OpenMP

Scientific Computing Lecture 19 and 20

SciNet, University of Toronto

Ramses van Zon

March 18 and 20, 2014

SciNet compute • calcul CANADA

OpenMP

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

OpenMP

- 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.

OpenMP basic operations

In code:

- In C++, you add lines starting with `#pragma omp`. This parallelizes 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 g++, add the `-fopenmp` flag to the compilation and link commands.

When running:

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

To get example code with makefile on SciNet do:

$ git clone /scinet/course/sc3/lc19

$ cd lc19

$ source setup

$ make.

SciNet compute • calcul CANADA

OpenMP example

```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```

SciNet
compute • calcul
CANADA

OpenMP example

```
$ g++ -O2 -o omp-hello-world omp-hello-world.cc -fopenmp

$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
...
```

Let's see what happens. . .

SciNet compute • calcul CANADA

OpenMP example

```
$ g++ -O2 -o omp-hello-world omp-hello-world.cc -fopenmp
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
Hello, world, from thread 6!
Hello, world, from thread 5!
Hello, world, from thread 4!
Hello, world, from thread 2!
Hello, world, from thread 1!
Hello, world, from thread 7!
Hello, world, from thread 3!
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
```

iNet
pute • calcul
CANADA

So what happened precisely?

- OMP_NUM_THREADS threads were launched.
- Each prints “Hello, world ...”;
- In seemingly random order.
- Only one “At start of program”.

```
$ g++ -O2 -o omp-hello-world omp-he
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
Hello, world, from thread 6!
Hello, world, from thread 5!
Hello, world, from thread 4!
Hello, world, from thread 2!
Hello, world, from thread 1!
Hello, world, from thread 7!
Hello, world, from thread 3!
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
```

So what happened precisely?

```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```

SciNet compute • calcul CANADA

So what happened precisely?

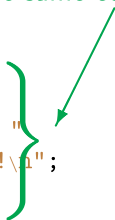
```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```

Program starts normally (single thread)

SciNet compute • calcul CANADA

So what happened precisely?

```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```



At start of parallel section, launching OMP_NUM_THREADS threads,
Each executes the same code!

So what happened precisely?

```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```

At end of parallel section, threads join back up, Execution continues serially.

SciNet compute • calcul CANADA

So what happened precisely?

```
#include <iostream>
#include <omp.h>
int main(){
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
                << omp_get_thread_num() << "!\n";
   }
}
```

Special function to find number of current thread (first=0).

SciNet compute • calcul CANADA

OpenMP functions (from omp.h)

```
#include <iostream>
#include <omp.h>
int main() {
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
            << omp_get_thread_num() << " of "
            << omp_get_num_threads() << "!\n";
   }
}
```

omp_get_num_threads() called by all threads.

Let's see if we can fix that. . .

SciNet compute • calcul CANADA

OpenMP functions (from omp.h)

```
#include <iostream>
#include <omp.h>
int main() {
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
            << omp_get_thread_num() << "!\n";
   }
   std::cout<<"There were "<<omp_get_num_threads()
         <<" threads.\n";
}
```

SciNet compute • calcul CANADA

OpenMP functions (from omp.h)

```
#include <iostream>
#include <omp.h>
int main() {
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
            << omp_get_thread_num() << "!\n";
   }
   std::cout<<"There were "<<omp_get_num_threads()
         <<" threads.\n";
}
```

Strange, says: "There were 1 threads."

SciNet compute • calcul CANADA

OpenMP functions (from omp.h)

