Shared Memory Programming with OpenMP

Ramses van Zon
SciNet HPC Consortium
University of Toronto

July 11-12, 2016

Outline

1. The OpenMP model: threads, memory, and performance

   Hands On 1: Parallelizing daxpy

2. Reductions and load balancing

   Hands-On 2: Mandelbrot set

3. Further OpenMP features

4. Heterogeneous computing with OpenMP

Assumed knowledge: C and/or Fortran scientific programming; experience editing and compiling code in a Linux environment.

The OpenMP programming model

SciNet compute • calcul CANADA

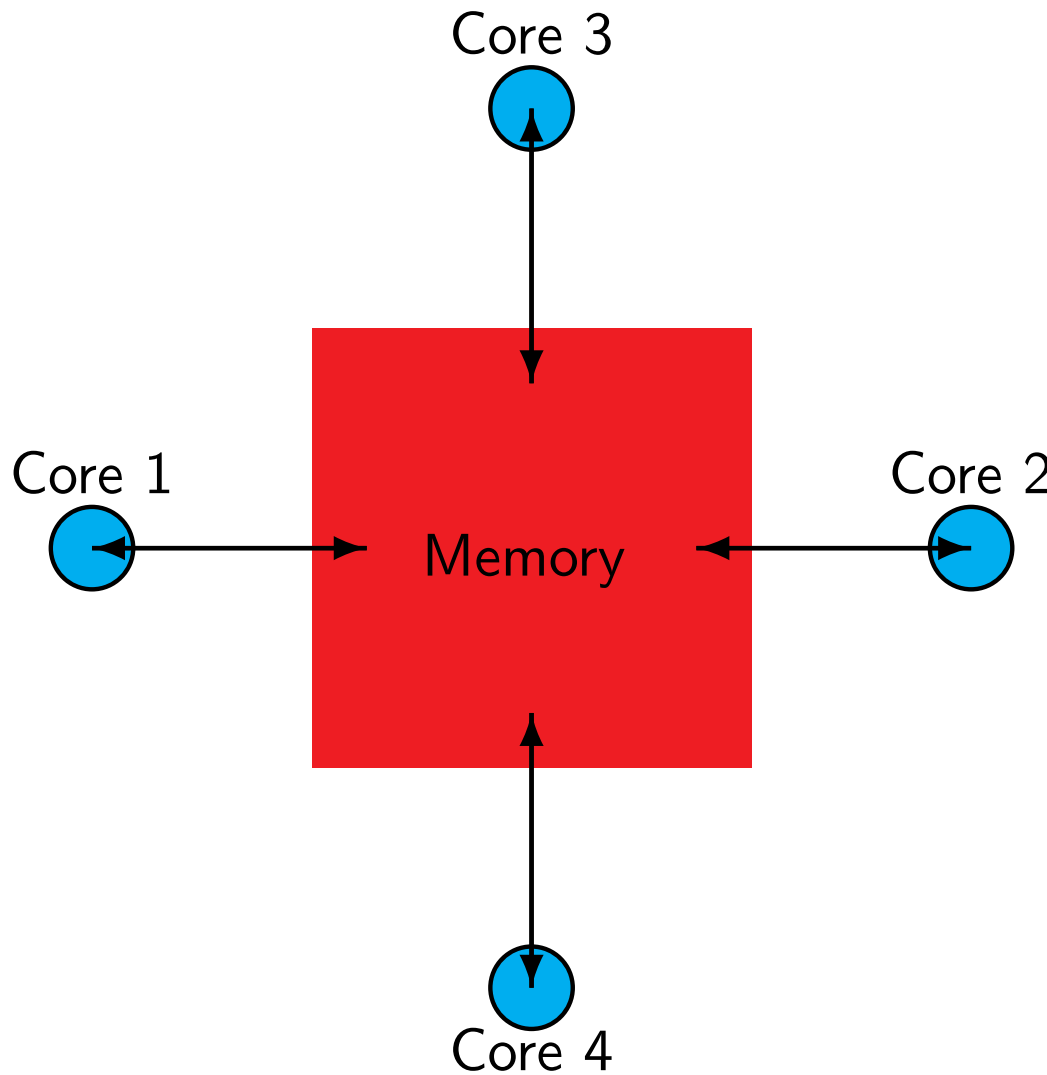
Shared Memory

One large chunk of memory, different computing cores acting on it. All 'see' same data.

Any coordination done through memory

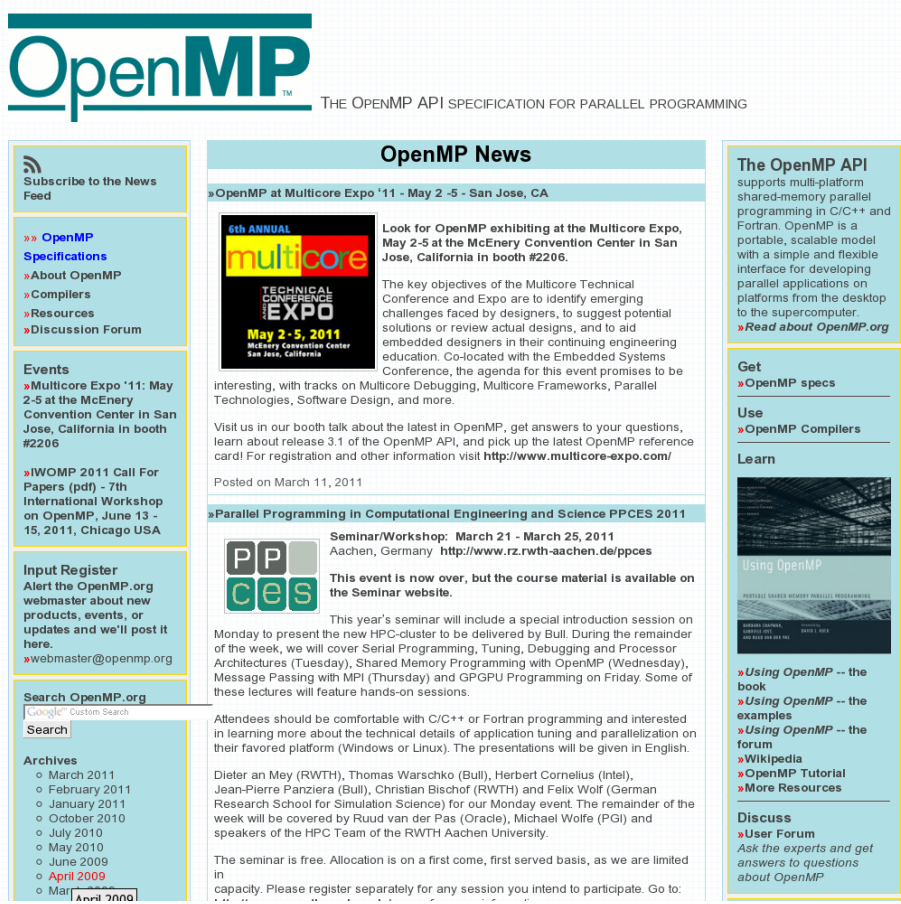
Could use message passing, but no need.

Each code is assigned a thread of execution of a single program that acts on the data.



OpenMP

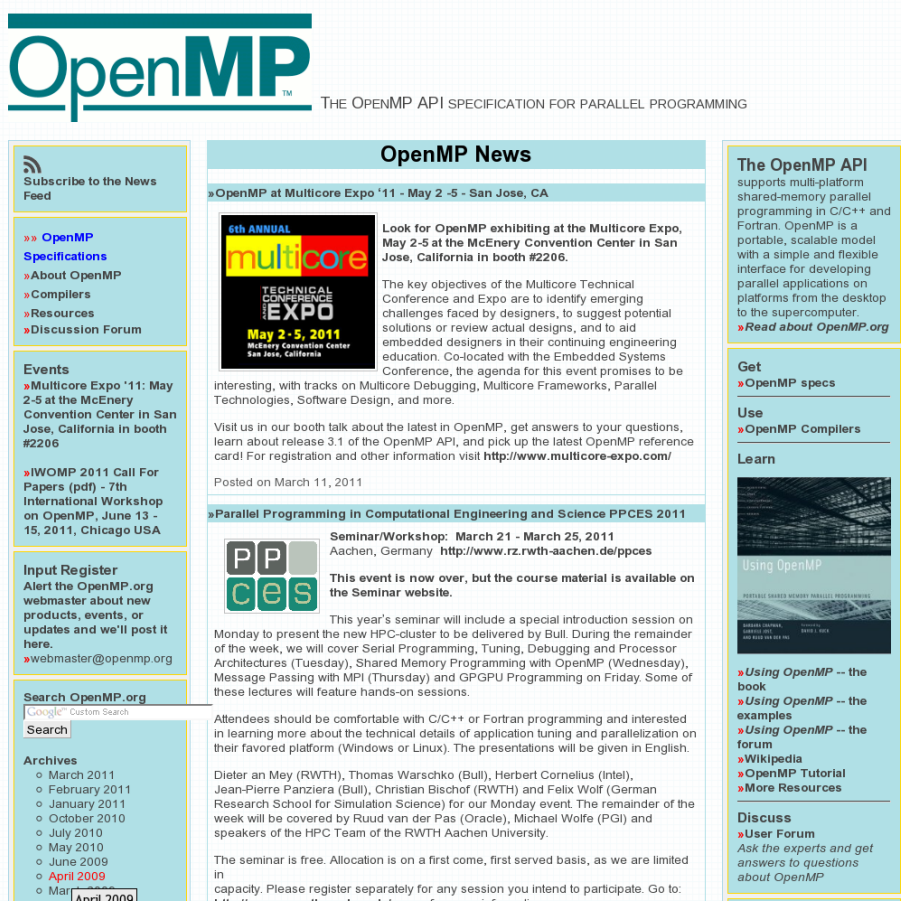
- For shared memory systems.
- Add parallelism to functioning serial code.
- http://openmp.org



