Before we start with OpenMP...

```
$ ssh -X login.scinet.utoronto.ca
$ ssh -X gpc01
$ qsub -X -I -l nodes=1:ppn=8,walltime=5:00:00,os=centos53develibA
```

to get a dedicated development node (ensure this works).

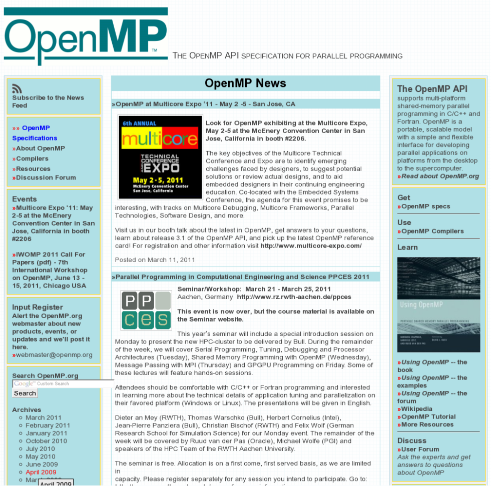
```
$ cp -r ~ljdursi/ppp ~/
$ source ~/ppp/setup
$ cd ~/ppp
$ cd util
$ make
$ cd ~/ppp/omp-intro
$ make mandel
$ mandel
```

Part V

Introduction to OpenMP

OpenMP

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



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events
»Multicore Expo '11: May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206

»IWOMP 2011 Call For Papers (pdf) - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Search

Archives
- March 2011
- February 2011
- January 2011
- October 2010
- July 2010
- May 2010
- June 2009
- April 2009
- Mar

OpenMP News

»OpenMP at Multicore Expo '11 - May 2 -5 - San Jose, CA

Look for OpenMP exhibiting at the Multicore Expo, May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206.

The key objectives of the Multicore Technical Conference and Expo are to identify emerging challenges faced by designers, to suggest potential solutions or review actual designs, and to aid embedded designers in their continuing engineering education. Co-located with the Embedded Systems Conference, the agenda for this event promises to be interesting, with tracks on Multicore Debugging, Multicore Frameworks, Parallel Technologies, Software Design, and more.

Visit us in our booth talk about the latest in OpenMP, get answers to your questions, learn about release 3.1 of the OpenMP API, and pick up the latest OpenMP reference card! For registration and other information visit **http://www.multicore-expo.com/**

Posted on March 11, 2011

»Parallel Programming in Computational Engineering and Science PPCES 2011

Seminar/Workshop: March 21 - March 25, 2011
Aachen, Germany **http://www.rz.rwth-aachen.de/ppces**

This event is now over, but the course material is available on the Seminar website.

This year's seminar will include a special introduction session on Monday to present the new HPC-cluster to be delivered by Bull. During the remainder of the week, we will cover Serial Programming, Tuning, Debugging and Processor Architectures (Tuesday), Shared Memory Programming with OpenMP (Wednesday), Message Passing with MPI (Thursday) and GPGPU Programming on Friday. Some of these lectures will feature hands-on sessions.

Attendees should be comfortable with C/C++ or Fortran programming and interested in learning more about the technical details of application tuning and parallelization on their favored platform (Windows or Linux). The presentations will be given in English.

Dieter an Mey (RWTH), Thomas Warschko (Bull), Herbert Cornelius (Intel), Jean-Pierre Panziera (Bull), Christian Bischof (RWTH) and Felix Wolf (German Research School for Simulation Science) for our Monday event. The remainder of the week will be covered by Ruud van der Pas (Oracle), Michael Wolfe (PGI) and speakers of the HPC Team of the RWTH Aachen University.

