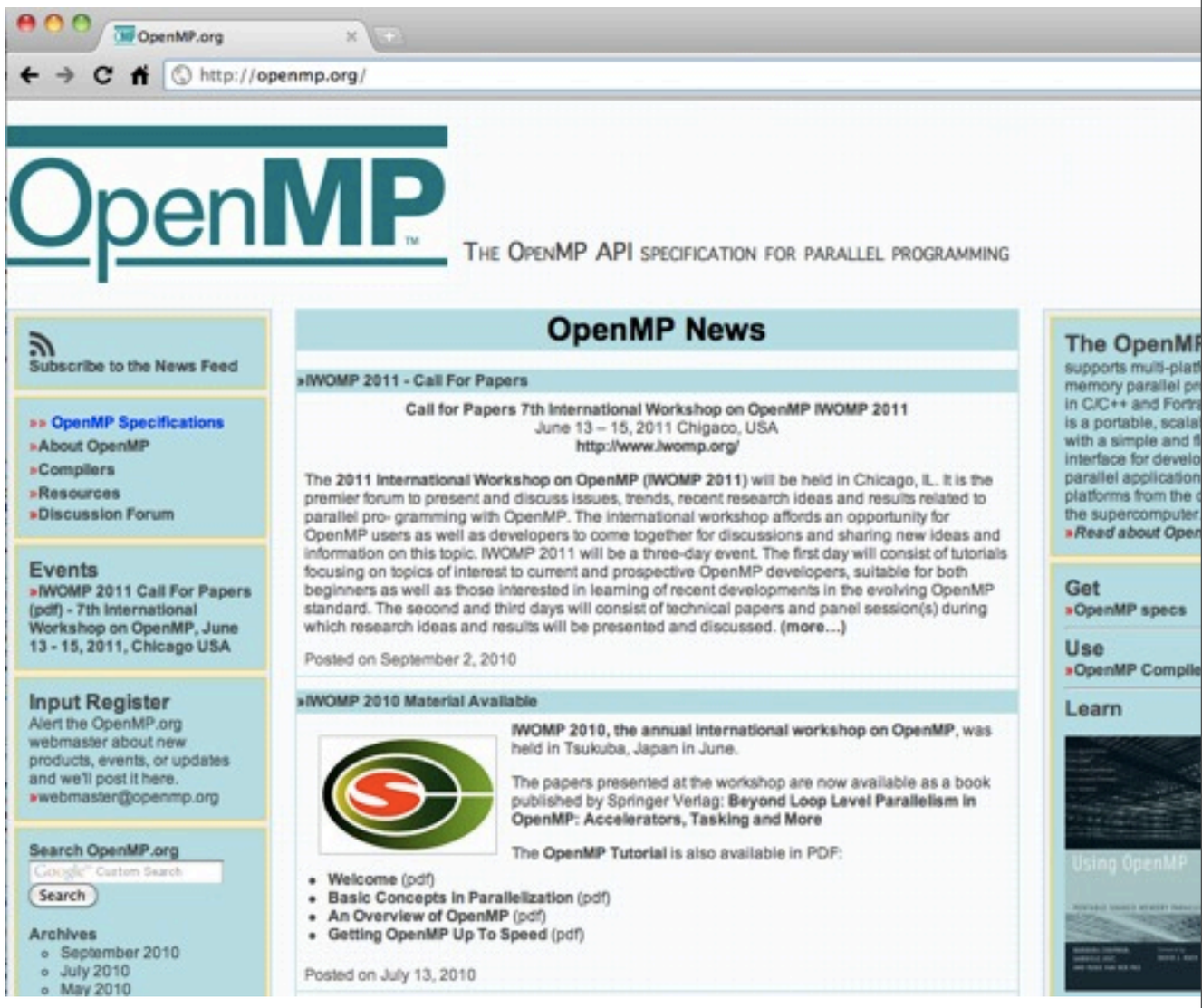
For those with laptops

- Log into SciNet GPC devel node (ssh -X or ssh -Y)
- cd
- cp -r /scinet/course/ppp .
- cd ppp
- source setup
- cd omp-intro
- make
- ./mandel

An introduction to OpenMP

OpenMP

- For Shared Memory systems
- Add Parallelism to functioning serial code
- Add compiler directives to code
- http://openmp.org - tonnes of useful info



SciNet

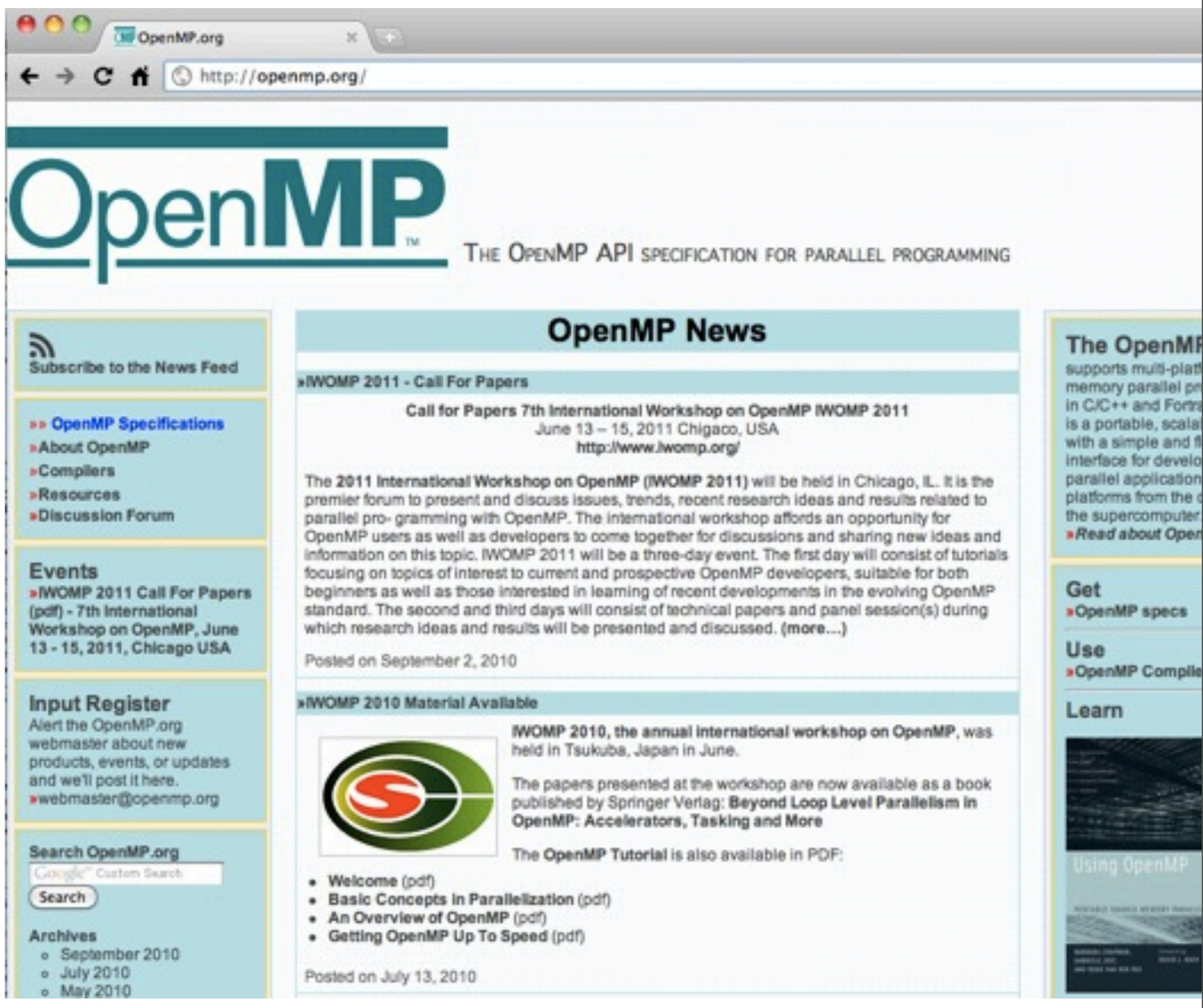
OpenMP

- 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

SciNet

OpenMP

- Mark off parallel regions - in those regions, all available threads do same work
- Markup designed to be invisible to non-OpenMP compilers; should result in working serial code



C: omp-hello-world.c

gcc -fopenmp -o omp-hello-world omp-hello-world.c -lgomp

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 0;
}
```

F90: omp-hello-world.f90

gfortran -fopenmp -o omp-hello-world omp-hello-world.f90 -lgomp

```
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
```

SciNet

```
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
  or
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp

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

```
gpc-f102n084-$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
gpc-f102n084-$ export OMP_NUM_THREADS=8
gpc-f102n084-$ ./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!
gpc-f102n084-$ export OMP_NUM_THREADS=1
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
gpc-f102n084-$ export OMP_NUM_THREADS=32
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
...
```

What did happen?