OpenMP

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

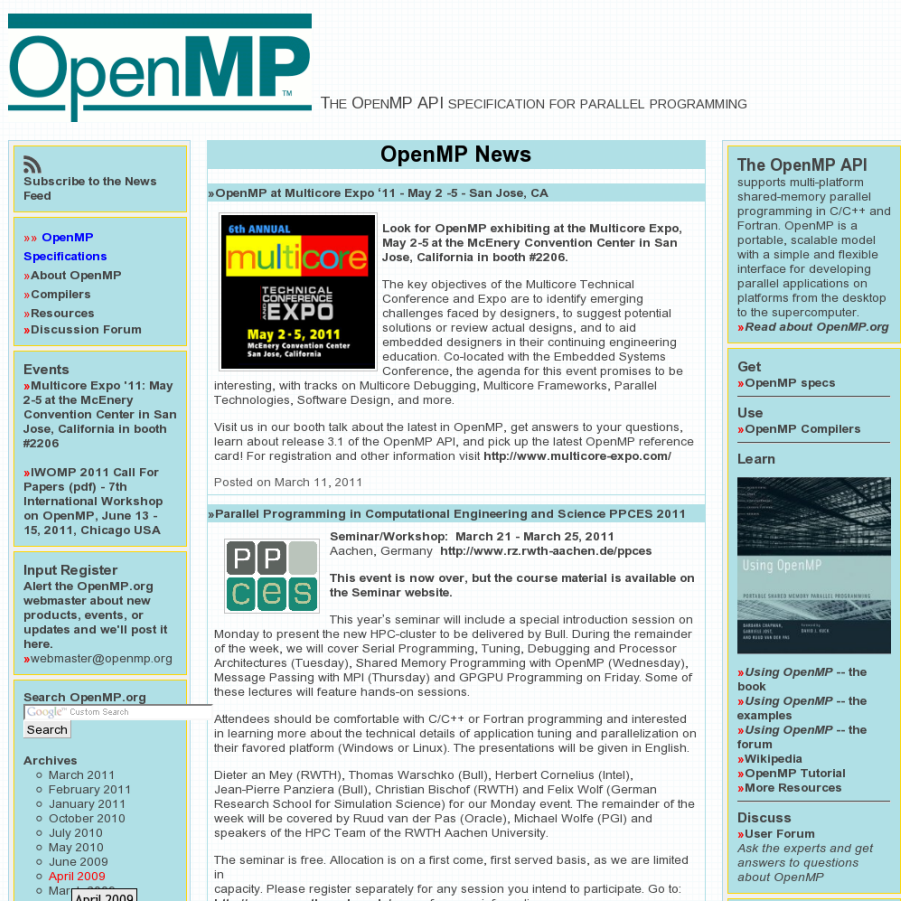
- Compiler, run-time env does most of the work
- But we have to tell it how to use variables, where to run in parallel, ...
- Mark parallel regions.
- Works by adding compiler directives to code.



OpenMP

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

- Compiler, run-time env does most of the 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.

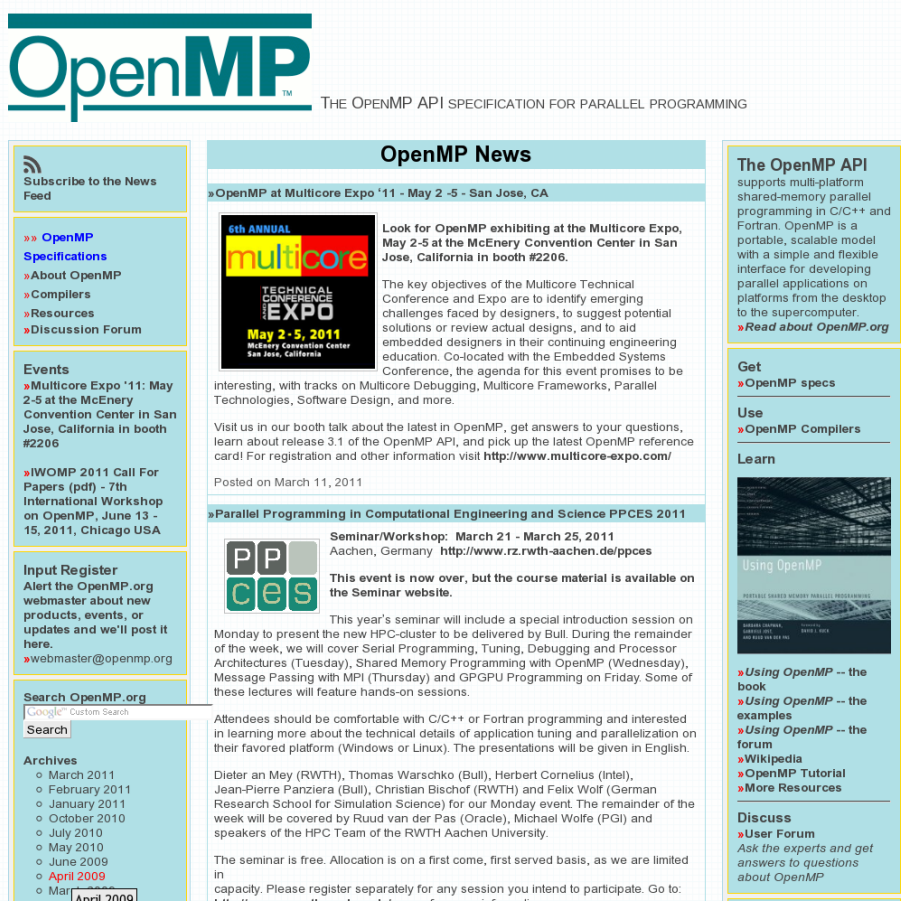


OpenMP

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

- Compiler, run-time env does most of the 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.



Incremental parallelism

Compiler directives-based parallization

- OpenMP
- OpenACC
  - Does for GPU programming what OpenMP does for threading
  - Alternative to CUDA (but no free implementation yet).
  - Similar incremental parallelism as for OpenMP
  - Differs from OpenMP in that memory needs to be copied over
- Intel MIC Compilers
  - MIC, or more proper, the Xeon Phi, is an Intel multicore co-processor with its own memory.
  - Host/Device setup is similar to the CPU, but internal architecture is shared-memory x86.
  - With the Intel compilers (v13+) you can use compiler directives for offloading to the MIC as well.
- Compiler-specific vectorization hints

Much of this is in OpenMP 4, but not fully supported by compilers.

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.

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.

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: 1_helloworld/omp-hello-world.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());
  }
}
```

Fortran: 1_helloworld/omp-hello-world-f.f90

```
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
```

Getting started with this code

```
$ ssh USER@login.scinet.utoronto.ca -X   #get into SciNet
$ ssh gpc01 -X                           #get on the GPC
$ qsub -l nodes=1:ppn=8,walltime=7:00:00 -q teach -X -I
...                                      #get your own compute node
$ cd $SCRATCH
$ cp -r /scinet/course/ss2016/openmp .
$ cd openmp/code
$ source setup
$ cd 1_helloworld
```

OpenMP example

```
$ gcc -fopenmp -o omp-hello-world omp-hello-world.c
or
$ gfortran -fopenmp -o omp-hello-world-f \
   omp-hello-world-f.f90
or
$ make omp-hello-world omp-hello-world-f

$ 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!
```


SciNet compute • 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".