The seminar is free. Allocation is on a first come, first served basis, as we are limited in
capacity. Please register separately for any session you intend to participate. Go to:

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»*Read about OpenMP.org*

Get
»OpenMP specs

Use
»OpenMP Compilers

Learn

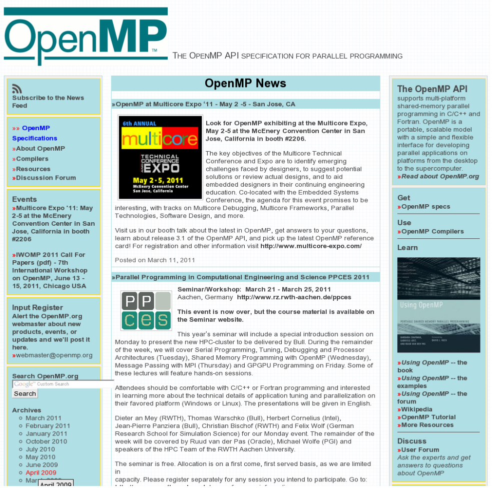
»*Using OpenMP* -- the book
»*Using OpenMP* -- the examples
»*Using OpenMP* -- the forum
»Wikipedia
»OpenMP Tutorial
»More Resources

Discuss
»User Forum
Ask the experts and get answers to questions about OpenMP

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.



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events
»Multicore Expo '11: May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206

»IWOMP 2011 Call For Papers (pdf) - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Search

Archives
- March 2011
- February 2011
- January 2011
- October 2010
- July 2010
- May 2010
- June 2009
- April 2009
- Mar

OpenMP News

»OpenMP at Multicore Expo '11 - May 2 -5 - San Jose, CA

Look for OpenMP exhibiting at the Multicore Expo, May 2-5 at the McEnery Convention Center in San Jose, California in booth #2206.

The key objectives of the Multicore Technical Conference and Expo are to identify emerging challenges faced by designers, to suggest potential solutions or review actual designs, and to aid embedded designers in their continuing engineering education. Co-located with the Embedded Systems Conference, the agenda for this event promises to be interesting, with tracks on Multicore Debugging, Multicore Frameworks, Parallel Technologies, Software Design, and more.

Visit us in our booth talk about the latest in OpenMP, get answers to your questions, learn about release 3.1 of the OpenMP API, and pick up the latest OpenMP reference card! For registration and other information visit **http://www.multicore-expo.com/**

Posted on March 11, 2011

»Parallel Programming in Computational Engineering and Science PPCES 2011

Seminar/Workshop: March 21 - March 25, 2011
Aachen, Germany **http://www.rz.rwth-aachen.de/ppces**

This event is now over, but the course material is available on the Seminar website.

This year's seminar will include a special introduction session on Monday to present the new HPC-cluster to be delivered by Bull. During the remainder of the week, we will cover Serial Programming, Tuning, Debugging and Processor Architectures (Tuesday), Shared Memory Programming with OpenMP (Wednesday), Message Passing with MPI (Thursday) and GPGPU Programming on Friday. Some of these lectures will feature hands-on sessions.

Attendees should be comfortable with C/C++ or Fortran programming and interested in learning more about the technical details of application tuning and parallelization on their favored platform (Windows or Linux). The presentations will be given in English.

Dieter an Mey (RWTH), Thomas Warschko (Bull), Herbert Cornelius (Intel), Jean-Pierre Panziera (Bull), Christian Bischof (RWTH) and Felix Wolf (German Research School for Simulation Science) for our Monday event. The remainder of the week will be covered by Ruud van der Pas (Oracle), Michael Wolfe (PGI) and speakers of the HPC Team of the RWTH Aachen University.

The seminar is free. Allocation is on a first come, first served basis, as we are limited in
capacity. Please register separately for any session you intend to participate. Go to:

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»*Read about OpenMP.org*

Get
»OpenMP specs

Use
»OpenMP Compilers

Learn

»*Using OpenMP* -- the book
»*Using OpenMP* -- the examples
»*Using OpenMP* -- the forum
»Wikipedia
»OpenMP Tutorial
»More Resources

Discuss
»User Forum
Ask the experts and get answers to questions about OpenMP

OpenMP basic operations

In code:

- In C, you add lines starting with `#pragma omp`.
  This parallelizes the subsequent code block.
- In Fortran, you add lines starting with `!$omp`.
  An `!$omp end ...` is needed to close the parallel region.
- These lines are skipped (for C, sometimes with a warning) by compilers that do not support OpenMP.