- OMP_NUM_THREADS threads launched
- Each print "Hello world..."
- In seemingly random order
- Only one 'At start of program'

```
gpc-f102n084-$ gcc -o omp-hello-world omp-hello-world.c
-fopenmp -lgomp
gpc-f102n084-$ export OMP_NUM_THREADS=8
gpc-f102n084-$ ./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!
gpc-f102n084-$ export OMP_NUM_THREADS=1
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
gpc-f102n084-$ export OMP_NUM_THREADS=32
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
...
```

SciNet

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 0;
}
```

Include definitions for OpenMP supporting library (omp_get_thread_num())

```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
```

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 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
```

Program starts normally (Single thread of execution)

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 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
```

At start of parallel section, OMP_NUM_THREADS threads are launched, each execute same code.

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 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
```

At end of parallel section, the threads join back up and back to serial execution

SciNet

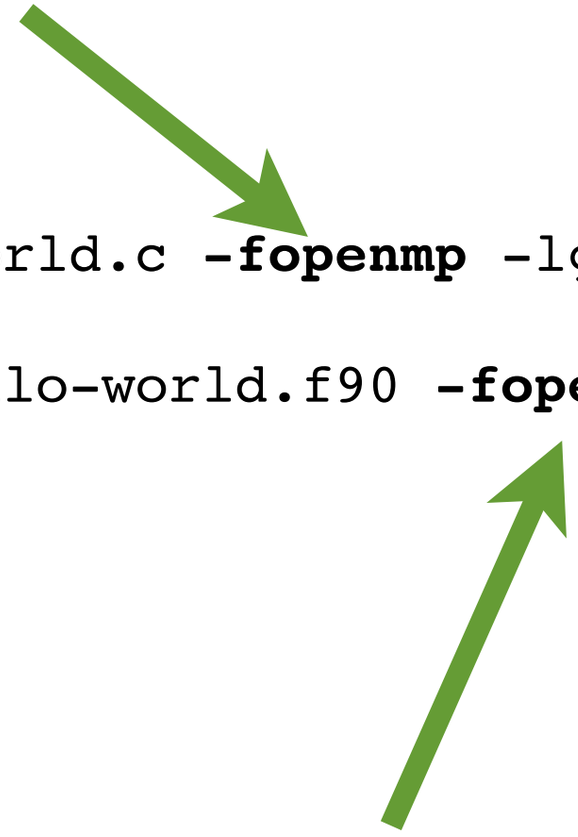
```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
                omp_get_thread_num());
    }
    return 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
```

Special OMP function called to find the thread number of current thread (first = 0)

Turn OpenMP on in compiler (default off; incantation varies from compiler to compiler. Intel: -openmp). Always needed for OpenMP code.

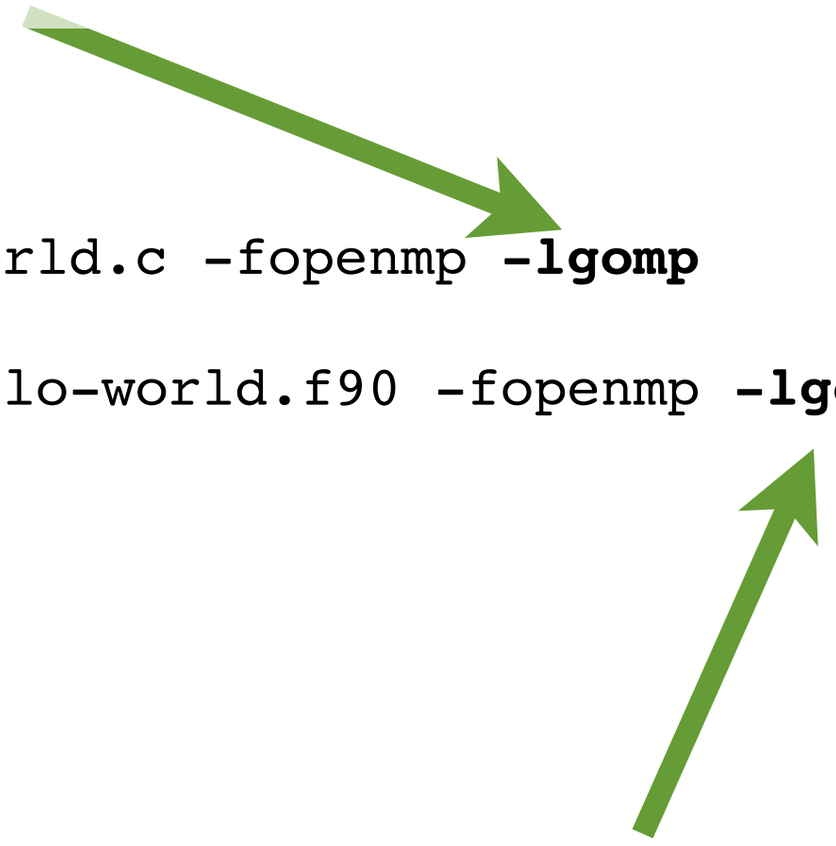
```
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
  or
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp
```



SciNet

Link in OpenMP libraries;
normally only needed if
you use functions like
omp_get_num_threads().
Only at link time.

```
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
  or
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp
```



```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    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());
    }
    return 0;
}
```

(Advanced: can set num_threads (but not thread_num), too.)

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    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() );
    return 0;
}
```

Variables in OpenMP

- Need to put a variable in the parallel section to store the value
- But variables in parallel sections are a little tricky.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    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() );
    return 0;
}
```

C: omp-vars.c
gcc -fopenmp -o omp-vars omp-vars.c -lgomp

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    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("Number of threads was %d.\n",nthreads);
    return 0;
}
```

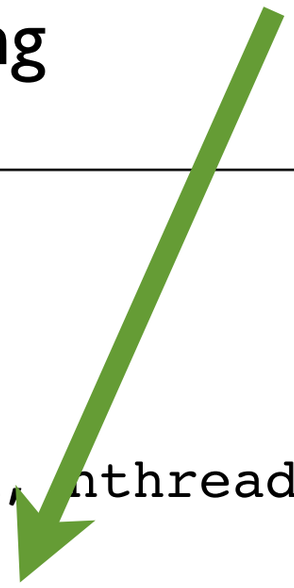
FORTRAN: omp-vars.f90
gfortran -fopenmp -o omp-vars omp-vars.f90 -lgomp

```
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
```

Variable definitions, and
how the are used in the parallel block.

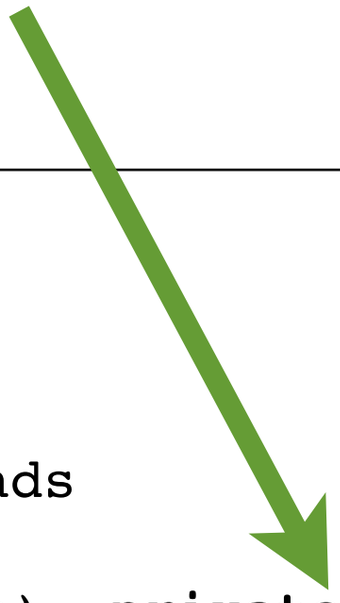
```
program omp_vars
use omp_lib
implicit none

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
```

Strongly, strongly, strongly recommended.
Inconvenient?
30 seconds of extra typing can save you *hours* of debugging



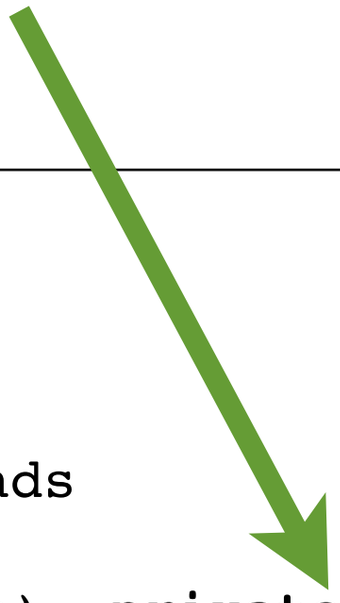
```
program omp_vars
use omp_lib
implicit none

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
```

Each thread gets its own private copy of mythread to do with as it pleases. No other thread can see, modify.



```
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
```

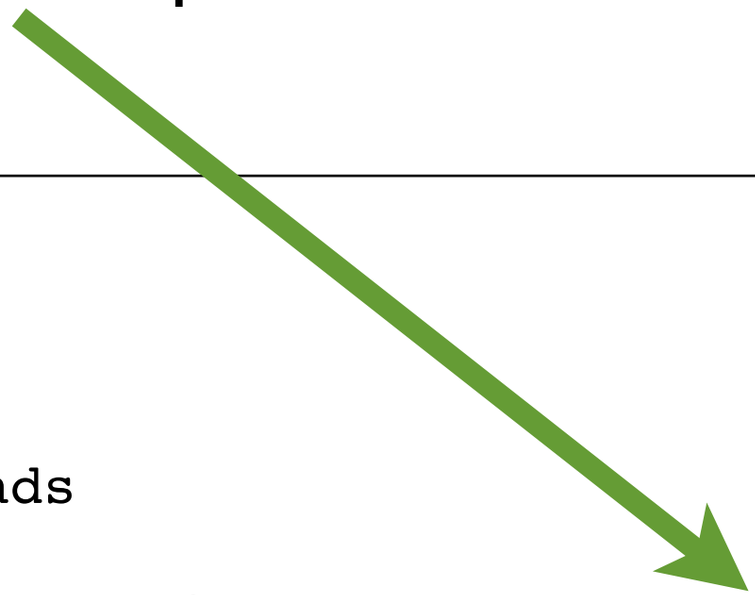
A thread-private variable has *undefined value* inside a parallel block.



```
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
```

(Advanced: firstprivate, lastprivate - copy in/out.)

Everyone can see (ok), modify (danger! danger!) a shared variable. Keeps its value between serial/parallel sections



```
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
```

Variables in OpenMP

- Program runs, launches threads.
- Each thread gets its own copy of mythread
- **Only** thread 0 writes to nthreads
- Outputs number of threads
- What would mythread be if we printed it?

```
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
```

For C folks:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int nthreads;
#pragma omp parallel default(none), shared(nthreads)
    {
        int mythread;
        mythread = omp_get_thread_num();
        if (mythread == 0)
           nthreads = omp_get_num_threads();
    }
    printf("Number of threads was %d.\n",nthreads);
    return 0;
}
```

Local definitions are powerful, and avoid lots of bugs! Variables defined in a parallel block are automatically thread private.