```
$ 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!
```


SciNet compute • calcul CANADA

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());
  }
}
```

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

```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

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());
  }
}
```

```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, launching OMP_NUM_THREADS threads,
Each executes the same code!

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());
  }
}
```

```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, threads join back up, Execution continues serially.

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)

By including `omp.h`, you get a smal number of omp functions:

- `omp_get_thread_num()`
- `omp_get_num_threads()`
- `...`

1_helloworld/omp-hello-world2.c

```
#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)

1_helloworld/omp-hello-world3.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());
  }
  printf("There were %d threads.\n", omp_get_num_threads());
}
```

What do you think, will this work?

OpenMP functions (from omp.h/omp_lib)

1_helloworld/omp-hello-world3.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());
  }
  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)

1_helloworld/omp-hello-world3.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());
  }
  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.

```
#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

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, launches 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, launches 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, launches threads.
- Each thread gets copy of mythread.
- Only thread 0 writes to nthreads.
- Good idea to declare mythread locally! (avoids many bugs)
- 1_helloworld/omp-hello-world4.c

Variables in OpenMP - Fortran version

1_helloworld/omp-hello-world4-f.f90

```
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

1_helloworld/omp-hello-world4.c

```
#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

```c
#include <stdio.h> // 1_helloworld/omp-hello-world5.c
#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);
}
```

```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

Loops in OpenMP

Consider following openmp programs with a loop.

SciNet compute • calcul CANADA

Loops in OpenMP

Consider following openmp programs with a loop.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none)\
   XXXX(i) private(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) private(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

Consider following openmp programs with a loop.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none)\
   XXXX(i) private(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) private(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
             ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

How should we declare `i`, as `private` or as `shared`?

Loops in OpenMP

Consider following openmp programs with a loop.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int i, mythread;
  #pragma omp parallel default(none)\
   XXXX(i) private(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) private(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
             ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

How should we declare `i`, as `private` or as `shared`?
What would you imagine this does when run with e.g. OMP_NUM_THREADS=8?

Loops in OpenMP

Consider following openmp programs with a loop.

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

2_loop/omp-loop.c

```f90
program omp_loop
use omp_lib
implicit none
integer ::  i, mythread
!$omp parallel default(none) &
!$omp XXXX(i) private(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, &
             ' gets i=', i
  enddo
!$omp end parallel
end program omp_loop
```

2_loop/omp-loop-f.f90

How should we declare `i`, as `private` or as `shared`?

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 work-sharing 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 work-sharing construct: omp for (or omp do).

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

2_loop/omp-loop2.c

SciNet compute • calcul CANADA

Fortran version

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

2_loop/omp-loop2-f.f90

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-loop2
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)
- Given serial implementation, will start adding OpenMP
- daxpy.c or daxpy.f90
- cd 3_daxpy; make daxpy or make daxpy-f

$$z = ax + y$$

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 fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z); free(y); free(x);
}
```

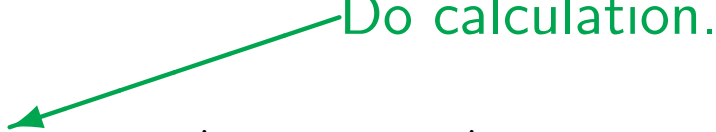
```c
#include <stdio.h>
#include <pca_utils.h>
void fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z); free(y); free(x);
}
```

Utilities for memory and timing

```
#include <stdio.h>
#include <pca_utils.h>
void fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  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 fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z); free(y); free(x);
}
```



```c
#include <stdio.h>
#include <pca_utils.h>
void fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z); free(y); free(x);
}
```

Driver (setup, call, timing).

```c
#include <stdio.h>
#include <pca_utils.h>
void fill(int n, double* x, double* y) {
  for (int i=0; i<n; i++) {
    x[i] = (double)i*i;
    y[i] = (double)(i+1)*(i+1);
  }
}
void daxpy(int n, double a, double *x, double *y, double *z) {
  for (int i=0; i<n; i++)
    z[i] += a*x[i] +y[i];
}
int main() {
  pca_time tt;
  int n=100000000;
  double a = 5./3.;
  double *x=vector(n), *y=vector(n), *z=vector(n);
  for(int i=0;i<n;i++) z[i]=0;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(z); free(y); free(x);
}
```

HANDS-ON: Try OpenMP-ing daxpy...

HANDS-ON 1:

Parallelize daxpy with OpenMP:
Edit the files omp-daxpy.c or omp-daxpy.f90.
Compile with make
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++)
      z[i] += a * x[i] + y[i];
  }
}
```

```fortran
subroutine calc_daxpy(n, a, x, y, z)
  implicit none
  integer, intent(in) ::  n
  double precision, intent(in) ::  a
  double precision, dimension(n), intent(out) ::  x,y,z
  integer ::  i
  !$omp parallel default(none) private(i) shared(a,x,b,y,z)
  !$omp do
  do i=1,n
    z(i) = a*x(i) + y(i)
  enddo
  !$omp end parallel
end subroutine
```

```
$ ./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

Reductions in OpenMP

Dot Product

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

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

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

Dot Product - serial

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

Dot Product - serial

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

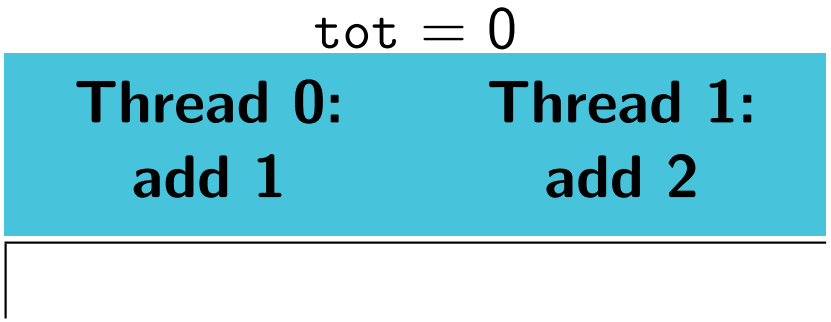
```
double ndot(int n, double *x, double *y){
  double tot=0;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ 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.
```

Wrong answer, and not much faster!

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.



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	

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)

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

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`

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.

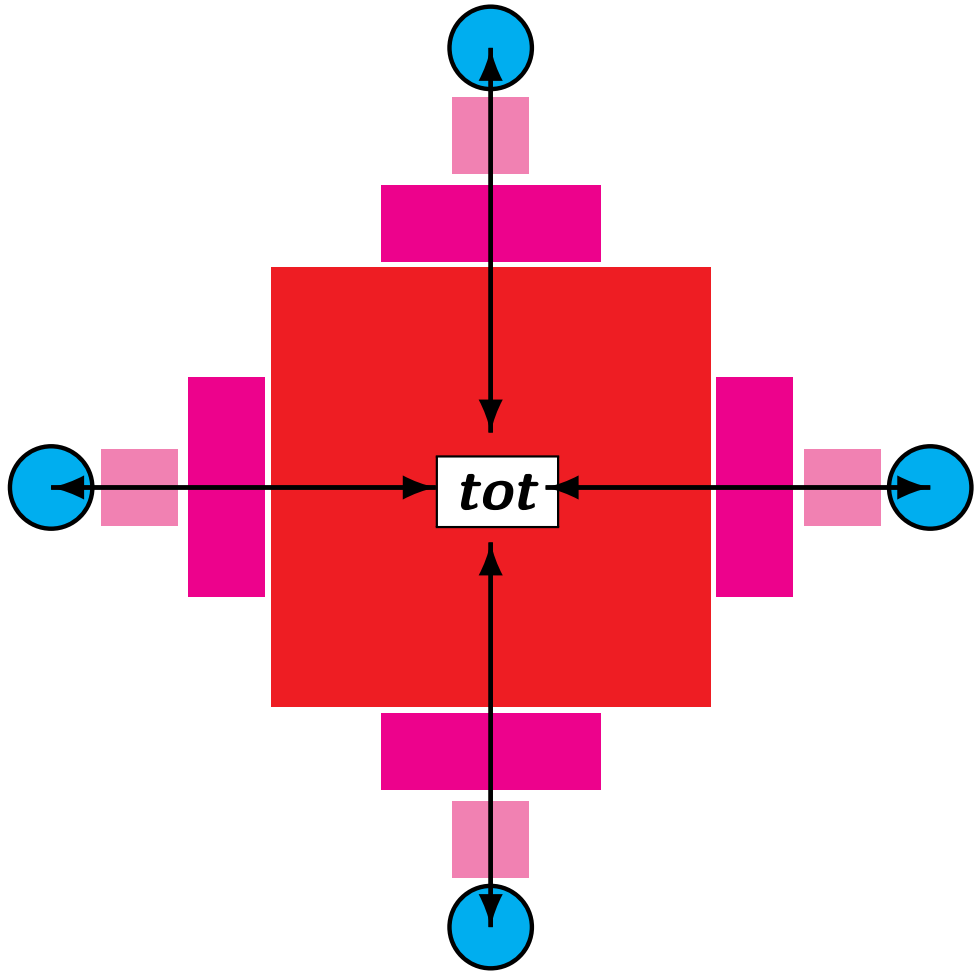
`tot` = 0

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 safe.
- `#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 [read|write|update|capture]`
- `!$omp atomic [read|write|update|capture]`

```
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 update
    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...

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

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 update
  tot += mytot;
}
```