When compiling:

- To turn on OpenMP support in gcc and gfortran, 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.

OpenMP example

C:

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

F90:

```fortran
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

OpenMP example

```
$ gcc -std=c99 -O2 -o omp-hello-world omp-hello-world.c -fopenmp
or
$ gfortran -O2 -o omp-hello-world omp-hello-world.f90 -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. . .

OpenMP example

```
$ gcc -o omp-hello-world omp-hello-world.c -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!
```

So what happened precisely?

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

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

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

So what happened precisely?

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

Program starts normally (single thread)

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

So what happened precisely?

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

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

```
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

So what happened precisely?

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

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

```fortran
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

So what happened precisely?

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

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

```fortran
program omp_hello_world
use omp_lib
implicit none
print *, 'At start of program'
!$omp parallel
  print *, 'Hello world from thread ', omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

OpenMP functions (from omp.h/omp_lib)

```
#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());
  }
}
```

omp_get_num_threads() called by all threads.

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

OpenMP functions (from omp.h/omp_lib)

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

What do you think, will this work?

OpenMP functions (from omp.h/omp_lib)

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

What do you think, will this work?

No:

Says 1 thread only!

OpenMP functions (from omp.h/omp_lib)

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

What do you think, will this work?

No:

Says 1 thread only!

Why?

Because that is true outside the parallel region!

Need to get the value from the parallel region somehow.

Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

Variables in OpenMP

Variables in parallel regions are a bit tricky.

Variable declarations
How used in parallel region

```c
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

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.)

Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

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

Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

- 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)

Variables in OpenMP

Variables in parallel regions are a bit tricky.

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

- 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)

Variables in OpenMP - Fortran version

```
program omp_vars
use omp_lib
implicit none
integer ::  mythread, nthreads
!$omp parallel default(none) private(mythread) shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel
print *,'Number of threads was ', nthreads, '.'
end program omp_vars
```

Single Execution in OpenMP

```
#include <stdio.h>
#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();
  }
  printf("There were %d threads.\n", nthreads);
}
```

- 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.

Single Execution in OpenMP

```
#include <stdio.h>
#include <omp.h>
int main() {
  int nthreads;
  #pragma omp parallel default(none) shared(nthreads)
  #pragma omp single
    nthreads = omp_get_num_threads();
  printf("There were %d threads.\n", nthreads);
}
```

```
program omp_vars
use omp_lib
implicit none
integer ::  nthreads
!$omp parallel default(none) shared(nthreads)
!$omp single
  nthreads = omp_get_num_threads()
!$omp end single
!$omp end parallel
print *,'Number of threads was ', nthreads, '.'
end program omp_vars
```

Loops in OpenMP

Take one of your openmp programs and add a loop.

Loops in OpenMP

Take one of your openmp programs and add a loop.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none) \
   XXXX(i) XXXX(mythread)
  {
    mythread = omp_get_thread_num();
    for (i=0; i<16; i++)
      printf("Thread %d gets i=%d\n",
              mythread, i);
  }
}
```

```fortran
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) &
!$omp XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
             ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

Loops in OpenMP

Take one of your openmp programs and add a loop.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none) \
   XXXX(i) XXXX(mythread)
  {
    mythread = omp_get_thread_num();
    for (i=0; i<16; i++)
      printf("Thread %d gets i=%d\n",
              mythread, i);
  }
}
```

```fortran
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) &
!$omp XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
            ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

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

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 (or omp do).

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 (or omp do).

```
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none) XXXX(i) XXXX(mythread)
  {
    mythread = omp_get_thread_num();
    #pragma omp for
    for (i=0; i<16; i++)
      printf("Thread %d gets i=%d\n",mythread,i);
  }
}
```

Fortran version

```
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  !$omp do
  do i=1,16
    print *, 'thread ', mythread, ' gets i=', i
  enddo
  !$omp end do
!$omp end parallel
end program omp_loop
```

Worksharing constructs in OpenMP

- omp for/omp do 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)
- Implement this, first serially, then with OpenMP
- daxpy.c or daxpy.f90
- make daxpy or make fdaxpy