Single Execution in OpenMP

- Do we care that it's thread 0 in particular that updates nthreads?
- Why did we pick 0?
- Often we just want the first thread through to do something, don't care who.

```
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
```

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int nthreads;
#pragma omp parallel default(none), shared(nthreads)
#pragma omp single
        nthreads = omp_get_num_threads();
    printf("Number of threads was %d.\n",nthreads);
    return 0;
}
```

```fortran
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

- Now let's try something a little more interesting

- copy one of your omp programs to omp_loop.c (or omp_loop.f90) and let's put a loop in the parallel section

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    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);
        }
    }
    return 0;
}
```

```fortran
program omp_loop
use omp_lib
implicit none

integer :: i, mythread

!$omp parallel default(none) XXXX(i) XXXX(mythread)
    mythread = omp_get_thread_num()
    do i=1,16
        print *, 'thread ', mythread, ' gets i=', i
    enddo
!$omp end parallel

end program omp_loop
```

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)

```
program omp_loop
use omp_lib
implicit none

integer :: i, mythread

!$omp parallel default(none) XXXX(i) XXXX(mythread)
    mythread = omp_get_thread_num()
    do i=1,16
        print *, 'thread ', mythread, ' gets i=', i
    enddo
!$omp end parallel

end program omp_loop
```

(Advanced: Can combine parallel and for into one omp line.)

SciNet

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    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);
        }
    }
    return 0;
}
```

```fortran
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 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!

```
$ ./omp_loop
 thread              3  gets i=              7
 thread              3  gets i=              8
 thread              4  gets i=              9
 thread              4  gets i=             10
 thread              5  gets i=             11
 thread              5  gets i=             12
 thread              6  gets i=             13
 thread              6  gets i=             14
 thread              1  gets i=              3
 thread              1  gets i=              4
 thread              0  gets i=              1
 thread              0  gets i=              2
 thread              2  gets i=              5
 thread              2  gets i=              6
 thread              7  gets i=             15
 thread              7  gets i=             16
$
```

(Advanced: can break up work of arbitrary blocks of code with "omp task" construct.)

DAXPY

- multiply a vector by a scalar, add a vector.

- (a X plus Y, in double precision)

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

- Implement this, first serially, then with OpenMP

- daxpy.c or daxpy.f90

- `make daxpy` or `make fdaxpy`

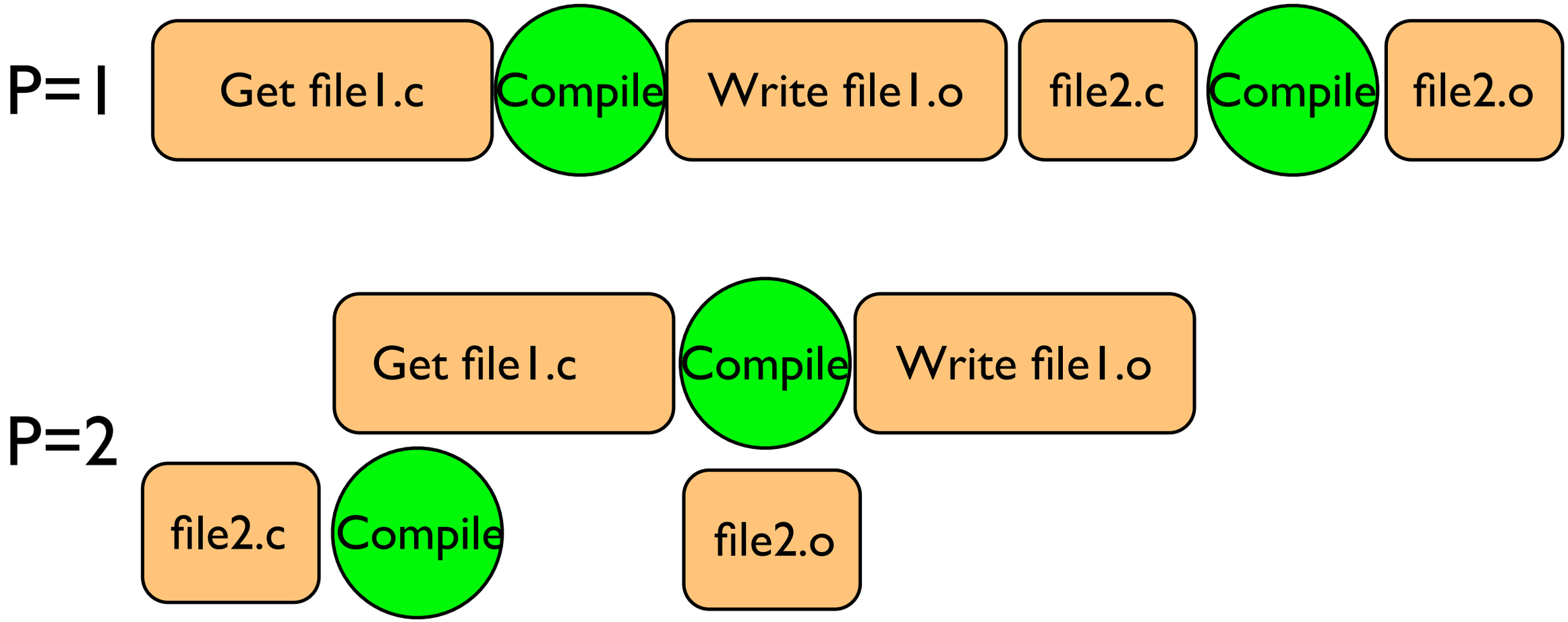
SciNet

make

- Make builds an executable from a list of source code files and rules
- Many files to do, of which order doesn't matter for most
- Parallelism!
- make -j N - launches N processes to do it
- make -j 2 often shows speed increase even on single processor systems

```
$ make
$ make -j 2
$ make -j
```

Overlapping Computation with I/O



```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i]  + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType 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; NType is a numerical type which can be set to sin or double precision

```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i]  + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType 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, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i]  + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType 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, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i]  + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
```

Driver - do timings, etc. (nothing needs to be changed in here).

OpenMPing DAXPY

- How do we OpenMP this?
- Try it (~5-10 min)

```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i]  + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
    return 0;
}
```

SciNet

```
void daxpy(int n, NType a, NType *x, NType *y, NType *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] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

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

```
!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$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.

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers     1.0347e-01 seconds.

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers     1.8619e-01 seconds.
```

```
$ ./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.

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers     1.0347e-01 seconds.

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers     1.8619e-01 seconds.
```

3.69x speedup, 46% efficiency

2.44x speedup, 61% efficiency

1.86x speedup, 93% efficiency

```
void daxpy(int n, NType a, NType *x, NType *y, NType *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] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

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

Why is this safe?
Everyone's modifying x,y,z

```
!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$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
```

Dot Product

- Dot product of two vectors

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

- Implement this, first serially, then with OpenMP

- ndot.c or ndot.f90

- `make ndot` or `make ndotf`

- Tells time, answer, correct answer.

```
$ ./ndot
Dot product is     3.3333e+20
(vs  3.3333e+20) for n=10000000.
Took   5.3578e-02 seconds.
```

```
  ...main program...
  print *, 'Dot product is ', res, '(vs ', ans,') for n = ',n,'.
Took ', time, 'sec.'

  deallocate(x,y)

  contains

  double precision function calc_ndot(n, x, y)
      implicit none
      integer, intent(in) :: n
      double precision, dimension(n)  :: x
      double precision, dimension(n)  :: y
      double precision :: ndot
      integer :: i

      ndot = 0.
      do i=1,n
          ndot = ndot + x(i)*y(i)
      enddo
      calc_ndot = ndot
  end function calc_ndot
```

How to OpenMP this?

SciNet

fomp_ndot_race.f90
omp_ndot_race.c

```
    double precision function calc_ndot(n, x, y)
        implicit none
        integer, intent(in) :: n
        double precision, dimension(n)  :: x
        double precision, dimension(n)  :: y
        double precision :: ndot
        integer :: i
!$omp parallel default(none) shared(ndot,x,y,n) private(i)
        ndot = 0.
        do i=1,n
            ndot = ndot + x(i)*y(i)
        enddo
!$omp end parallel
        calc_ndot = ndot
    end function calc_ndot
```

fomp_ndot_race.f90
omp_ndot_race.c

```
    double precision function calc_ndot(n, x, y)
        implicit none
        integer, intent(in) :: n
        double precision, dimension(n)  :: x
        double precision, dimension(n)  :: y
        double precision :: ndot
        integer :: i
!$omp parallel default(none) shared(ndot,x,y,n) private(i)
        ndot = 0.
        do i=1,n
            ndot = ndot + x(i)*y(i)
        enddo
!$omp end parallel
        calc_ndot = ndot
    end function calc_ndot
```

```
$ ./ndotf
 Dot product is   3.3333283333717098E+020 (vs   3.3333336346987 3840E+020 )
for n =     10000000 .  Took   5.00000007E-02 sec.
$ export OMP_NUM_THREADS=8
$ ./fomp_ndot_race
 Dot product is   6.0689806100371 2922E+019 (vs   3.3333336346987 3840E+020 )
for n =     10000000 .  Took   0.16300000     sec.
```

Wrong answer - and slower!

SciNet

Race Condition - why it's wrong

- Classic 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

ndot = 0.