```
#include <iostream>
#include <omp.h>
int main() {
   std::cout << "At start of program\n";
   #pragma omp parallel
   {
      std::cout << "Hello world from thread "
            << omp_get_thread_num() << "!\n";
   }
   std::cout<<"There were "<<omp_get_num_threads()
         <<" threads.\n";
}
```

Strange, says: “There were 1 threads.”

Why?

Because that is true outside the parallel region!

Need to get the value from the parallel region somehow.

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int mythread, nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   private(mythread)
   {
      mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int mythread, nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   private(mythread)
   {
      mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

Variable declarations
How used in parallel region

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int mythread, nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   private(mythread)
   {
      mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

Variable declarations
How used in parallel region

- default(none) can save you hours of debugging!
- shared: each thread sees it and can modify (be careful!).
  Preserves value.
- private: each thread gets it own copy, invisible for others
  Initial and final value undefined!
  (Advanced: firstprivate, lastprivate – copy in/out.)

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int mythread, nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   private(mythread)
   {
      mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

- Program runs, lauches threads.
- Each thread gets copy of mythread.
- Only thread 0 writes to nthreads.

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int mythread, nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   private(mythread)
   {
      mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

- Program runs, lauches threads.
- Each thread gets copy of mythread.
- Only thread 0 writes to nthreads.
- Good idea to declare mythread locally! (avoids many bugs)

SciNet compute • calcul CANADA

Solution Requires Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   {
      int mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

- Program runs, lauches threads.
- Each thread gets copy of mythread.
- Only thread 0 writes to nthreads.
- Good idea to declare mythread locally!
  (avoids many bugs)

SciNet compute • calcul CANADA

Single Execution using Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   {
      int mythread = omp_get_thread_num();
      if (mythread == 0)
         nthreads = omp_get_num_threads();
   }
   std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

- Do we care that it's thread 0 in particular that updates nthreads?
- Often, we just want the first thread to go through, do not care which one.

SciNet compute • calcul CANADA

Single Execution without Variables in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
   int nthreads;
   #pragma omp parallel default(none) shared(nthreads)
   #pragma omp single
      nthreads = omp_get_num_threads();
   std::cout << "There were " << nthreads << "
   threads.\n";
}
```

SciNet compute • calcul CANADA

Loops in OpenMP

Let's add a loop.

SciNet
compute • calcul
CANADA

Loops in OpenMP

Let's add a loop.

```
#include <iostream>
#include <omp.h>
int main() {
   int i, mythread;
   #pragma omp parallel default(none) \
   private(i,mythread) shared(std::cout)
   {
      mythread = omp_get_thread_num();
      for (i=0; i<16; i++)
         std::cout << "Thread " << mythread
                   << " gets i=" << i << "\n";
   }
}
```

SciNet compute • calcul CANADA

Loops in OpenMP

Let's add a loop.

```
#include <iostream>
#include <omp.h>
int main() {
   int i, mythread;
   #pragma omp parallel default(none) \
   private(i,mythread) shared(std::cout)
   {
      mythread = omp_get_thread_num();
      for (i=0; i<16; i++)
         std::cout << "Thread " << mythread
                       << " gets i=" << i << "\n";
   }
}
```

What would you imagine this does when run with e.g. OMP_NUM_THREADS=8?

SciNet compute • calcul CANADA

Worksharing constructs in OpenMP

- 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.

SciNet compute • calcul CANADA

Worksharing constructs in OpenMP

- 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.

```
#include <iostream>
#include <omp.h>
int main() {
   #pragma omp parallel default(none) shared(std::cout)
   {
      int mythread = omp_get_thread_num();
      #pragma omp for
      for (int i=0; i<16; i++)
         std::cout << "Thread " << mythread
                   << " gets i=" << i << "\n";
   }
}
```

Worksharing constructs in OpenMP

- omp for construct breaks up the iterations by thread.
- If doesn't divide evenly, does the best it can.
- Allows easy breaking up of work!
- Advanced: can break up work of arbitrary blocks of code with omp task construct.

```
./omp_loop
thread 3 gets i= 6
thread 3 gets i= 7
thread 4 gets i= 8
thread 4 gets i= 9
thread 5 gets i= 10
thread 5 gets i= 11
thread 6 gets i= 12
thread 6 gets i= 13
thread 1 gets i= 2
thread 1 gets i= 3
thread 0 gets i= 0
thread 0 gets i= 1
thread 2 gets i= 4
thread 2 gets i= 5
thread 7 gets i= 14
thread 7 gets i= 15
$
```

Less trivial example: DAXPY

- multiply a vector by a scalar, add a vector.
- (a X plus Y, in double precision)
- Will implement this, first serially, then with OpenMP
- daxpy.cc
- make daxpy

$$\mathbf{z} = \mathbf{z} + \mathbf{a}\mathbf{x} + \mathbf{y}$$

Warning

This is a common linear algebra construct that you really shouldn't implement yourself. Various BLAS implementations will do a much better job than you. But good for illustration.