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

Warning

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

```c
#include <stdio.h>
#include "pca_utils.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=1e7;
 double *x = vector(n);
 double *y = vector(n);
 double *z = vector(n);
 double a = 5./3.;
 pca_time tt;
 tick(&tt);
 daxpy(n,a,x,y,z);
 tock(&tt);
 free(z);
 free(y);
 free(x);
}
```

```
#include <stdio.h>
#include "pca_utils.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=1e7;
  double *x = vector(n);
  double *y = vector(n);
  double *z = vector(n);
  double a = 5./3.;
  pca_time tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z);
  free(y);
  free(x);
}
```

Utilities for this course; `double` is a numerical type which can be set to single or double precision

```c
#include <stdio.h>
#include "pca_utils.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=1e7;
  double *x = vector(n);
  double *y = vector(n);
  double *z = vector(n);
  double a = 5./3.;
  pca_time tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z);
  free(y);
  free(x);
}
```

Fill arrays with calculated values.

```
#include <stdio.h>
#include "pca_utils.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=1e7;
  double *x = vector(n);
  double *y = vector(n);
  double *z = vector(n);
  double a = 5./3.;
  pca_time tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z);
  free(y);
  free(x);
}
```

Do calculation.

```
#include <stdio.h>
#include "pca_utils.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=1e7;
  double *x = vector(n);
  double *y = vector(n);
  double *z = vector(n);
  double a = 5./3.;
  pca_time tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z);
  free(y);
  free(x);
}
```

$\longleftarrow$ Driver (setup, call, timing).

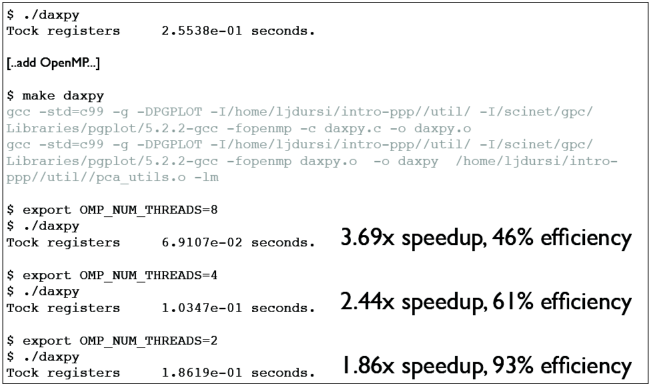
```
#include <stdio.h>
#include "pca_utils.h"
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*(double)i;
    y[i] = ((double)i+1.)*((double)i-1.);
  }
  for (int i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}
int main() {
  int n=1e7;
  double *x = vector(n);
  double *y = vector(n);
  double *z = vector(n);
  double a = 5./3.;
  pca_time tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z);
  free(y);
  free(x);
}
```

HANDS-ON: Try OpenMPing. . .

HANDS-ON 1:
Parallelize daxpy with OpenMP.
Also do the scaling analysis!

```c
void daxpy(int n, double a, double *x, double *y, double *z) {
  #pragma omp parallel default(none) shared(n,x,y,a,z) private(i)
  {
    #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];
  }
}
```

```fortran
!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$omp do
do i=1,n
  x(i) = (i)*(i)
  y(i) = (i+1.)*(i-1.)
enddo
!$omp do
do i=1,n
  z(i) = a*x(i) + y(i)
enddo
!$omp end parallel
```



```
$ ./daxpy
Tock registers      2.5538e-01 seconds.

[..add OpenMP...]

$ make daxpy
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp -c daxpy.c -o daxpy.o
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp daxpy.o  -o daxpy  /home/ljdursi/intro-
ppp//util//pca_utils.o -lm

$ export OMP_NUM_THREADS=8
$ ./daxpy
Tock registers      6.9107e-02 seconds.   3.69x speedup, 46% efficiency

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers      1.0347e-01 seconds.   2.44x speedup, 61% efficiency

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers      1.8619e-01 seconds.   1.86x speedup, 93% efficiency
```

```c
void daxpy(int n, double a, double *x, double *y, double *z) {
  #pragma omp parallel default(none) shared(n,x,y,a,z) private(i)
  {
    #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];
  }
}
```

