matrix multiply runs fast with twice as many program threads as physical cores (hyperthreading). However, matrix factorizations run slower with more threads.
- omp_set_num_threads(int) sets or changes the number of threads during runtime.

SciNet
compute • calcul
CANADA

omp_set_num_threads() in action

```
#include "stdio.h"
#include "omp.h"
int main(int argc,char *argv[]){
  //find # of physical cores
  //(an openmp library routine)
  int maxthreads=omp_get_num_procs();
  int n=atoi(argv[1]);
  //set # threads equal to input
  //if it's less than maxthreads
  if (n<maxthreads)
    omp_set_num_threads(n);
  else
    omp_set_num_threads(maxthreads);
  #pragma omp parallel
  #pragma omp single
  printf("Running with %d
  threads for n=%d.\n",
  omp_get_num_threads(),n)
}
```

We have changed the # of threads during the program. We could always change the number later on in the same code, if we so desired.

Note the use of `omp_get_num_procs`, a library call to detect the number of available processors to the OS.

Non-loop construct

OpenMP supports non-loop parallelism as well:

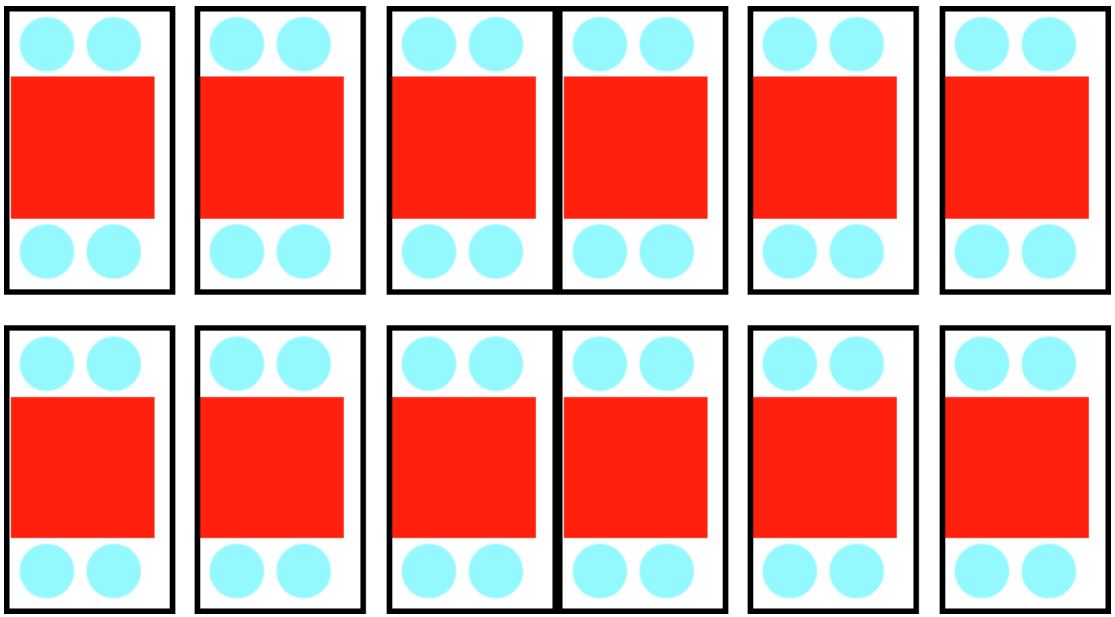
- Sections:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      something to do
    }
    #pragma omp section
    {
      something to do at the
      same time
    }
  }
}
```

- More flexible: tasks

Tasks

- OpenMP $\geq$ 3.0 supports the #pragma omp task directive.
- A task is a job assigned to a thread. Powerful way of parallelizing non-loop problems.
- Tasks should help omp/mpi hybrid codes - one task can do communications, rest of threads keep working.
- Like all omp, tasks must be called from parallel region.
- Raises complication of nested parallelism (what happens if a parallel loop called from parallel loop?).



SciNet compute • calcul CANADA

Tasks: test_task.c

```
#include <stdio.h>
#include <omp.h>
int main(){
  #pragma omp parallel
  #pragma omp single
  {
    printf("hello");
    #pragma omp task
    {
      printf("hello 1 from
      %d.",omp_get_thread_num());
    }
    #pragma omp task
    printf("hello 2 from
    %d.",omp_get_thread_num());
  }
}
```

Often want to start tasks from as if from serial region. Must be in parallel for tasks to spawn, so #pragma omp parallel followed by #pragma omp single very useful. What would happen w/out #pragma omp single?

Beauty of Tasks

- Some otherwise-hard-to-parallelize problems fit well into tasks.
- Example (from standard): parallel tree processing.
- Each node has left, right pointers, process each subpointer with a task.
- Look how short the parallel tree is!
- Works for a variety of non-array structure (linked lists, etc.)

```
typedef struct node {
  struct node *left, right;
  ...
};
void traverse(struct node* p) {
  if (p->left)
    #pragma omp task firstprivate(p)
    traverse(p->left);
  if (p->right)
    #pragma omp task firstprivate(p)
    traverse(p->right);
  process(p);
}
```

Parallel traversal starts as follows:

```
int process_tree(struct node* root)
{
  #pragma omp parallel
  #pragma omp single
  traverse(root);
}
```

How'd you do this without tasks?

Beauty of Tasks #2

Linked list:

```
typedef struct node {
  struct node *next;
  ...
};
void traverse_linked_list(struct node* head) {
  #pragma omp parallel
  #pragma omp single
  {
    struct node* n = head;
    while (n != NULL) {
      #pragma omp task firstprivate(n)
      process(n);
      n = n->next;
    }
  }
}
```

How'd you OpenMP this without tasks?

The Cost of Beauty

- While elegant there's substantial overhead for tasks:
- Need to store code and data together as a package (that's why all the firstprivate clauses are needed).
- Task has to be put in some sort of queue, and executed when a thread is idle.
- In contrast, in a default-scheduled loop, there is only one task per thread.
- Tasks only cost effective if the 'process' is compute-heavy.
- For fairly light tasks, 'serializing' the tree or linked list, i.e., convering it to an array and openmp-ing that may be necessary to get good scaling.