Daxpy - Serial

```
#include "ticktock.h"
void daxpy(int n,double a,double *x,double *y,double *z) {
   for (int i=0; i<n; i++) {
      x[i] = (double)i*(double)i;
      y[i] = ((double)i+1.)*((double)i-1.);
   }
   for (int i=0; i<n; i++)
      z[i] += a * x[i] + y[i];
}
int main() {
   int n=10*1000*1000;
   double*x=new double[n],*y=new double[n],*z=new dou-
   ble[n];
   double a = 5./3.;
   TickTock tt;
   tt.tick();
   daxpy(n,a,x,y,z);
   tt.tock();
   delete [] x; delete [] y; delete [] z;
}
```

SciNet
compute • calcul
CANADA

Daxpy - Parallel

```
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++) {
         x[i] = (double)i*(double)i;
         y[i] = ((double)i+1.)*((double)i-1.);
      }
      #pragma omp for
      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
   }
}
```

SciNet
compute • calcul
CANADA

```
$ make daxpy
$ ./daxpy
Tock registers 0.2427 seconds.

 [add OpenMP]

$ make daxpy-parallel
g++ -c -I/scinet/gpc/Libraries/boost_1_54_0-gcc4.8.1/include -g
-fopenmp -o daxpy-parallel.o daxpy-parallel.cc
g++ -L/scinet/gpc/Libraries/boost_1_54_0-gcc4.8.1/lib -o
daxpy-parallel daxpy-parallel.o -fopenmp -lboost_system
-lboost_chrono

$ export OMP_NUM_THREADS=2
$ ./daxpy-parallel
Tock registers 0.1458 seconds.
```

1.66x speedup, 83% efficiency

```
$ export OMP_NUM_THREADS=4
$ ./daxpy-parallel
Tock registers 0.09254 seconds.
```

2.62x speedup, 66% efficiency

```
$ export OMP_NUM_THREADS=8
$ ./daxpy-parallel
Tock registers 0.06496 seconds.
```

3.74x speedup, 47% efficiency

Dot Product

$$\mathbf{n} = \vec{\mathbf{x}} \cdot \vec{\mathbf{y}} = \sum_{\mathbf{i}} \mathbf{x_i}\,\mathbf{y_i}$$

- Dot product of two vectors
- Start from a serial implementation, then will add OpenMP
- Program tells time, answer correct answer.

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

SciNet compute • calcul CANADA

Dot Product - Serial

```
#include <iostream>
#include "ticktock.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() {
   long int n=10*1000*1000;
   double *x=new double[n], *y=new double[n];
   for (int i=0; i<n; i++) x[i]=y[i]=i;
   double ans=(n-1)*n*(2.0*n-1)/6.0;
   TickTock tt;
   tt.tick();
   double dot=ndot(n,x,y);
   std::cout << "Dot product:  " << dot << " (vs "
             << ans << ") for n=" << n << "\n";
   tt.tock();
   delete [] x; delete [] y;
}
```

SciNet
compute • calcul
CANADA

Dot Product - Serial

```
#include <iostream>
#include "ticktock.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() {
   long int n=10*1000*1000;
   double *x=new double[n], *y=new double[n];
   for (int i=0; i<n; i++) x[i]=y[i]=i;
   double ans=(n-1)*n*(2.0*n-1)/6.0;
   TickTock tt;
   tt.tick();
   double dot=ndot(n,x,y);
   std::cout << "Dot product:  " << dot << " (vs "
             << ans << ") for n=" << n << "\n";
   tt.tock();
   delete [] x; delete [] y;
}
```

```
$ make ndot
$ ./ndot
Dot product:  3.33333e+20
(vs 3.33333e+20) for n=10000000
Tock registers 0.05371 seconds.
```