Why is this safe?
Everyone is modifying x,y,z!

```fortran
!$omp parallel default(none) private(i) shared(n,a,x,y,z)
!$omp do
do i=1,n
  x(i) = (i)*(i)
  y(i) = (i+1.)*(i-1.)
enddo
!$omp do
do i=1,n
  z(i) = a*x(i) + y(i)
enddo
!$omp end parallel
```

Dot Product

- Dot product of two vectors
- Implement this, first serially, then with OpenMP
- ndot.c or ndot.f90
- make ndot or make ndotf
- Tells time, answer, correct answer.

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

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

Dot Product - serial

```c
#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

```c
#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;
}
```

```
$ 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.

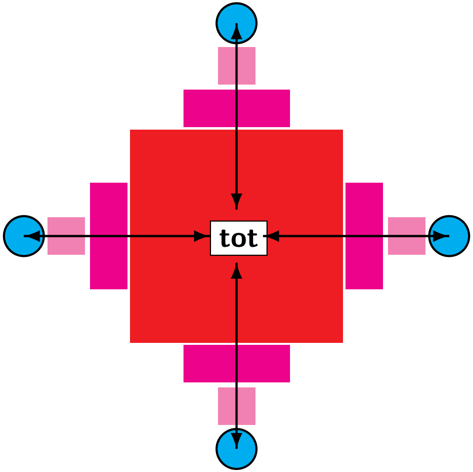
`tot` = 0

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`
- `!$omp critical`
  `!$omp end 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`
- `!$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 ($10^7/P$ additions).
- And then sums to `tot` (only P additions with critical or atomic. . .
- HANDS-ON: Try it!

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

HANDS-ON 2:
Parallelize ndot with partial sums.
copy one of the omp_ndot.c's (or fomp_ndot.c's) to omp_ndot_local.c (or fomp_ndot_local.f90).

Local variables

```
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;
}
```

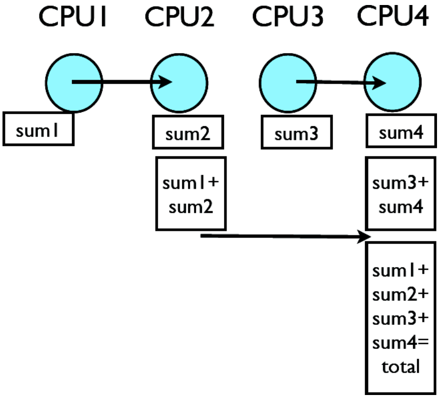
```
ndot = 0.
!$omp parallel shared(x,y,n,ndot) &
!$omp private(i,mytot)
mytot = 0.
!$omp do
do i=1,n
  mytot = mytot + x(i)*y(i)
enddo
!$omp atomic
ndot = ndot + mytot
!$omp end parallel
```

```
$ 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, fomp_ndot_reduction.f90



OpenMP Reduction Operations

```
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];
}
```

```
ndot = 0.
!$omp parallel shared(x,y,n) &
!$omp private(i) reduction(+:ndot)
!$omp do
do i=1,n
  ndot = ndot + x(i)*y(i)
enddo
!$omp end parallel
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is 3.3333e+20
(vs 3.3333e+20) for n=10000000.
Took 1.8134-02 seconds.
```

Same speed, simpler code!

OpenMP Reduction Operations

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

```
ndot = 0.
!$omp parallel do shared(x,y,n) &
!$omp private(i) reduction(+:ndot)
do i=1,n
  ndot = ndot + x(i)*y(i)
enddo
!$omp end parallel
```

```
$ 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!

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 107 long vectors, with 2 numbers 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.