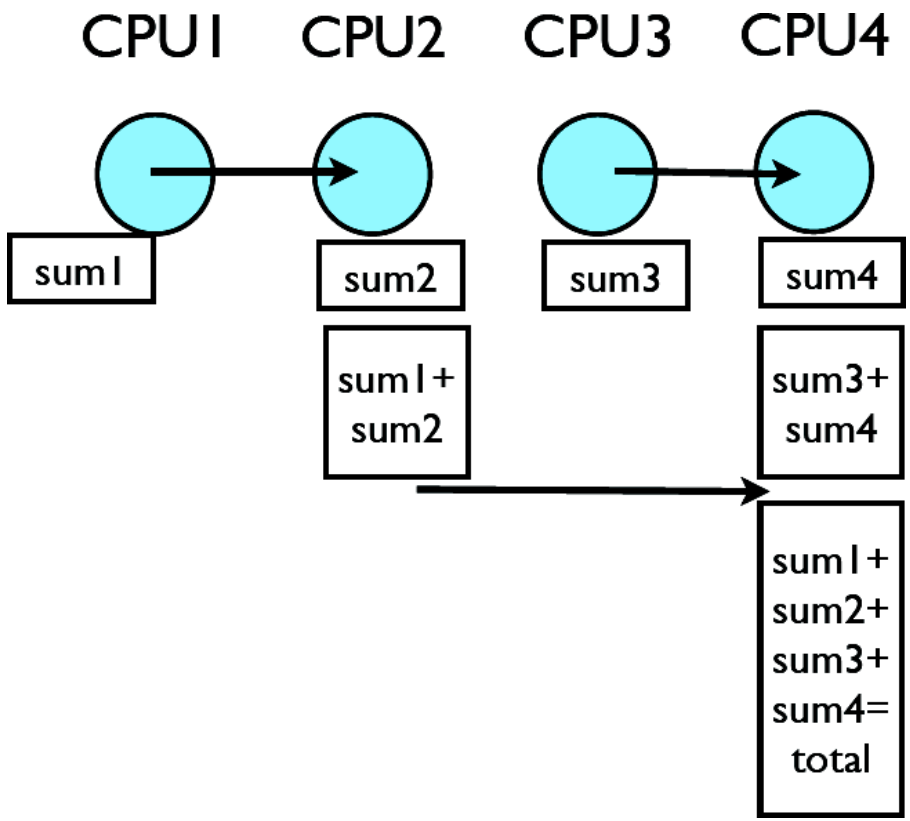
```
ndot = 0.
!$omp parallel shared(x,y,n,ndot) &
!$omp private(i,mytot)
mytot = 0.
!$omp do
do i=1,n
  mytot = mytot + x(i)*y(i)
enddo
!$omp atomic update
ndot = ndot + mytot
!$omp end parallel
```

```
$ 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.

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 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.

Load Balancing in OpenMP

SciNet compute • calcul CANADA

Load Balancing in OpenMP

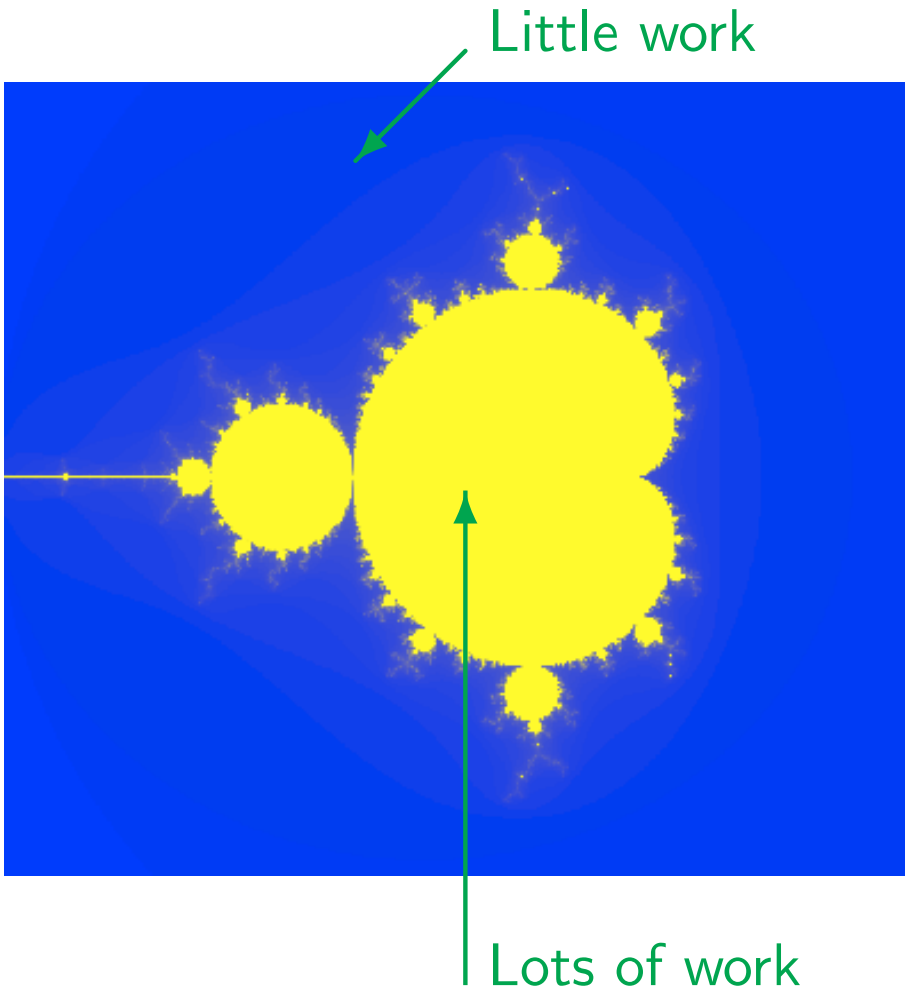
- So far every iteration of the loop had the same amount of work.
- Not always the case
- Sometimes cannot predict beforehand how unbalanced the problem is

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 a where $|b_\infty|$ finite, with $b_0 = 0$ and $b_{n+1} = b_n^2 + a$.
  If $|b_n| > 2$, 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



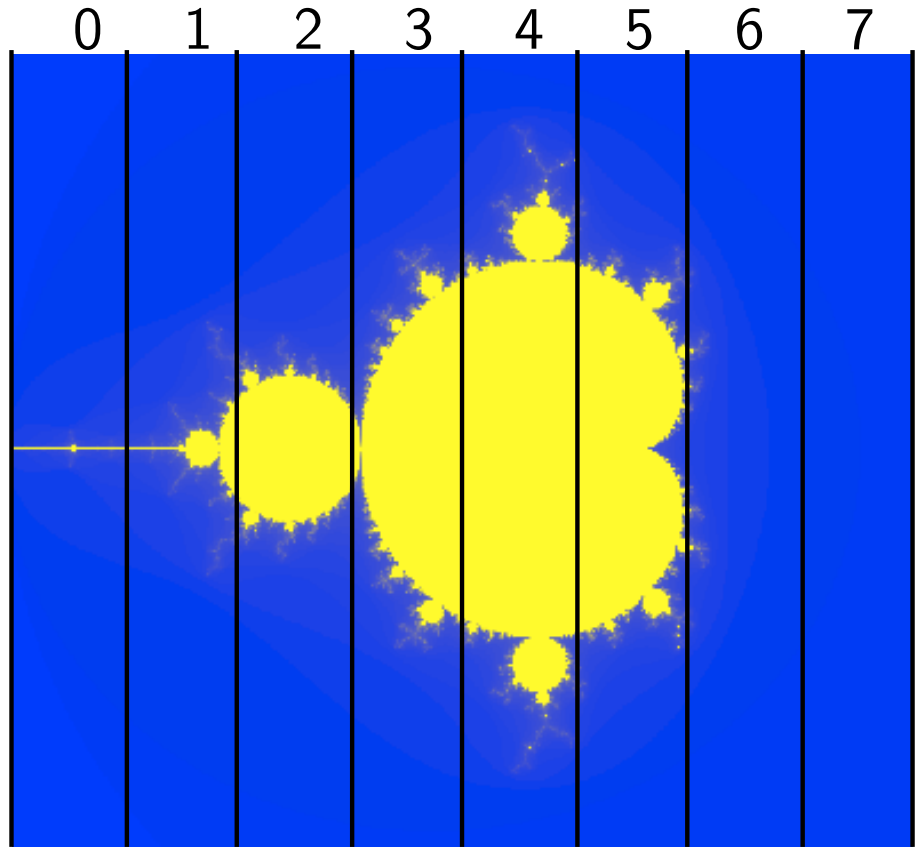