SciNet compute • calcul CANADA

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:  4.99065e+19 (vs 3.33333e+20) for
n=10000000
Tock registered 0.1636 seconds.
```

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

SciNet compute • calcul CANADA

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

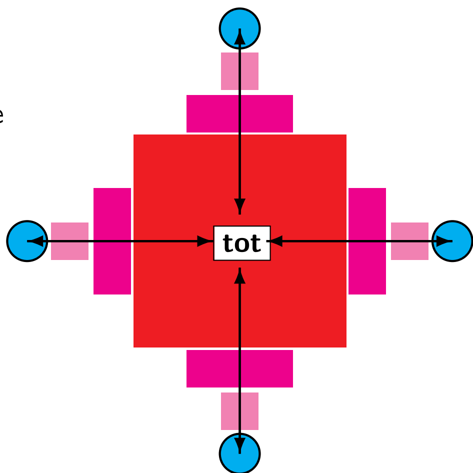
Thread 0: add 1	Thread 1: add 2
read `tot`(=0) into register	
reg = reg+1	read `tot`(=0) into register
store reg(=1) into `tot`	reg=reg+2
	store reg(=2) into `tot`

`tot` = 2

SciNet compute • calcul CANADA

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.



SciNet compute • calcul CANADA

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:  3.33333e+20
(vs 3.33333e+20) for n=10000000
Tock registers 1.243 seconds.
```

Correct, but 23x slower than serial version!

SciNet compute • calcul CANADA

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:  3.33333e+20
(vs 3.33333e+20) for n=10000000
Tock registers 0.6732 seconds.
```

Correct, and better – only 13x slower than serial.

SciNet compute • calcul CANADA

How should we fix the slowdown?

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

$$
\begin{aligned}
n &= \vec{x} \cdot \vec{y} \\
&= \sum_i x_i \, y_i \\
&= \sum_p \left(\sum_i x_i \, y_i \right)
\end{aligned}
$$

SciNet
compute • calcul
CANADA

Local variables

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

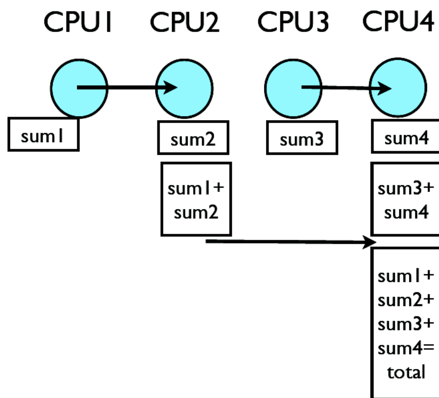
```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product:  3.3333e+20
(vs 3.3333e+20) for n=10000000
Tock registered 0.01201 seconds.
```

Now we're talking! 4x faster than serial.

SciNet
compute • calcul
CANADA

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: `+ * min max` ...
- omp_ndot_reduction.cc



SciNet compute • calcul CANADA

OpenMP Reduction Operations

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

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_reduction
Dot product:  3.33333e+20
(vs 3.33333e+20) for n=10000000
Tock registered 0.01162 seconds.
```

About the same speed as local variables, simpler code!

SciNet compute • calcul CANADA

Performance

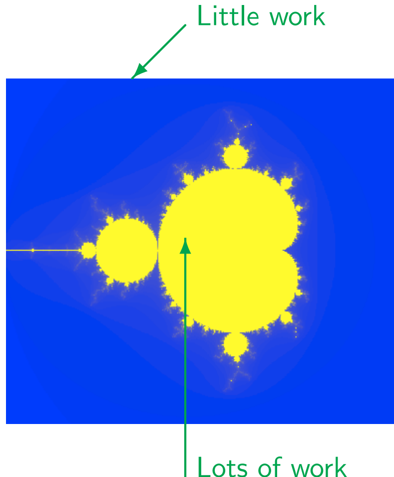
- We threw in 8 cores, got a factor of 4 speedup. Why?
- Often we are limited not by CPU power but by how quickly we can feed CPUs.
- For this problem, we had $\mathbf{10^7}$ long vectors, with 2 numbers 8 bytes long flowing through in 0.012 seconds.
- Combined bandwidth from main memory was 13 GB/s. Not far off of what we could hope for on the GPC.
- 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.