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

ndot =2

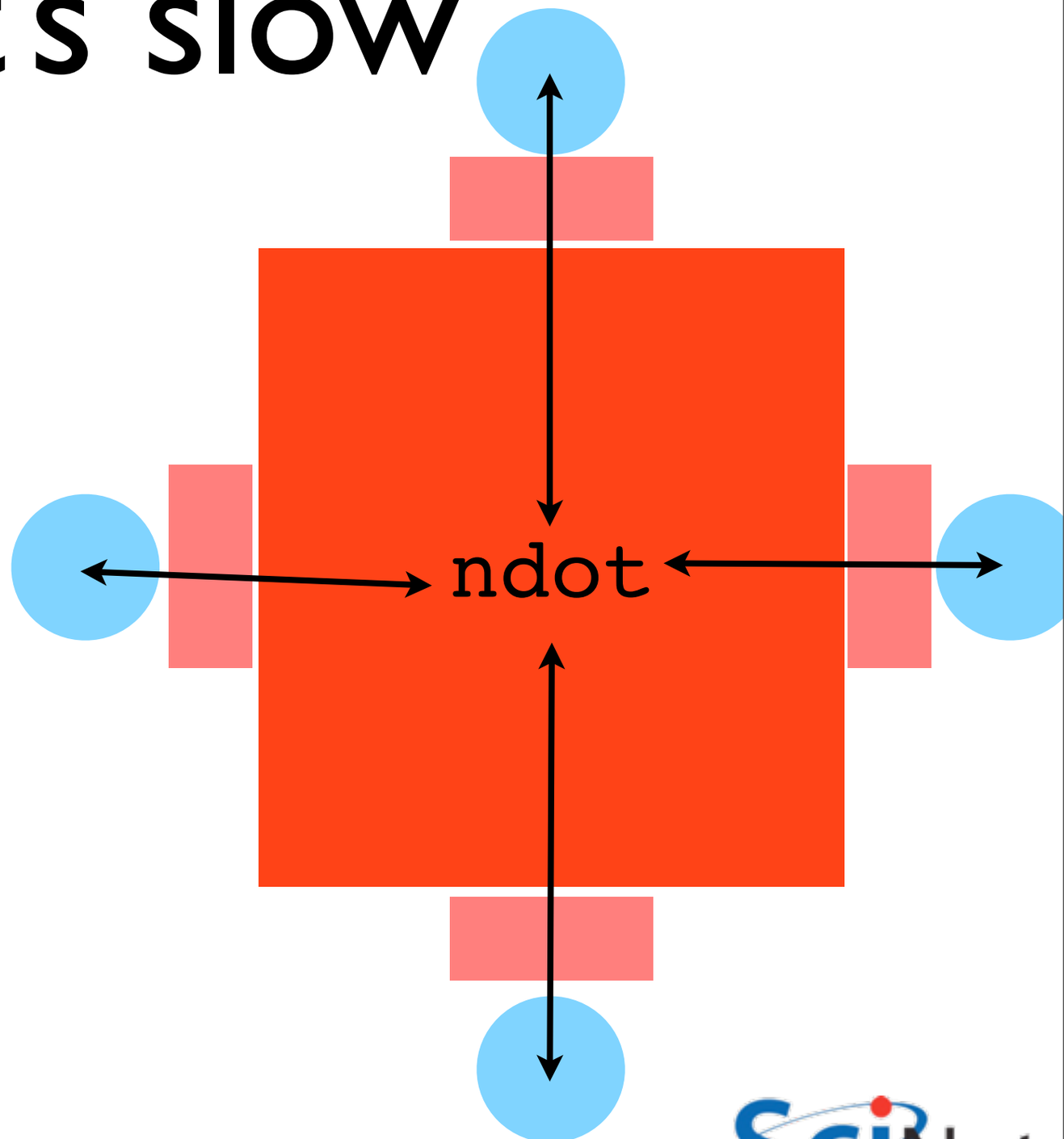
SciNet

Memory contention - 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

OpenMP critical construct

- Defines a "critical region"
- Only one thread can be operating within this region at a time
- Keeps modifications to shared resources safe
- `#pragma omp critical` or `!$omp critical` / `!$omp end critical`

```
NType ndot_critical(int n, NType *x, NT
{
    NType tot=0;
#pragma omp parallel for shared(x,y,n,t
    for (int i=0; i<n; i++)
#pragma omp critical
        tot += x[i] * y[i];
    return tot;
}
```

```
        ndot = 0.
!$omp parallel default(none) shared(ndo
!$omp do
        do i=1,n
!$omp critical
            ndot = ndot + x(i)*y(i)
!$omp end critical
        enddo
!$omp end parallel
        calc_ndot = ndot
    end function calc_ndot
```

SciNet

OpenMP atomic construct

- Most hardware has support for atomic (indivisible - eg, can't get interrupted) instructions
- Small subset, but load/add/store usually one
- Not as general as critical
- Much lower overhead
- Better -- 'only' 18x slower than serial! Still some overhead, still memory contention.

```
$ ./ndot
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    5.3570e-02 seconds.

$ ./omp_ndot_atomic
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    9.7981e-01 seconds.
```

SciNet

How should we fix this?

$$
\begin{aligned}
n &= \hat{x} \cdot \hat{y} \\
&= \sum_i x_i y_i
\end{aligned}
$$

How should we fix this?

- Local sums
- Each processor sums its local value (10^7/P additions)
- And *then* sums to ntot (only P additions) with critical, or atomic..
- Try this (5-10 min)
- cp one of the omp_ndot.c's or fomp_ndot.c's to omp_ndot_local.c (or fomp_ndot_local.f90)

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

SciNet

Local variables:

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

    #pragma omp atomic
    tot += mytot;
}
```

```
ndot = 0.
!$omp parallel default(none)
  shared(ndot,n,x,y) private(i,mytot)
  mytot = 0.
!$omp do
    do i=1,n
       mytot = mytot + x(i)*y(i)
    enddo
!$omp atomic
    ndot = ndot + mytot
!$omp end parallel
calc_ndot = ndot
```

```
$ ./ndot
Dot product is     3.3333e+20
(vs     3.3333e+20) for n=10000000.
Took   5.3570e-02 seconds.
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is     3.3333e+20
(vs     3.3333e+20) for n=10000000.
Took   1.8334e-02 seconds.
```

SciNet

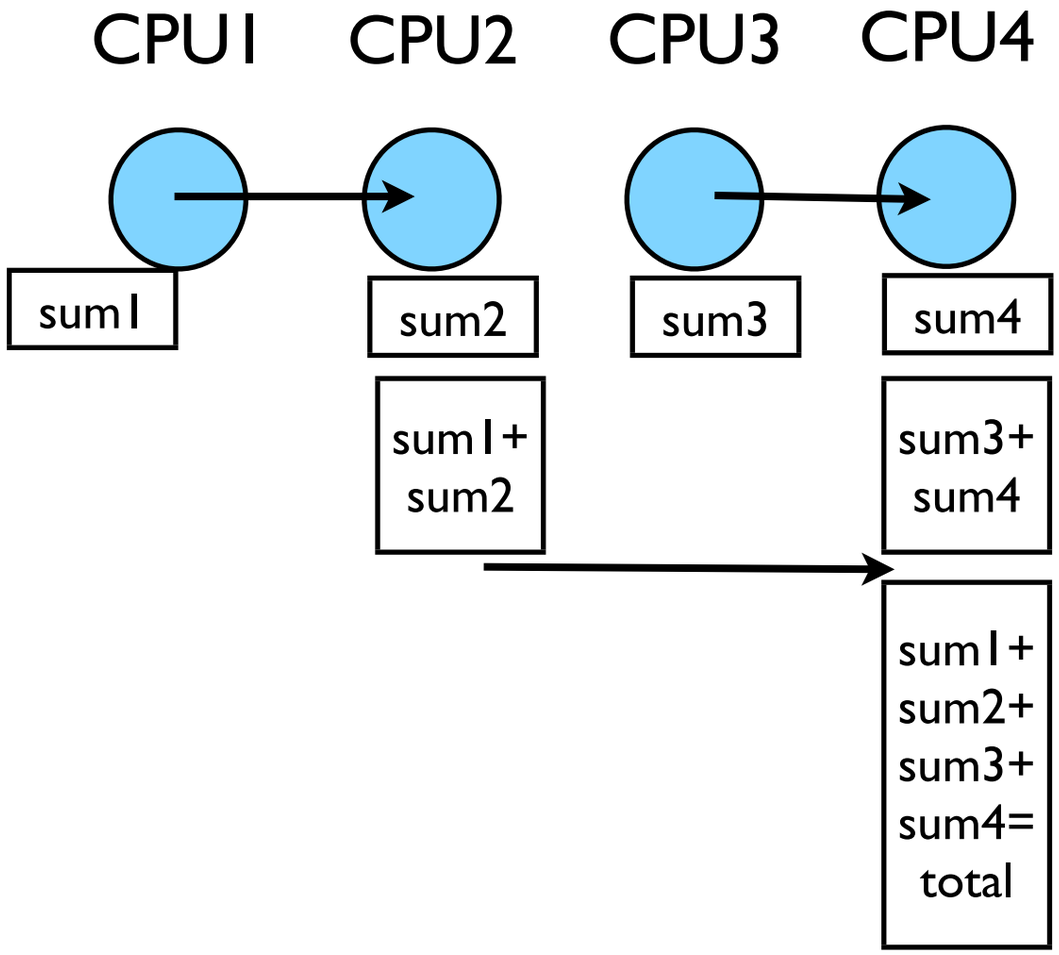
OpenMP Reduction Operations

- This is such a common operation, there 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



Reduction; works for a variety of operators (+,*,min,max...)

OpenMP Reduction Operations

```
NType ndot_atomic(int n, NType *x, NType *y)
{
    NType 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];
}
    return tot;
}
```

OpenMP Reduction Operations