HANDS-ON 2:

Use OpenMP to parallelize the main loop in mandel.c or mandel-f.f90.

Perform scaling analysis.

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.

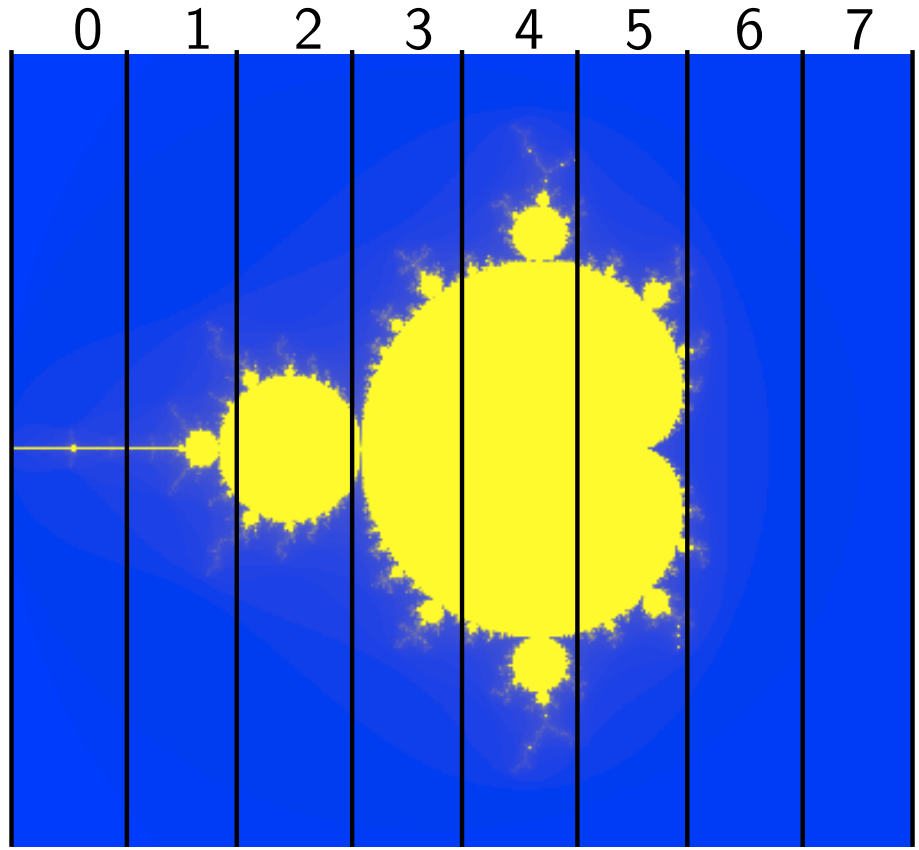


800x800 pix; N/nthreads $\sim$ 100x800

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

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.

HANDS-ON 3:

Use the OpenMP scheduling constructs to try and make `mandel` more efficient.

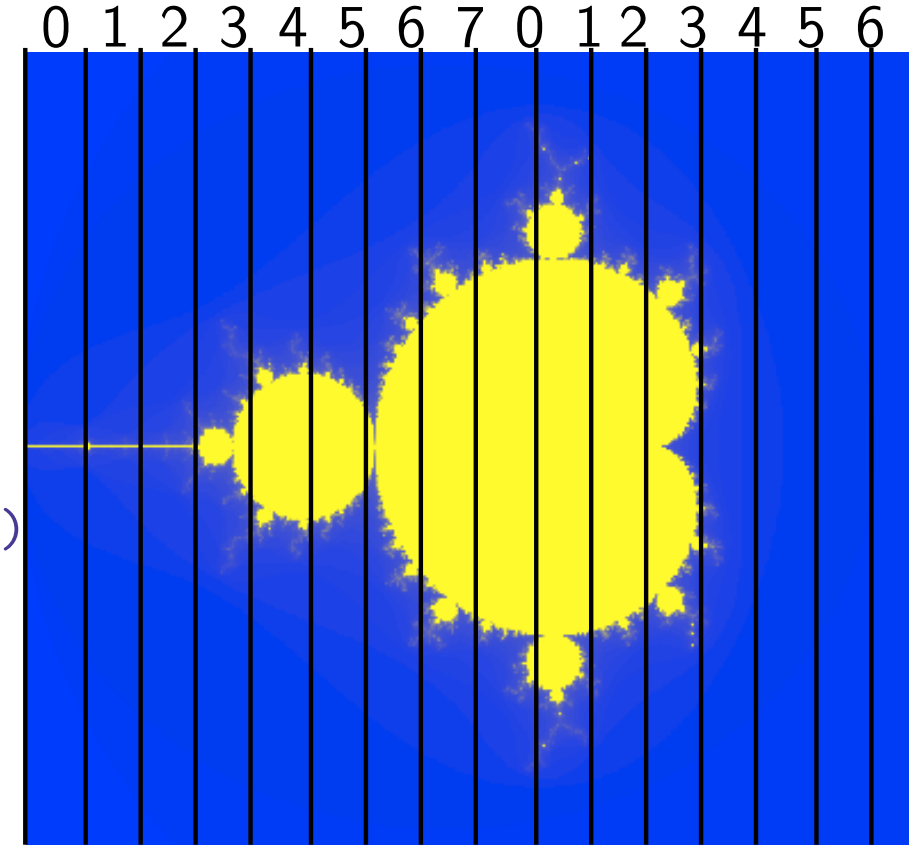
Second Try OpenMP Mandelbrot Set

- Can change the chunk size different from $\sim$ ***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

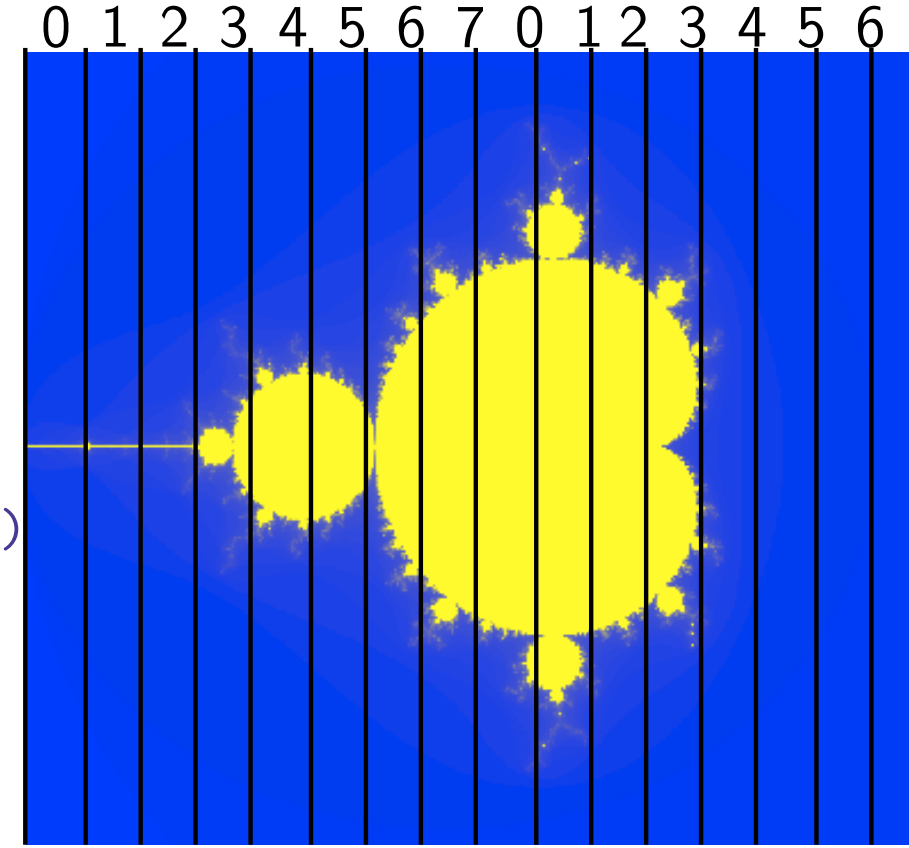
- Can change the chunk size different from $\sim$ ***N/nthreads***
- In this case, more columns – work distributed a bit better.
- Now, for instance, chunk size 50, and thread 7 gets both a big work chunk and a little one:

```
#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

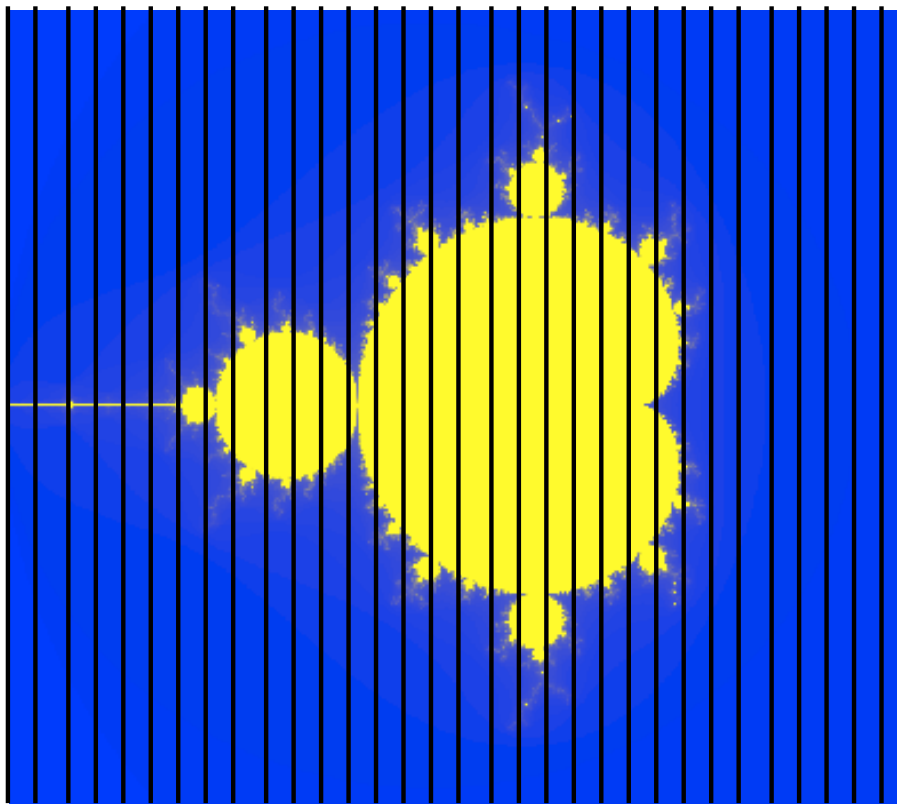
Third Try: Schedule dynamic

- 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

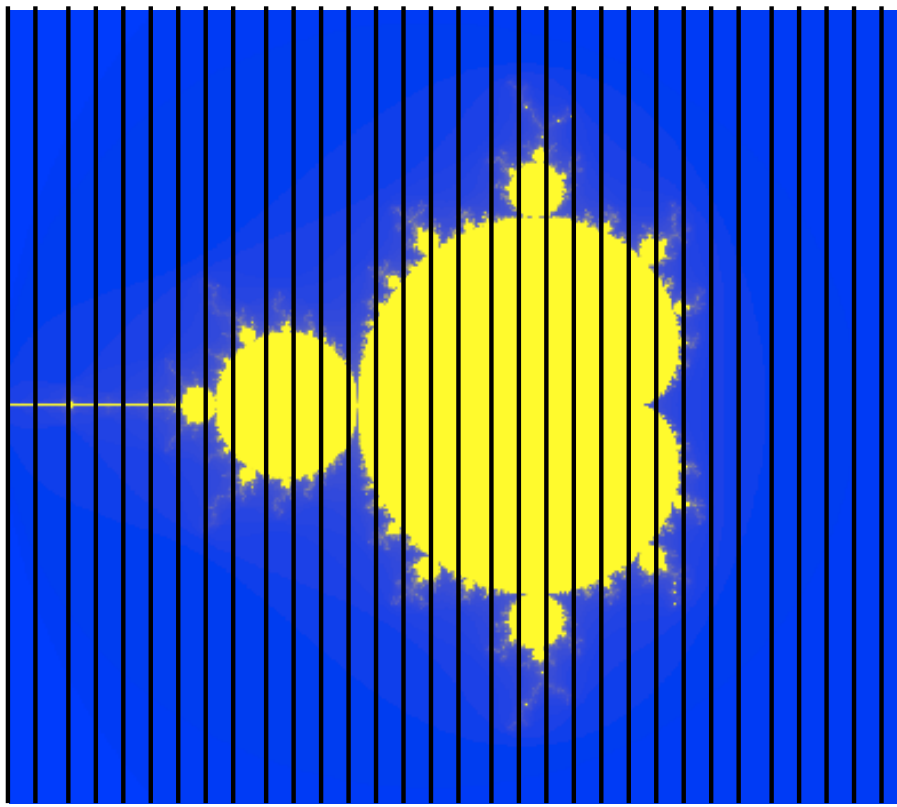
- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

```
#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 loops were all the work is.

Almost without exception, want the `pragma omp` on the outside-most loop.
Why?

You can also use
#pragma omp for collapse(n)
to schedule nested loops as one big loop.

```
#pragma omp for
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(a,maxiter);
  }
```

Two-level loops

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

Almost without exception, want the `pragma omp` on the outside-most loop.
Why?

You can also use
#pragma omp for collapse(n)
to schedule nested loops as one big loop.

```
#pragma omp for collapse(2)
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(a,maxiter);
  }
```

Summary

- 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 `[read|write|update|capture]`
- In a parallel region, have only one process do something:
  #pragma omp single

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.

Further OpenMP Features

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

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);
}
```

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)
}
```

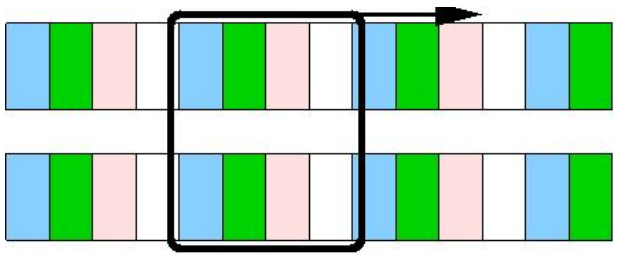
We have changed the # of threads during the program.

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.

Vectorization with OpenMP

Vectorization with OpenMP



- OpenMP can enable vectorization of both serial as well as parallelized loops.
- *vectorization* = processing multiple elements of an array at the same time.
- This is done using SIMD instructions.
- SIMD=single instruction multiple data. Usually 2, 4,or 8 *SIMD lanes* wide.
- Can also indicate to OpenMP to create versions of functions that can be invoked across SIMD lanes.

Note: This is new in OpenMP 4.0; supported by gcc v4.9.0 for c/c++ (not fortran), and by the intel compilers v14.

Directives for SIMD Support (new in OpenMP 4.0)

- `omp simd`
  marks a loop to be executed using SIMD lanes

- `omp declare simd`
  marks a function that can be called from a SIMD loop

- `omp parallel for simd`
  `omp parallel do simd`
  marks a loop for thread work-sharing as well as SIMDing

OpenMP SIMD Loop Example

```
#include <stdio.h>
#define N 65536
int main()
{
   double a[N], b[N];
   long long d1=0;
   double d2=0.0;
   #pragma omp simd reduction(+:d1)
   for (int i=0;i<N;i++)
      d1+=i*(N+1-i);
   #pragma omp simd
   for (int i=0; i<N;i++) {
      a[i]=i;
      b[i]=N+1-i;
   }
   #pragma omp parallel for simd reduction(+:d2)
   for (int i=0; i<N; i++)
      d2+=a[i]*b[i];
   printf("result1 = %lld\nresult2 = %.2lf\n", d1, d2);
}
```

OpenMP SIMD Loop Example - Fortran

```
program examplesimd1f
    implicit none
    integer, parameter :: N=65536
    double precision :: a(N), b(N), d2=0.0
    integer(kind=8) :: d1=0, i
!$omp simd reduction(+:d1)
    do i=1,N
      d1=d1+(i-1)*(N+1-(i-1))
    end do
!$omp simd
    do i=1,N
      a(i)=i-1
      b(i)=N+1-(i-1)
    end do
!$omp parallel do simd reduction(+:d2)
    do i=1,N
      d2=d2+a(i)*b(i)
    end do
    print *,"result1 =",d1; print *,"result2 =",d2
end program examplesimd1f
```

Nested OpenMP parallelism

Nested OpenMP parallelism

Could one create a parallel region from within a parallel region?
Yes... if implementation supports it, and you turn the capability on:

```
#include <stdio.h>
#include <omp.h>
int main()
{
  omp_set_nested(1);
  omp_set_dynamic(0);
  #pragma omp parallel num_threads(2)
  {
    int othread=omp_get_thread_num();
    #pragma omp parallel num_threads(3)
    {
      int ithread=omp_get_thread_num();
      printf("othread=%d, ithread=%d\n", othread, ithread);
    }
  }
}
```

Nested OpenMP parallelism

```
program examplenestf
    use omp_lib
    integer:: ithread, othread
    call omp_set_nested(.TRUE.)
    call omp_set_dynamic(.FALSE.)
!$omp parallel num_threads(2) private(othread)
       othread=omp_get_thread_num()
!$omp parallel num_threads(3) private(ithread)
          ithread=omp_get_thread_num();
          print *, "othread=", othread, "ithread=", ithread
!$omp end parallel
!$omp end parallel
end program examplenestf
```

- Instead of hardcoding the number of threads, can specify `OMP_NUM_THREADS=2,3`.
- Most useful when calling openmp-parallelized functions.
- For nested loops, collapse typically better.

Task-based parallelism

Non-loop constructs

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;
};
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);
```

Many task-related features in OpenMP

- Can abort parallel OpenMP execution by conditional cancellation at implicit and user-defined cancellation points.
- Tasks can be grouped to into task groups can be aborted to reflect completion of cooperative tasking activities such as search.
- Task-to-task synchronization is supported through the specification of task dependency.