Load Balancing in OpenMP

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

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

Example - Mandelbrot Set



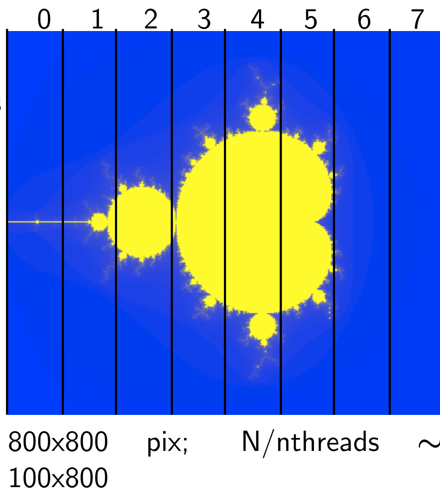
- Mandelbrot set simple example of non-balanced problem.
- Defined as complex points $\mathbf{a}$ where $|\mathbf{b}_\infty|$ finite, with $\mathbf{b_0 = 0}$ and $\mathbf{b_{n+1} = b_n^2 + a}$. If $|\mathbf{b_n}| > \mathbf{2}$, point diverges.
- Calculation:
  - pick some **nmax**
  - iterate for each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



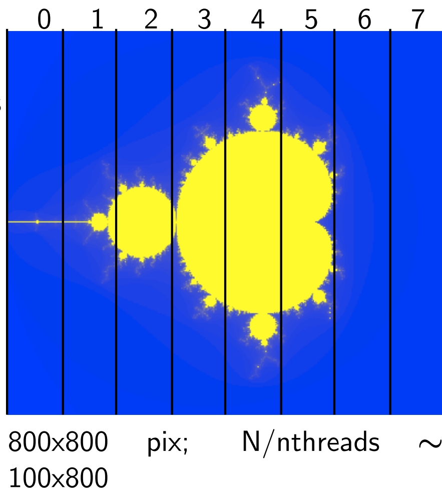
800x800 pix; N/nthreads $\sim$ 100x800

SciNet compute • calcul CANADA

First OpenMP Mandelbrot Set

- Default work sharing breaks **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 $\sim$ 100x800

SciNet compute • calcul CANADA

Scheduling constructs in OpenMP

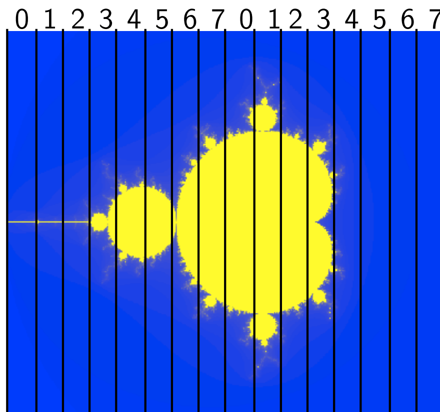
- Default: each thread gets a big consecutive chunk of the loop. Often better to give each thread many smaller interleaved chunks.
- Can add `schedule` clause to `omp for` to change work sharing.
- We can decide either at compile-time (static schedule) or run-time (dynamic schedule) how work will be split.
- `#pragma omp for schedule(static, m)` gives m consecutive loop elements to each thread instead of a big chunk.
- With `schedule(dynamic, m)`, each thread will work through m loop elements, then go to the OpenMP run-time system and ask for more.
- Load balancing (possibly) better with dynamic, but larger overhead than with static.

SciNet compute • calcul CANADA

Second Try OpenMP Mandelbrot Set

```
#pragma omp for schedule(static,50)
```

- Can change the chunk size different from ∼ **N/nthreads**
- In this case, more columns – work distributed a bit better.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:



800x800 pix; each threads: 50x800

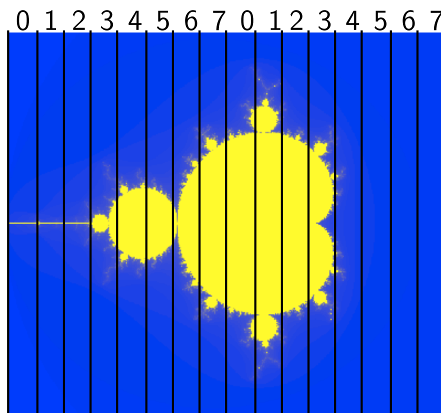
SciNet compute • calcul CANADA

Second Try OpenMP Mandelbrot Set

```
#pragma omp for schedule(static,50)
```

- Can change the chunk size different from ∼ **N/nthreads**
- In this case, more columns – work distributed a bit better.
- 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%