Load Balancing in OpenMP

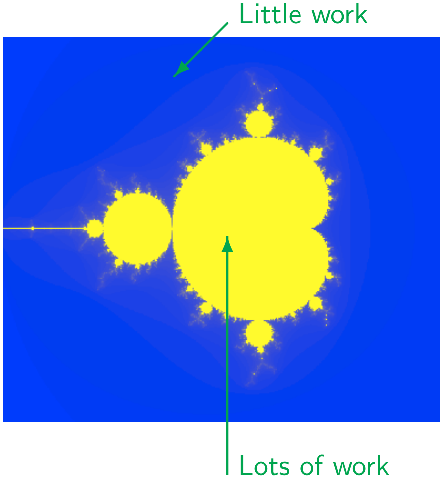
- So far every iteration of the loop had the same amount of work.
- Not always the case
- Sometimes cannot predict beforehand how unbalanced the problem is

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

Example - Mandelbrot Set

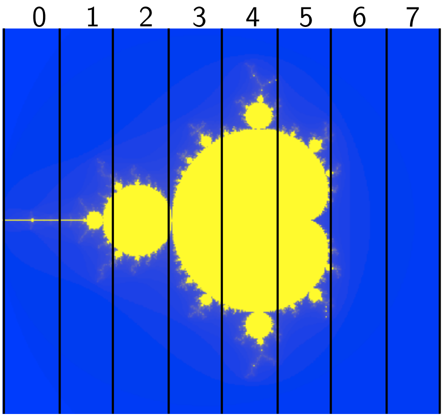
- Mandelbrot set simple example of non-balanced problem.
- Defined as complex points $\mathbf{a}$ where $|\mathbf{b}_\infty|$ finite, with $\mathbf{b_0} = \mathbf{0}$ and $\mathbf{b_{n+1}} = \mathbf{b_n^2} + \mathbf{a}$.
  If $|\mathbf{b_n}| > \mathbf{2}$, point diverges.
- Calculation:
  - pick some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



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.

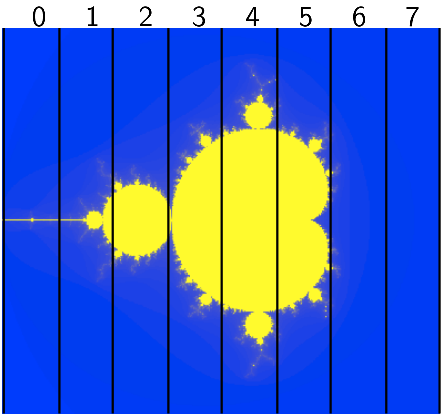


800x800 pix; N/nthreads $\sim$ 100x800

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

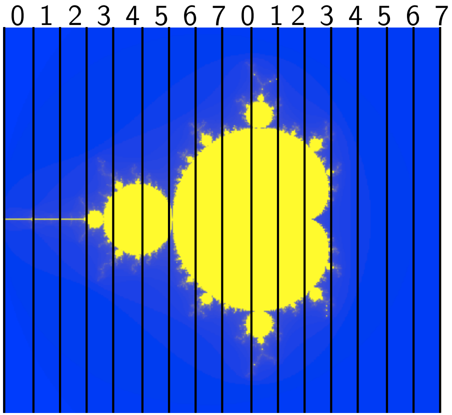
Second Try OpenMP Mandelbrot Set

- 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:

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

or

```
!$omp do schedule(static,50)
```



800x800 pix; each threads: 50x800

Second Try OpenMP Mandelbrot Set

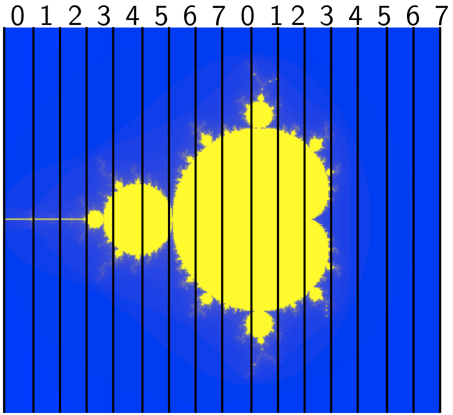
- Can change the chunk size different from ∼ **N/nthreads**
- In this case, more columns – work distributed a bit better.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:

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

or

```
!$omp do schedule(static,50)
```

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



800x800 pix; each threads: 50x800

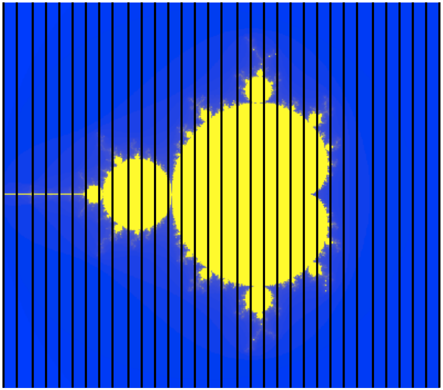
Third Try: Schedule dynamic

- 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.