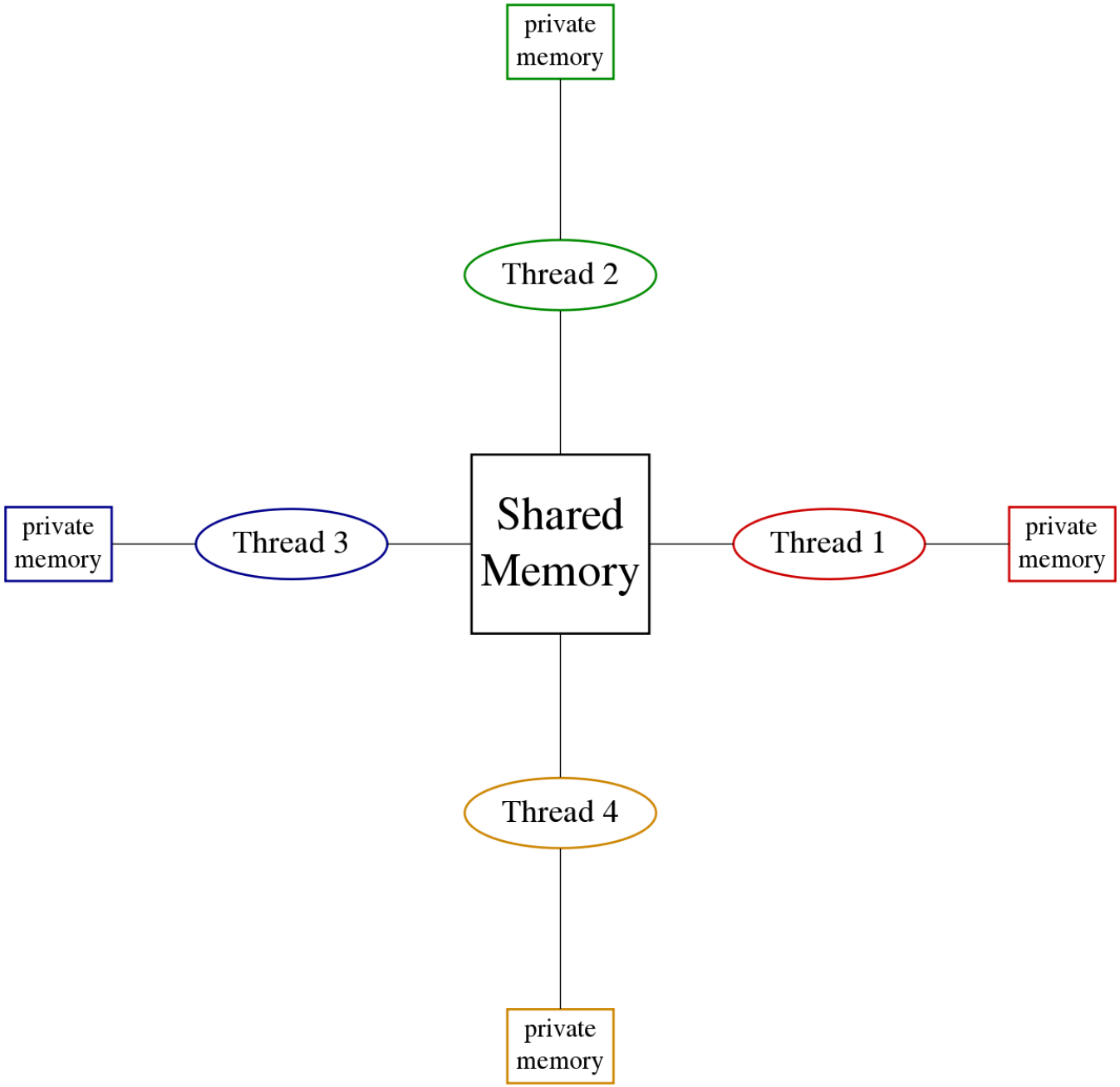
However, for many scientific codes the overhead of scheduling these tasks cancels any benefits; often, rewriting your computation to use a loop is more efficient.

Heterogeneous Computing with OpenMP

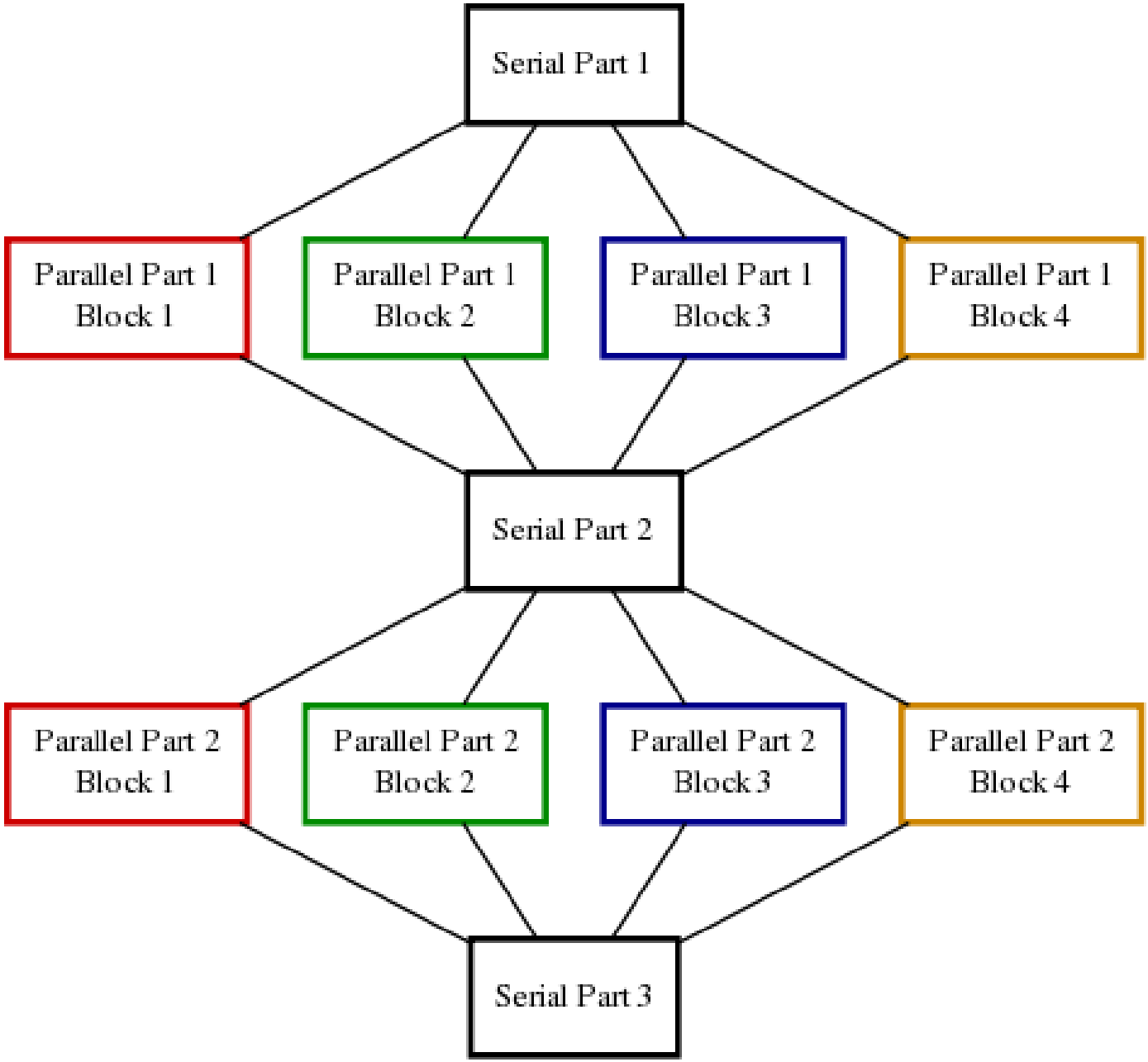
Heterogeneous Computing with OpenMP

First, let's recap the OpenMP model thus far.