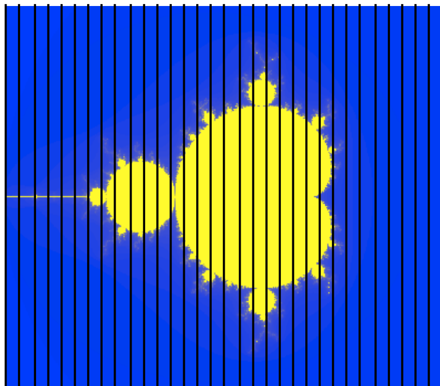
800x800 pix; each threads: 50x800

SciNet
compute • calcul
CANADA

Third Try: Schedule dynamic

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

- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.



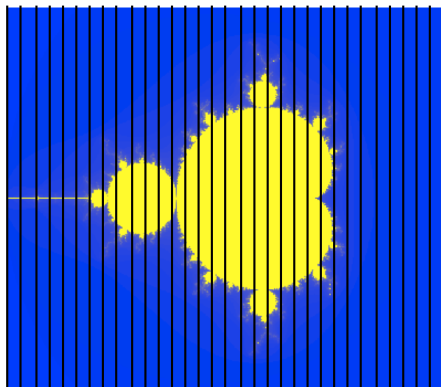
SciNet
compute • calcul
CANADA

Third Try: Schedule dynamic

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

- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

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



SciNet
compute • calcul CANADA

Tuning

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

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

SciNet compute • calcul CANADA

Two level loops

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

Almost without exception, want the loop 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=((double)i)/((double)npix);
      double  y=((double)j)/((double)npix);
      std::complex<double> a(x,y);
      mymap[i][j]=how_many_iter_real(a,maxiter);
   }
```

SciNet compute • calcul CANADA

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.

SciNet compute • calcul CANADA

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.

SciNet
compute • calcul
CANADA

Conditional OpenMP in Action

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

SciNet compute • calcul CANADA

Controlling # of Threads

- Sometimes you might want more or fewer threads. May even want to change while running.
- 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 <iostream>
#include <omp.h>
int main(int argc,char *argv[]){
   //find # of physical cores
   //this is an openmp library routine.
   int max_threads=omp_get_num_procs();
   int n=atoi(argv[1]);
   //set # threads equal to input
   //assuming it's less than max_threads if
   (n<max_threads)
      omp_set_num_threads(n);
   else
      omp_set_num_threads(max_threads);
   #pragma omp parallel
   #pragma omp single
   std::cout << "Running with " <<
   omp_get_num_threads() << " threads for n=" << n
   << ".\n";
}
```

SciNet
compute • calcul
CANADA

Non-loop construct

OpenMP supports non-loop parallelism as well:

- Sections:

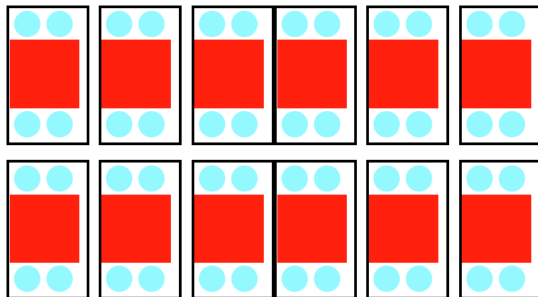
```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
         something to do
      }
      #pragma omp section
      {
         something to do
         at the same time
      }
   }
}
```

- More flexible: tasks

SciNet compute • calcul CANADA

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.cc

```
#include <iostream>
#include <omp.h>
int main(){
   #pragma omp parallel
   #pragma omp single
   {
      std::cout << "hello";
      #pragma omp task
      {
         std::cout << "hello 1 from " <<
         omp_get_thread_num() << ".";
      }
      #pragma omp task
      std::cout << "hello 2 from " <<
      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

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.
- Works for a variety of non-array structure (linked lists, etc.)

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

Parallel traversal starts thusly:

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

Beauty of Tasks #2

Linked list:

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

SciNet compute • calcul CANADA

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: Homework!

Homework 10