```
double precision function calc_ndot(n, x, y)
implicit none
integer, intent(in) :: n
double precision, dimension(n)  :: x
double precision, dimension(n)  :: y
double precision :: ndot
integer :: i

ndot = 0.
!$omp parallel default(none) shared(n,x,y) reduction(+:ndot) private(i)
!$omp do
    do i=1,n
        ndot = ndot + x(i)*y(i)
    enddo
!$omp end parallel
calc_ndot = ndot

end function calc_ndot
```

SciNet

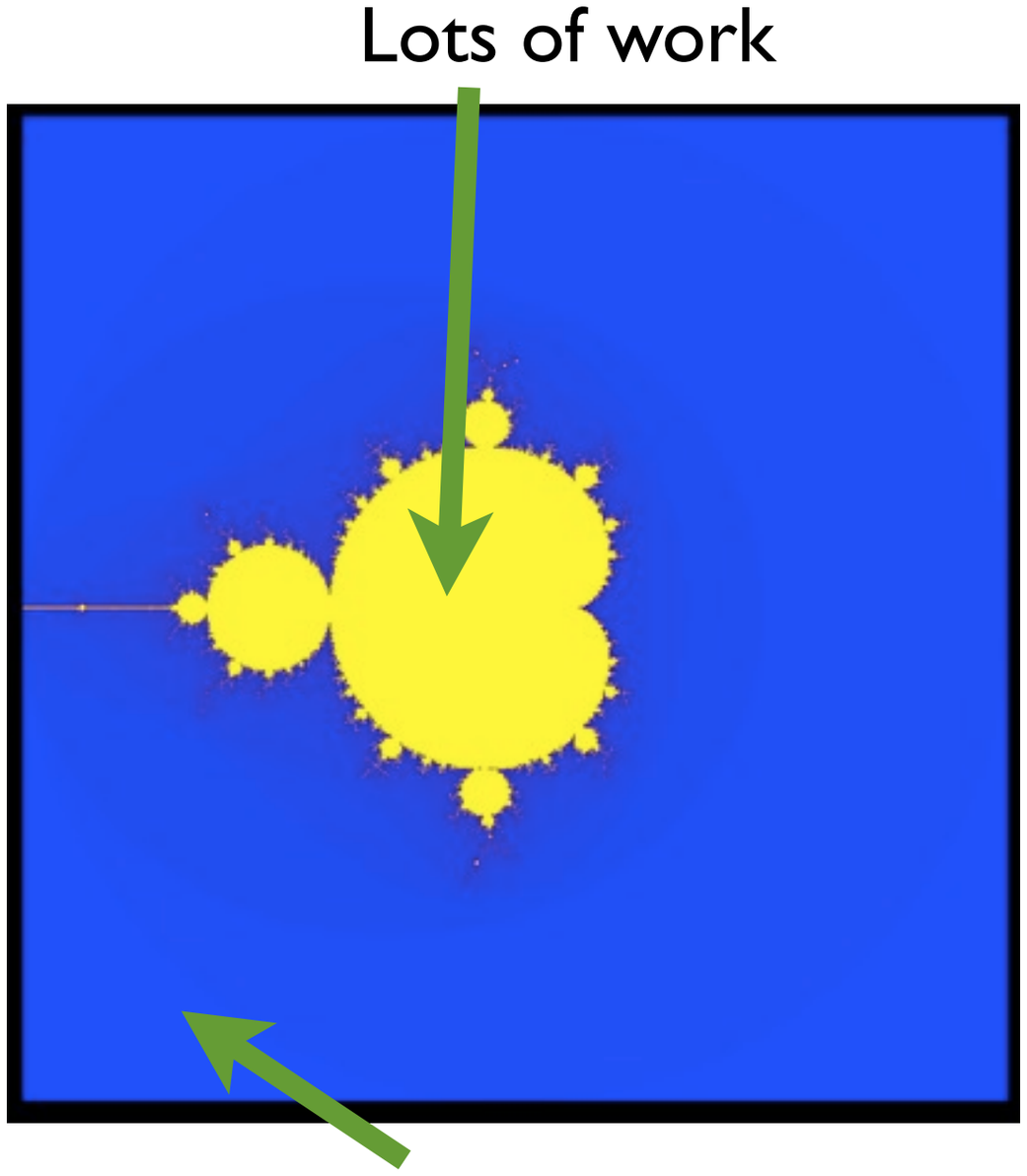
Load-Balancing

- So far, every iteration of the loop has had the same amount of work:

- Not always the case

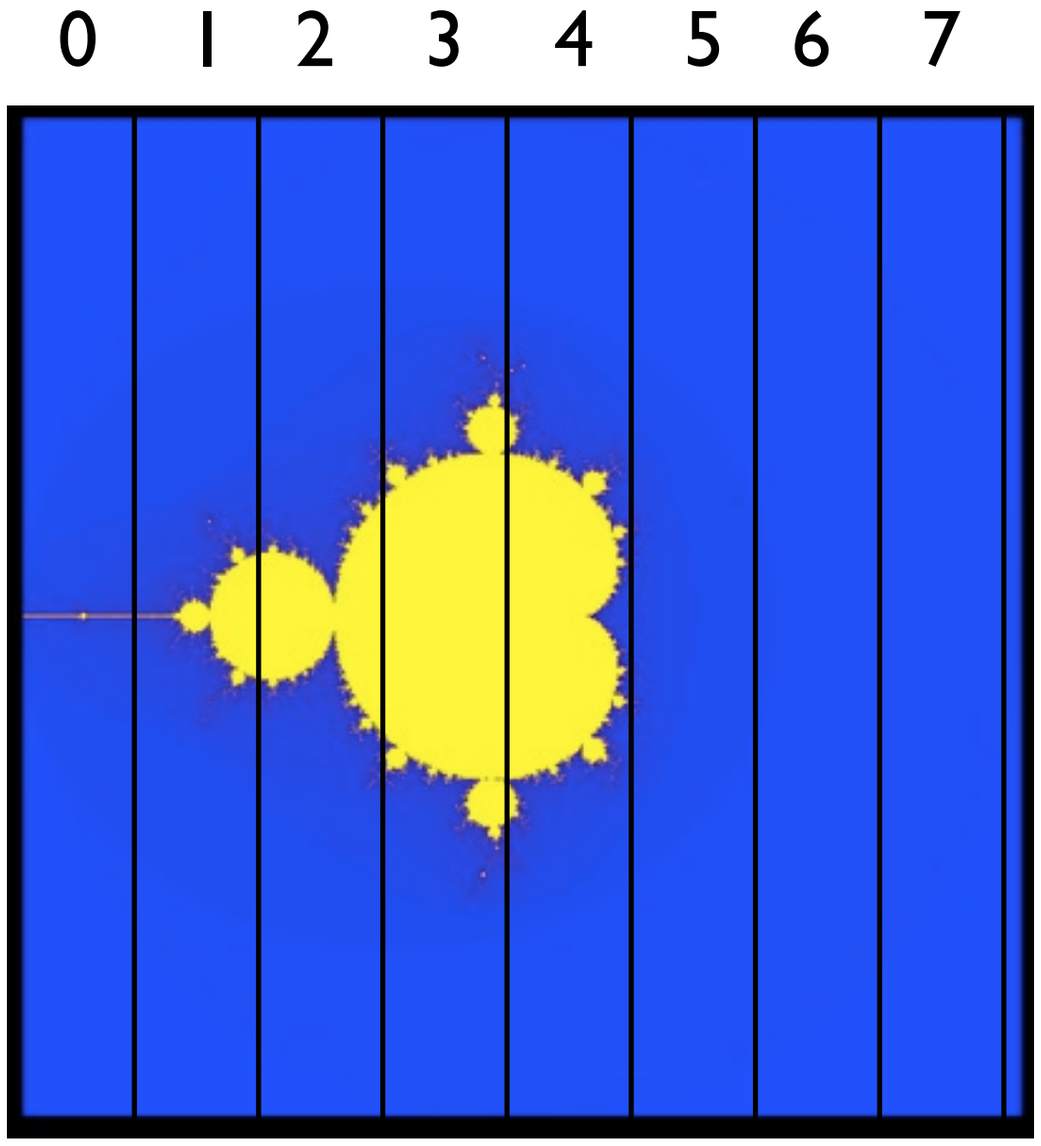
- make mandel; ./mandel