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

or

```
!$omp do schedule(dynamic)
```



Third Try: Schedule dynamic

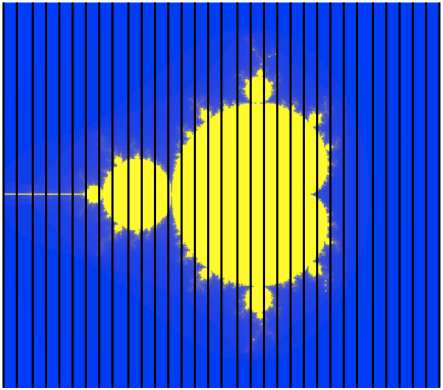
- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

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

or

```
!$omp do schedule(dynamic)
```

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 chuck 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 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);
    double complex a=x+I*y;
    mymap[i][j]=how_many_iter_real(a,maxite
  }
```

Summary so far

- Start a parallel region:
  #pragma omp parallel shared() private() default()
- Parallelize a loop:
  #pragma omp for schedule(static/dynamic, chunk)
- Mark off a region only one thread can be in at a time:
  #pragma omp critical
- Safely update a single memory location:
  #pragma omp atomic
- In a parallel region, have only one process do something:
  #pragma omp single

See: http://openmp.org/wp/openmp-specifications/ for more info.
Strongly encouraged - many good sample programs.

Summary So Far II

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.

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? lastprivate():

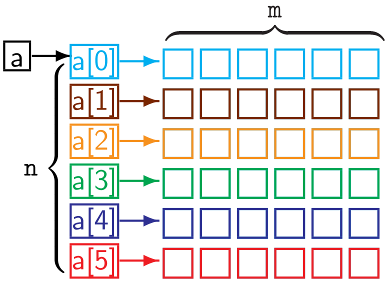
```
#include <stdio.h>
int main() {
  int n;
  #pragma omp parallel for lastprivate(n)
  for (int i=0;i<100;i++)
    if (i>70) n=i;
  printf("Last n was %\",n);
}
```

Memory Access — a seemingly unrelated intermezzo

Memory Access

- Processors work on local bits of memory in their cache.
- Cache is small and fast. Main memory is big, but slow.
- There is a large latency in getting things from main memory — often hundreds of clock cycles. The fewer times we access main memory, the faster we will go.
- Computers bring in chunks of memory at a time. If you access data in contiguous memory chunks, much of it may already be in cache. Always try to do this - serial or parallel.
- C - last index is rapidly varying. Fortran first index.

Memory layout



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.

Example - Matrix Multiplication

Example

- Linear algebra a classic example.
- Matrix multiplication: $\mathbf{C} = \mathbf{A} * \mathbf{B}$, or $\mathbf{c[i][j]} = \sum \mathbf{a[i][k]} * \mathbf{b[k][j]}$.
- Different implementations can take 10-100x longer than optimal. Slowness entirely due to memory access.
- The more you do with stuff youve pulled from main memory, the faster youll run.

Slow Matrix Multiplication

```
void matmult_slowest(double **a,
double **b, double **c, int n) {
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]=a[i][k]*b[k][j];
    }
}
```

```
int main() {
  pca_time tt;
  int n=500;
  double **a=matrix(n,n);
  double **b=matrix(n,n);
  double **c=matrix(n,n);
  fill_random_matrix(a,n);
  tick(&tt);
  matmult_slowest(a,b,c,n);
  printf("Time to multiply %dx%d ma-
  trices with slow multiplication is
  %f\",n,n,tocksilet(&tt));
  printf("Sum of elements is
  %e\n",matrix_sum(c,n));
}
```