- Plots a function at every pixel with different amount of work - in fact, amount of work is basically the plotted color.



SciNet

Load-Balancing



- Default work sharing breaks N iterations into ~N/nthreads contiguous chunks and assigns them to threads

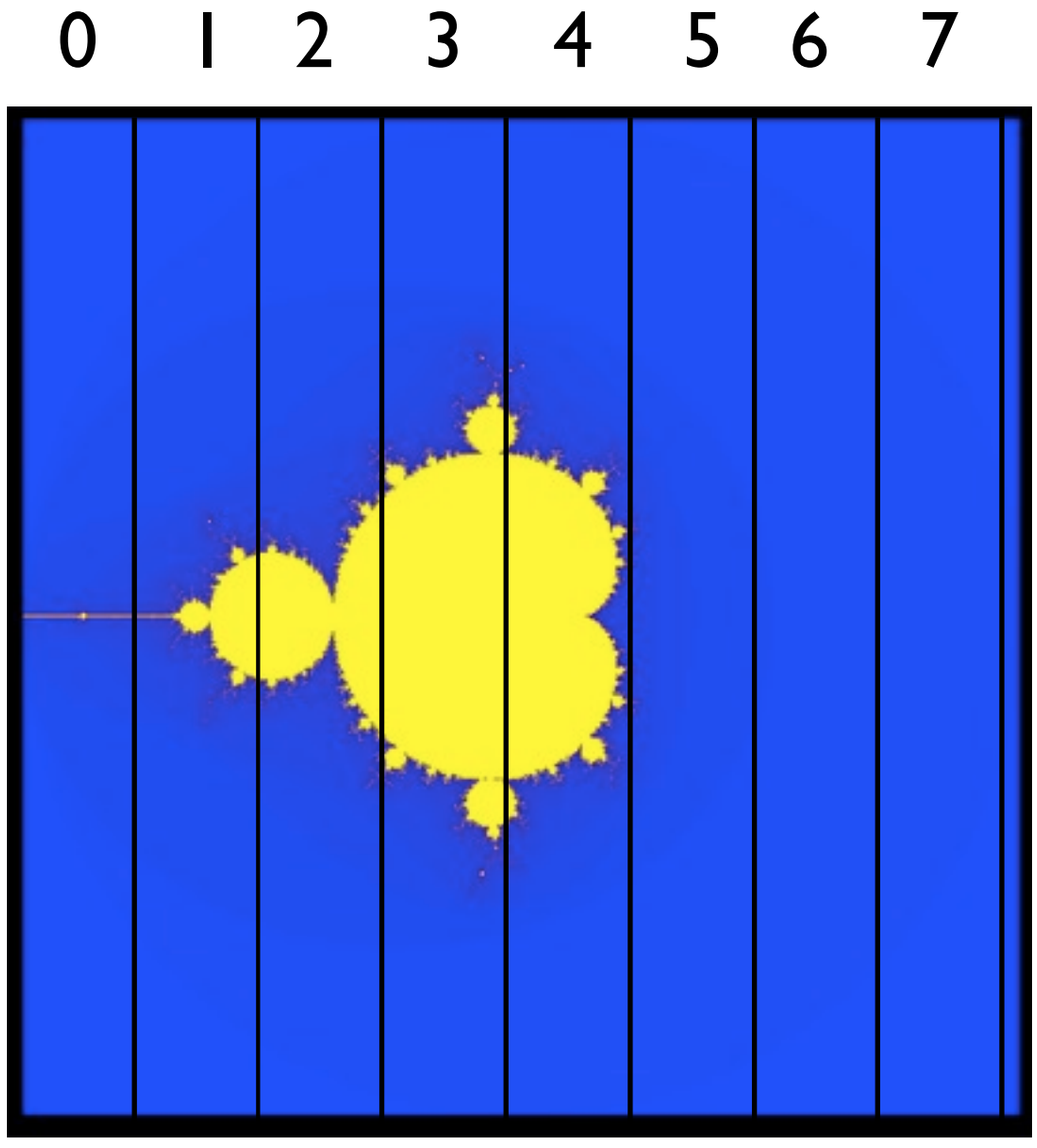
- But now threads 7, 6, 5 will be done and sitting idle while threads 3,4 work alone...

- Inefficient use of resources

SciNet

Load-Balancing

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



800x800 pix; N/nthreads ~ 100x800

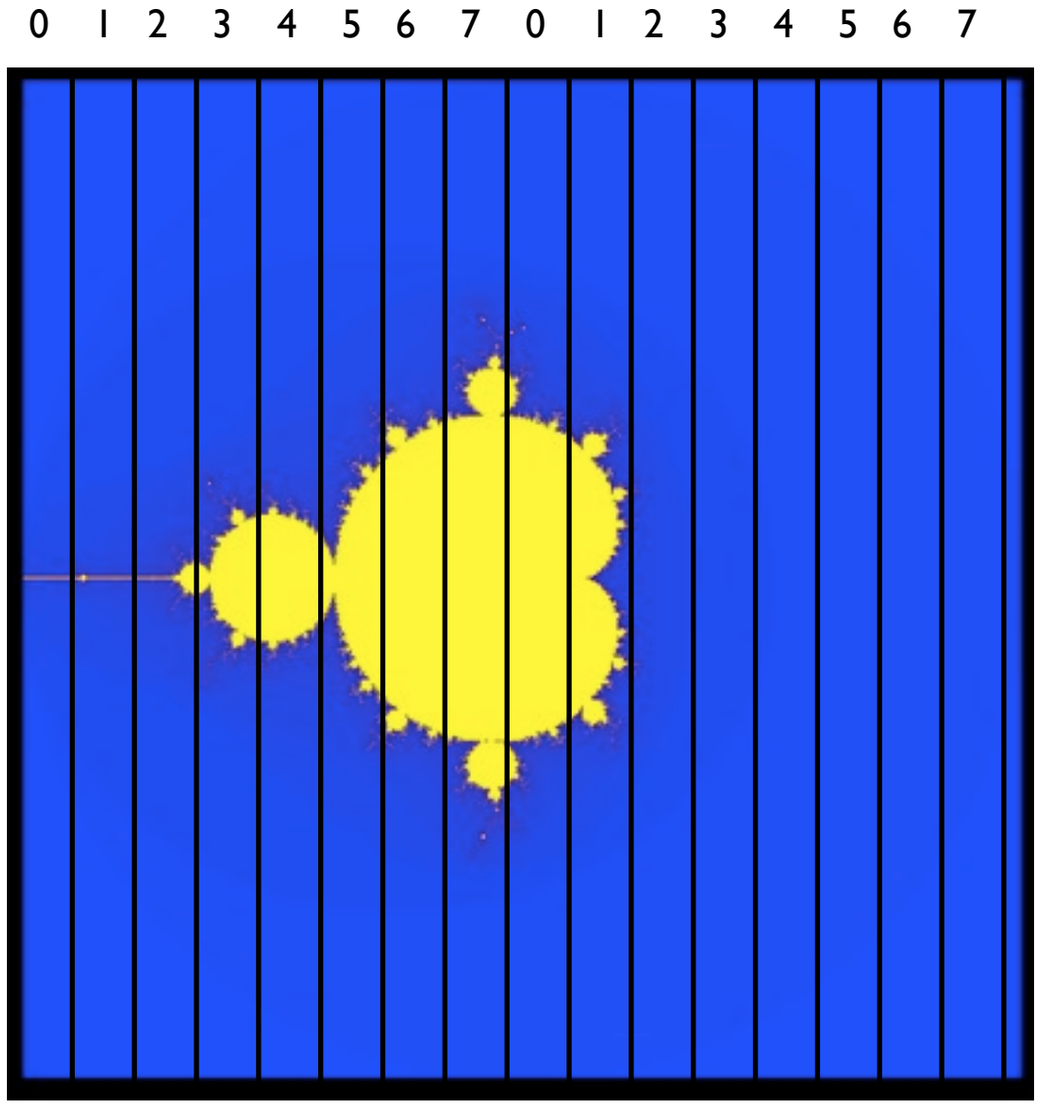
SciNet

Load-Balancing

- Can change the `chunk size' from ~N/nthreads to arbitrary number

- 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 work chunk.



SciNet

Load-Balancing

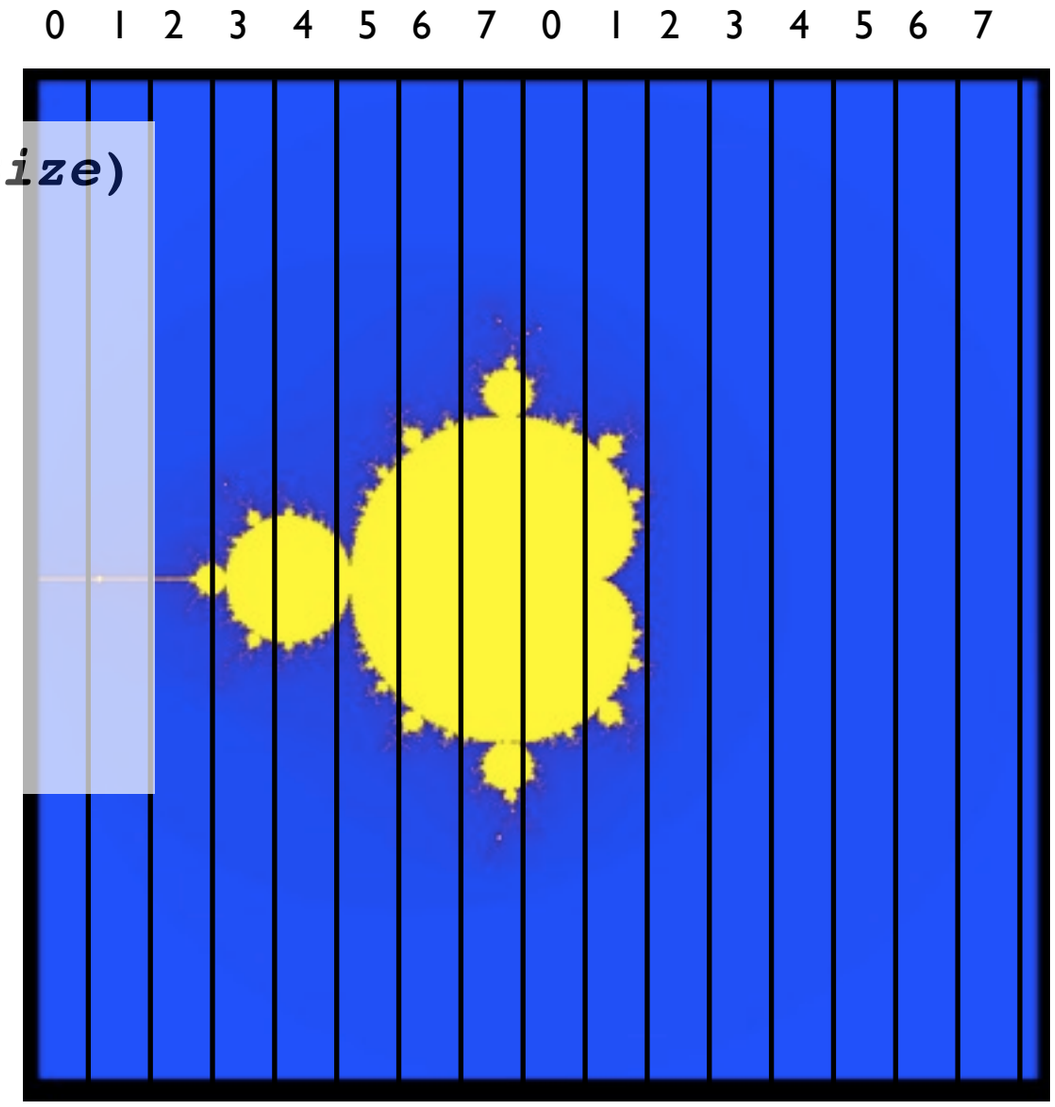
```
#pragma omp for schedule(static,chunksize)
```

or

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

Here, ***chunksize*** = 50.

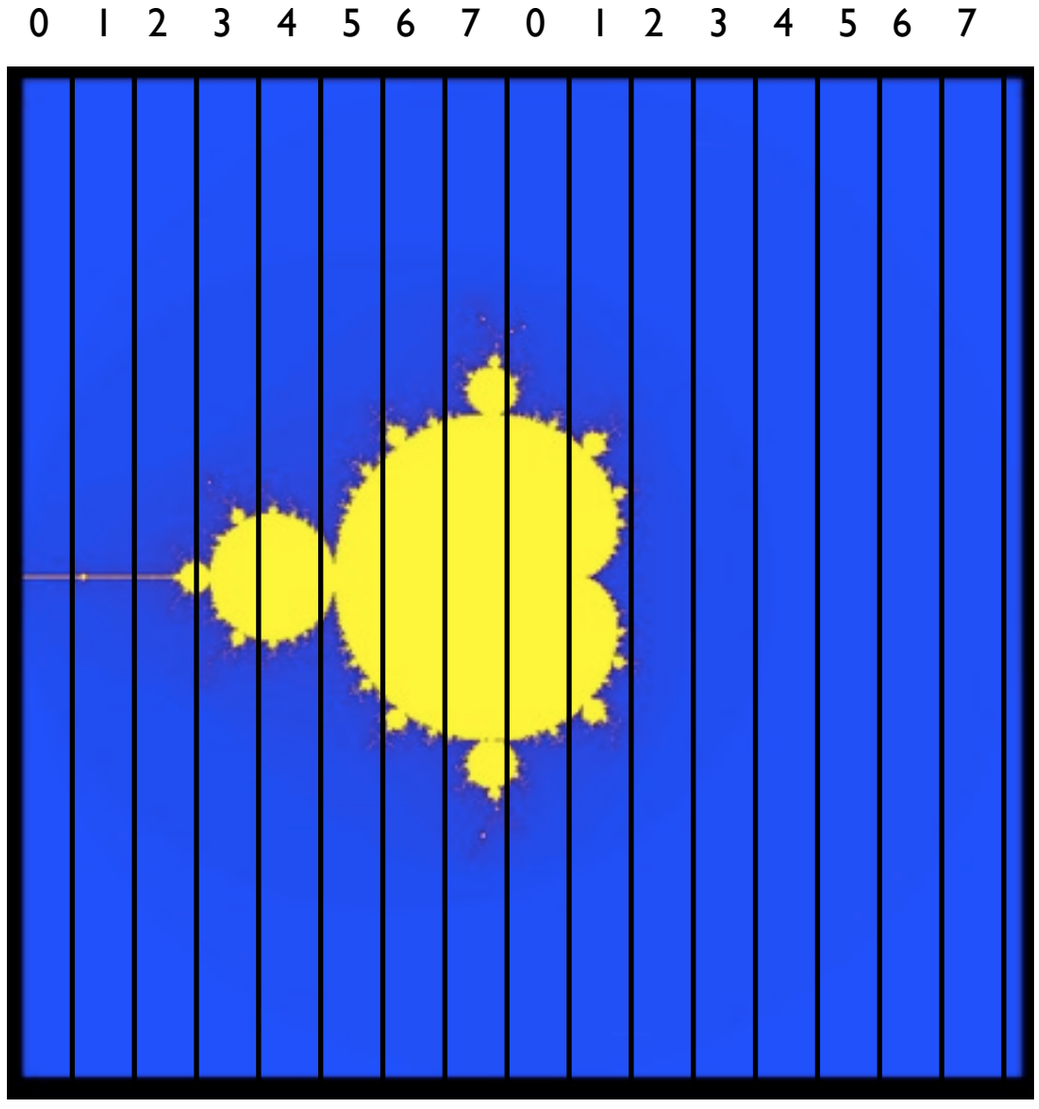
Static scheduling



SciNet

schedule(static,50)

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



SciNet

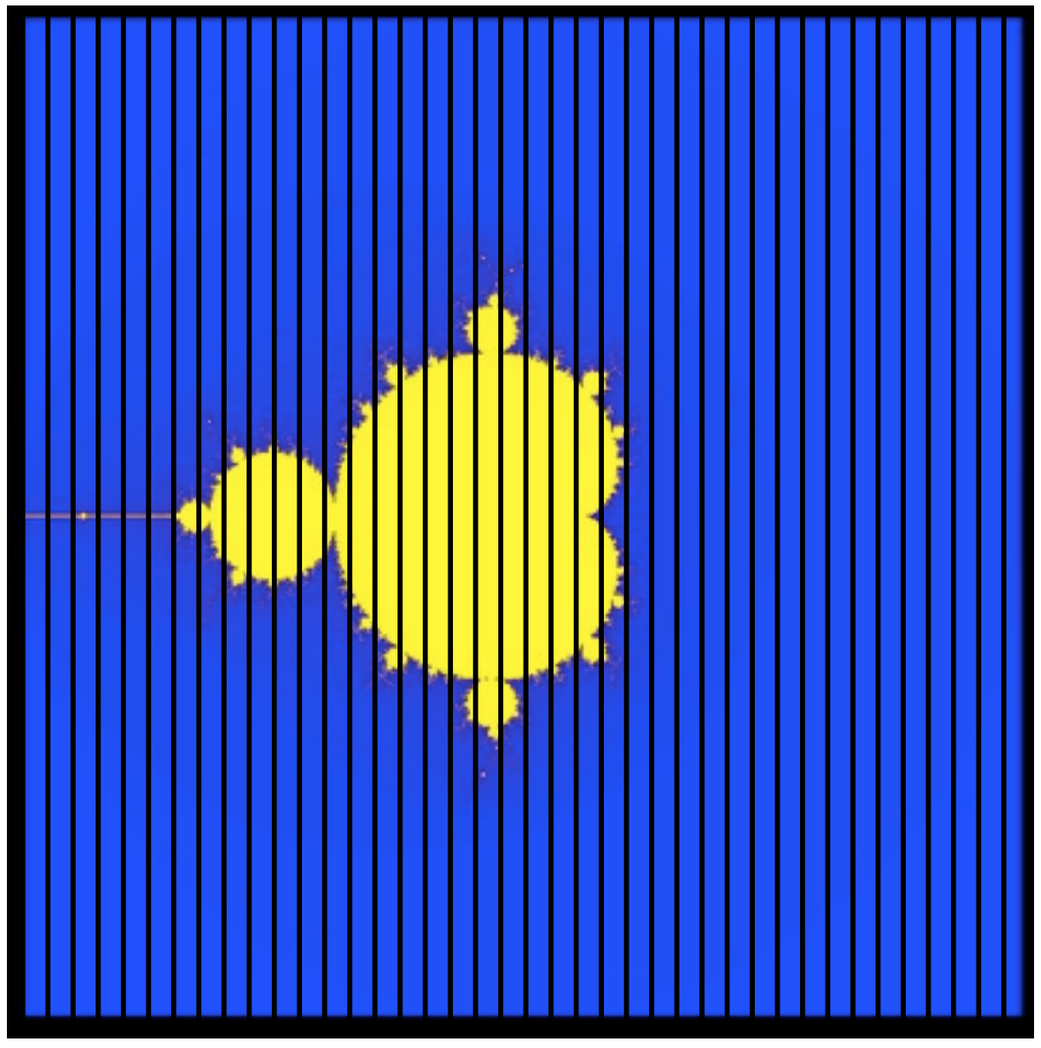
schedule(dynamic)

- Still another choice is to break it up into many pieces and hand them to threads when they are ready

- dynamic scheduling

- Has increased overhead, but can do a very good job

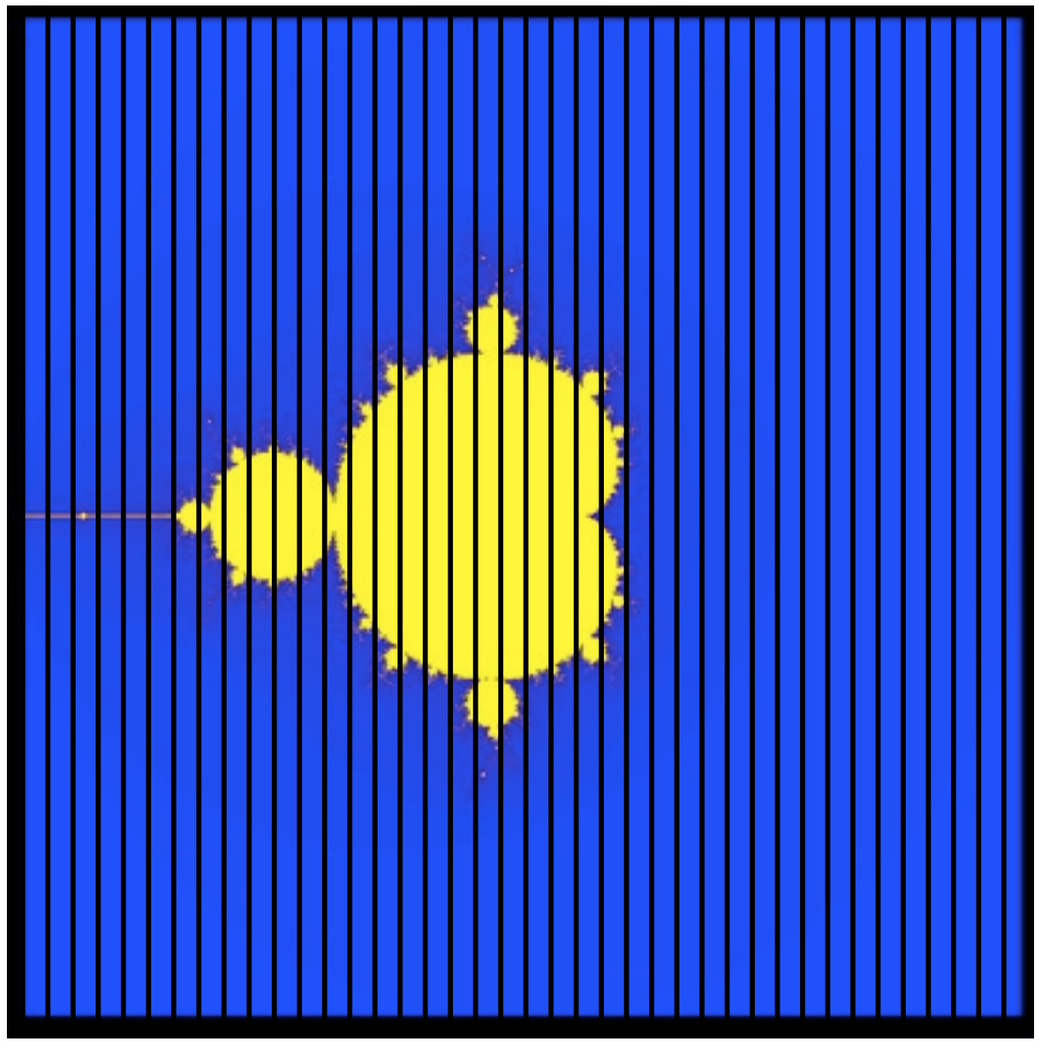
- can also choose chunksize for dynamic