```
$ ./matmult_slow
Time to multiply 1000 x 1000 matrices with slow multiplication is 12.4637
Sum of elements is 2.4997e+08
$
```

Slow Matrix Multiplication

- What happened? For every element in C, we had to pull a fast direction from A, but a slow direction from B.
- Could change the order of the loops, making B fast, but then A would be slow.
- We pulled a slow vector for each element in C, for a total of n2 slow column pulls.
- Could make the transpose of B, then we would always pull from the fast columns. Only have to do n slow pulls this way.
- Drawback: must make a copy of B. If B is large, can take lots of memory.

Transpose Multiplication

```
void matmult_transport(double **x, double **b, double **c, int n) {
  double **bt=matrix(n,n);
  matrix_transpose(b,bt,n);
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++) {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]=a[i][k]*bt[j][k];
    }
}
```

```
$ ./matmul_transporse
Time to multiply 1000 x 1000 matrices with transpose multiplication is 8.8756
Sum of elements is 2.4997e+08
$
```

About 40% faster than slow version.

Blocks

- Multiplication was still kind of slow. Why?
- For every column of C we calculate, we have to process all of B, for a total of n times. That's a lot of memory throughput.
- Recall $\mathbf{c_{ij} = a_{ik} * b_{kj}}$. Nowhere have we said that $\mathbf{c_{ij}}$, $\mathbf{a_{jk}}$, and $\mathbf{b_{kj}}$ are scalars. They could be blocks of the matrices. If we treat them as blocks, then we'll have to go to main memory less often.
- Say blocks are 20x20. Then I have to pull all of B each time I process a column of blocks. Or a total of n/20 times. Much less stress on system memory.

```
void matmult_block(double **a, double
**b, double **c, int n, int bs) {
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      c[i][j]=0;
  int nb=n/bs;
  assert(nb*bs==n);
  double**myblock_1=matrix(bs,bs);
  double**myblock_2=matrix(bs,bs);
  double**myblock_3=matrix(bs,bs);
  for (int ib=0;ib<nb;ib++)
    for (int jb=0;jb<nb;jb++)
      for (int kb=0;kb<nb;kb++) {
        int ii=ib*bs;
        int jj=jb*bs;
        int kk=kb*bs;
        // Pull blocks from A and B out
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_1[i][j]=a[i+ii][j+kk];
            myblock_2[i][j]=b[j+kk][i+jj];
          }
        // Do the block multiplication
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_3[i][j]=0;
            for (int k=0;k<bs;k++)
              myblock_3[i][j]+=
                myblock_1[i][k]
                *myblock_2[j][k];
          }
        //Accumulate the product into c
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++)
            c[i+ii][j+jj]+=myblock_3[i][j];
      }
  free_matrix(myblock_1);
  free_matrix(myblock_2);
  free_matrix(myblock_3);
}
```

```
$ ./matmul_block
Time to multiply 1000 x 1000 matrices
with block multiplication is 9.5774
Sum of elements is 2.4997e+08
$
```

Blocks Debrief

- Well, managed to do better in memory, calculation time was similar (slightly larger actually).
- You may gather that writing a fast, parallel matrix multiplier isn't easy. You are right.
- People have spent a long time optimizing matrix multiplication, and gotten to 80-90% of theoretical max, using block-based algorithms (look up goto blas)
- Important corollary: Think you need to code something? Don't! See if someone else has done it. For core routines, they have, and better than you will ever do it.
- For the same problem, Goto runs in 0.1972 – 50x faster.
  (module load gotoblas)

Big Lesson #5

Make sure serial performance is good before worrying about parallel!

End of intermezzo on memory access

HANDS-ON 3:
OpenMP-Parallelize Matrix Multiplications.
Consider both the blocked and the non-blocked versions.

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 - IBM P6 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.

omp_set_num_threads() in action

```
#include "stdio.h"
#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
  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 physical number of available processors.

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 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?).

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

How would you do this problem without tasks?

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node*);
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);
```