SciNet

schedule(dynamic)

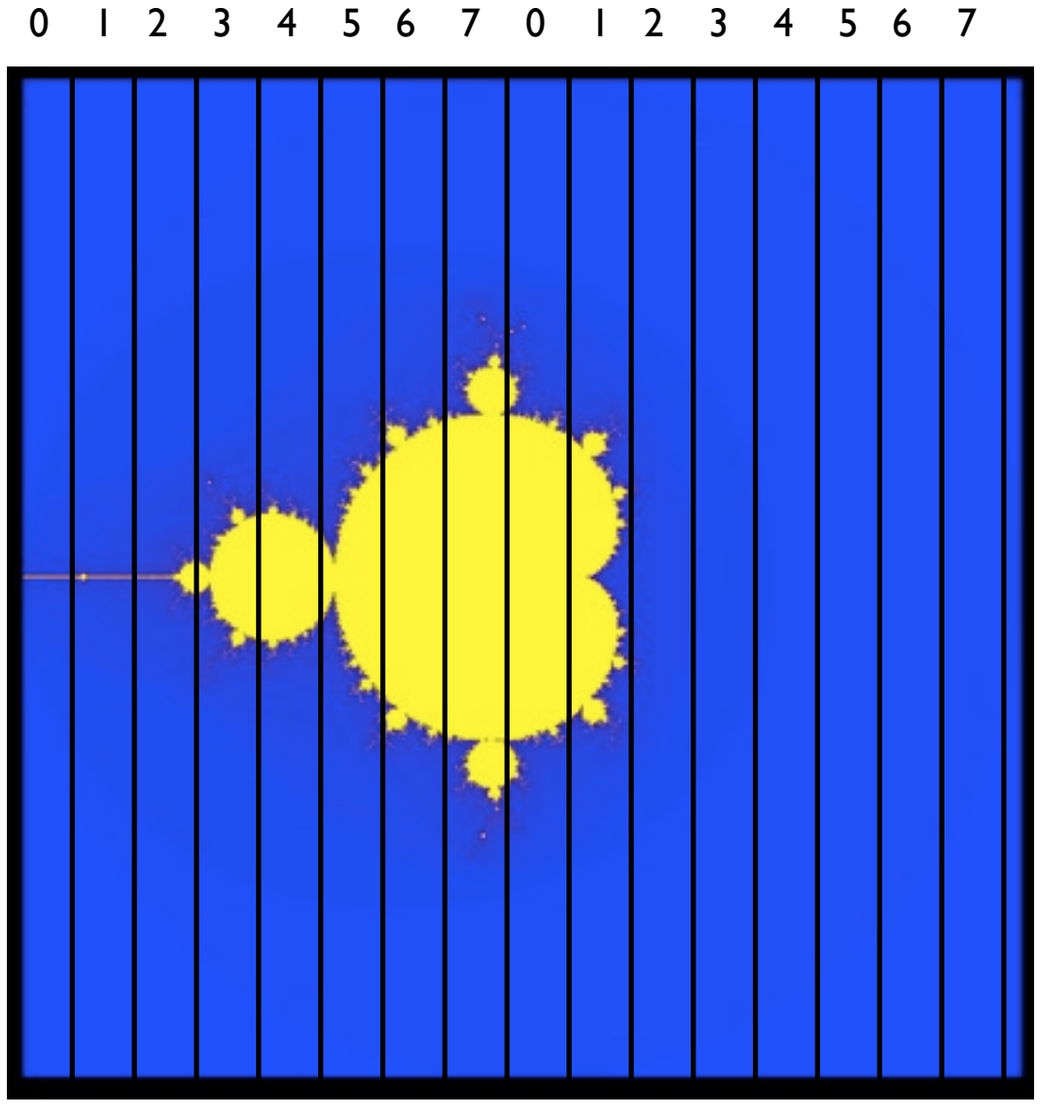
Serial	0.63s
Nthreads=8	0.10
Speedup	6.3x
Efficiency	79%



SciNet

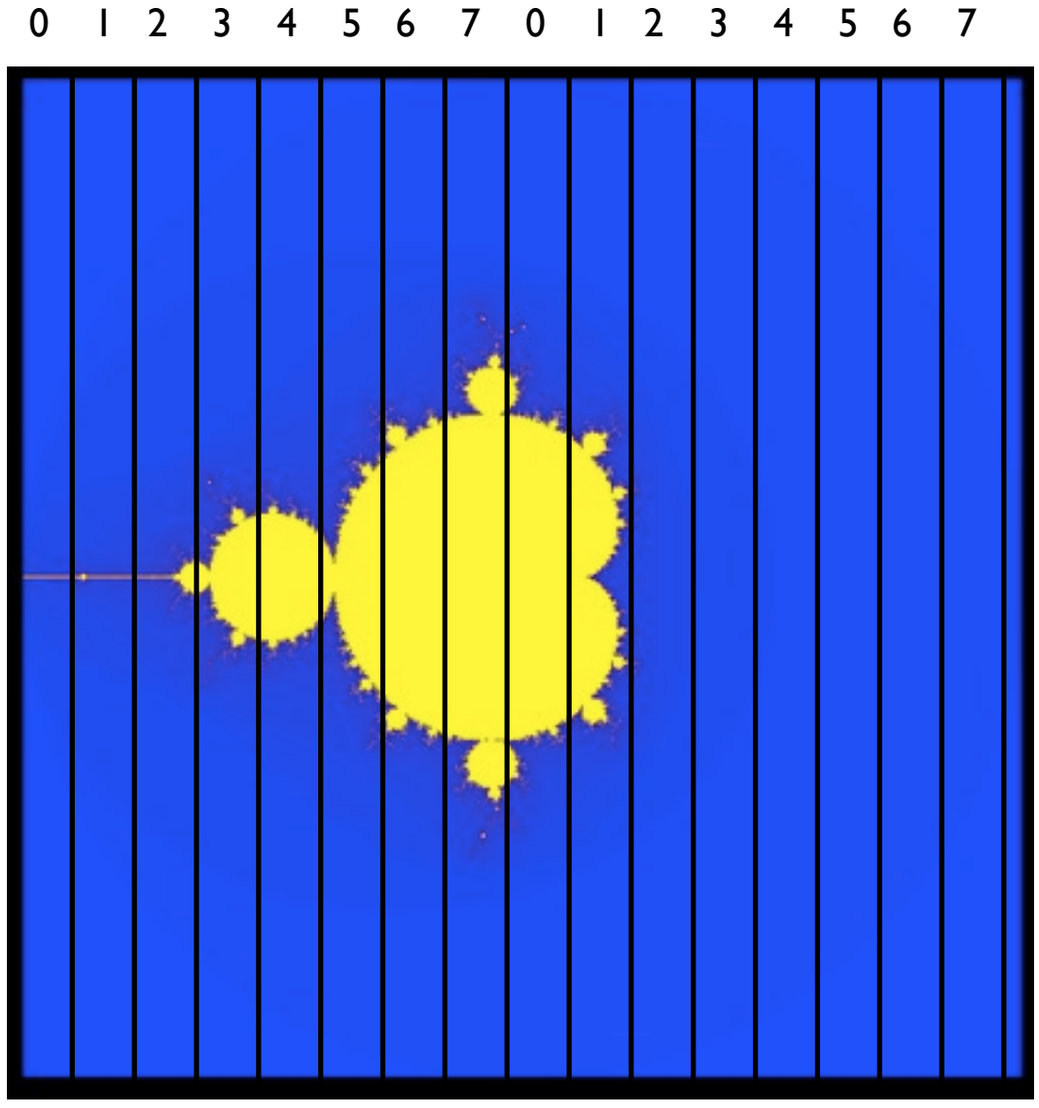
Tuning

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



Tuning

(static,4)	(dynamic,16)
0.084s	0.099s
7.6x	6.4x
95%	80%



SciNet

Two-level loops

- In scientific code, we usually have nested loops where 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)np
       double y=((double)j)/((double)np
       double complex a=x+I*y;
       mymap[i][j]=how_many_iter_real(a
}
```

mandel.c

Summary

- omp parallel
- omp single
- shared/private/reduction variables
- omp atomic, omp critical
- omp for