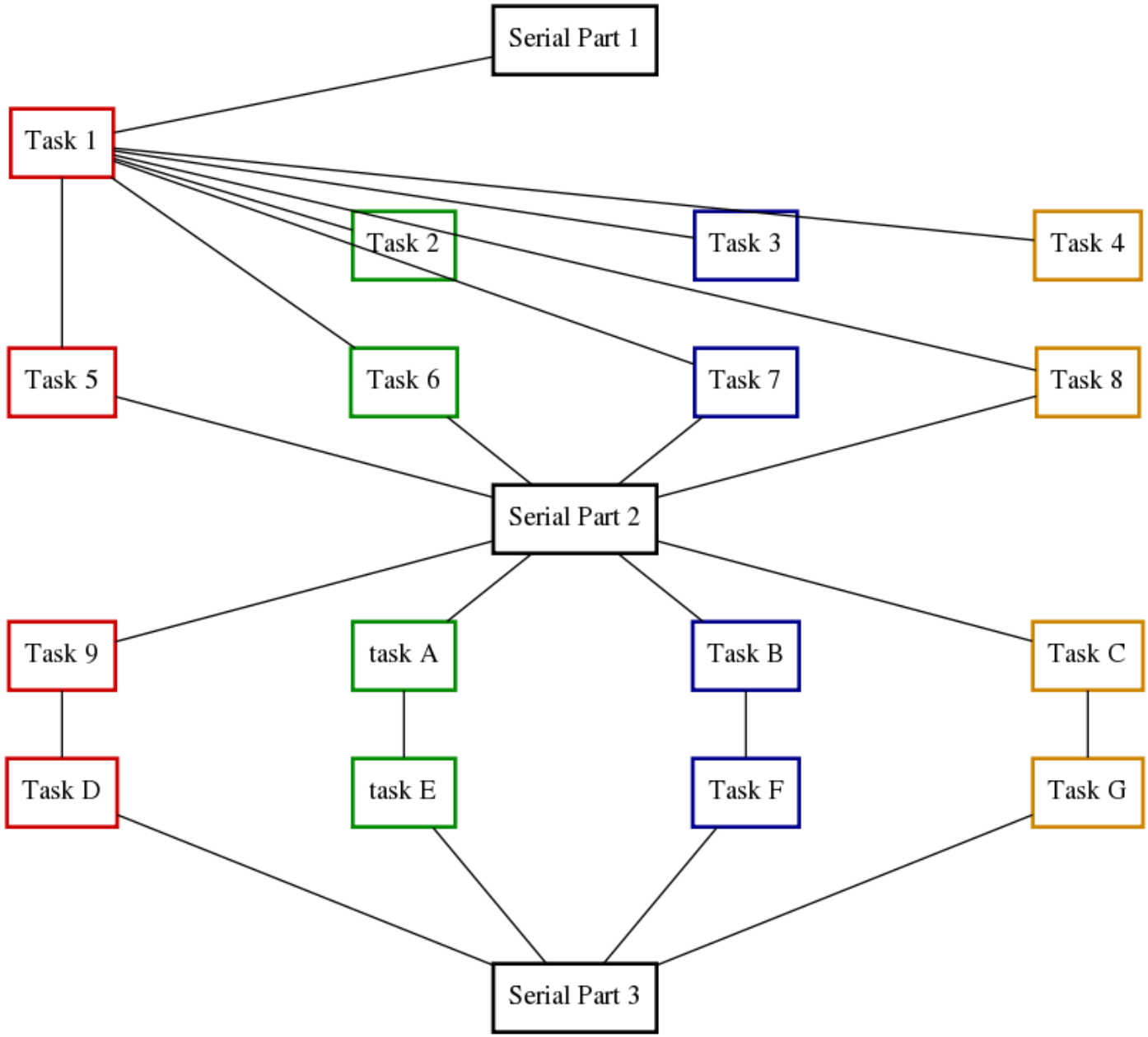
Memory Model in OpenMP (3.1)



Execution Model in OpenMP

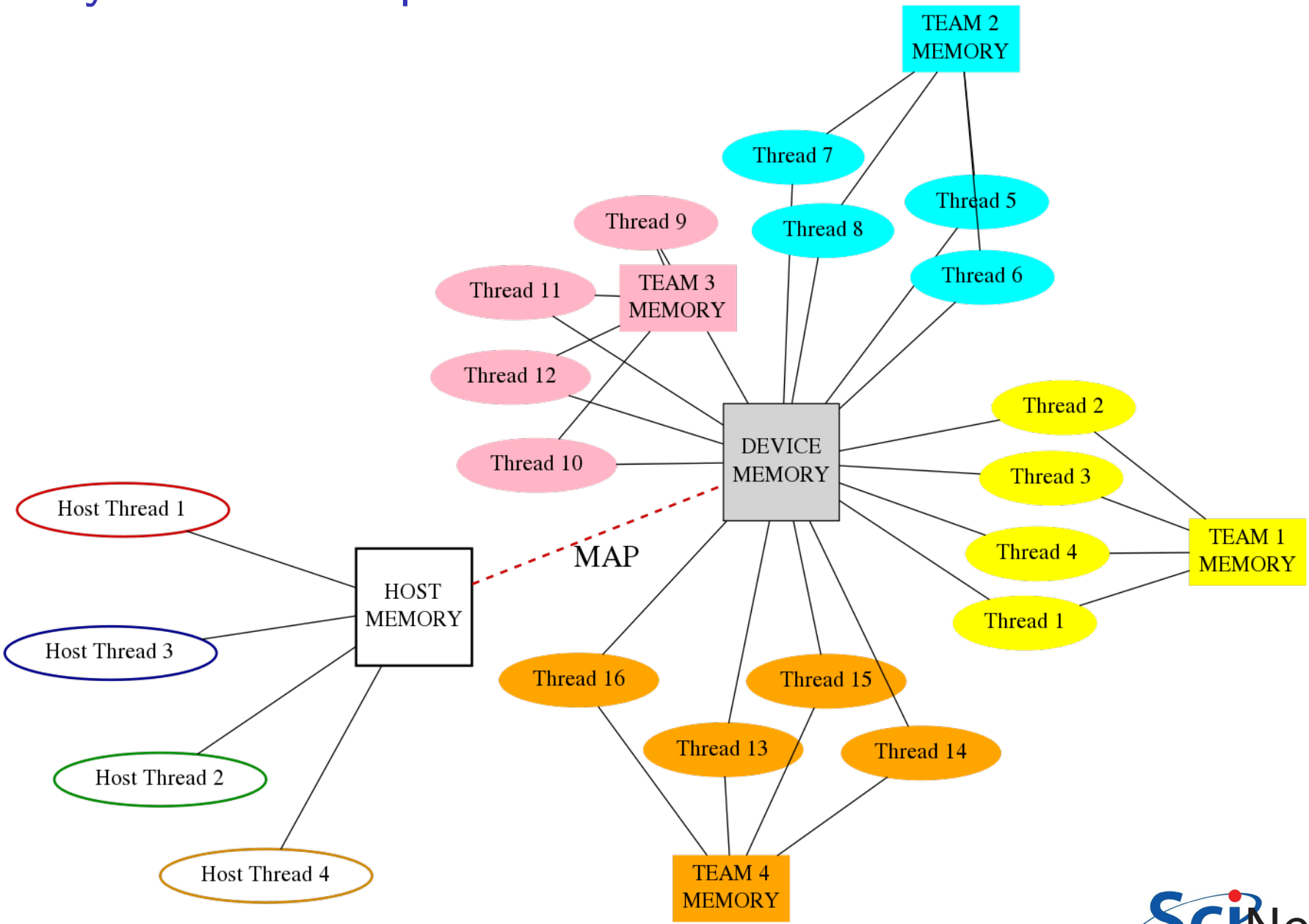


Execution Model in OpenMP with Tasks



Enter OpenMP 4.0

Memory Model in OpenMP 4.0



Memory Model in OpenMP 4.0

- Device has its own data environment
- And its own shared memory
- Threads can be bundled in a teams of threads
- These threads can have memory shared among threads of the same team
- Whether this is beneficial depends on the memory architecture of the device. (team $\approx$ CUDA thread blocks, MPI_COMM?)

Data mapping

- Host memory and device memory usually district.
- OpenMP 4.0 allows host and device memory to be shared.
- To accommodate both, the relation between variables on host and memory gets expressed as a *mapping*

  Different types:
  - `to`: existing host variables copied to a corresponding variable in the target before
  - `from`: target variables copied back to a corresponding variable in the host after
  - `tofrom`: Both `from` and `to`
  - `alloc`: Neither `from` nor `to`, but ensure the variable exists on the target but no relation to host variable.

  Note: arrays and array sections are supported.

OpenMP Device Example using `target`

```c
/* example2.c */
#include <stdio.h>
#include <omp.h>
int main()
{
  int host_threads, trgt_threads;
  host_threads = omp_get_max_threads();
  #pragma omp target map(from:trgt_threads)
  trgt_threads = omp_get_max_threads();
  printf("host_threads = %d\n", host_threads);
  printf("trgt_threads = %d\n", trgt_threads);
}
```

```
> $CC -fopenmp example2.c -o example2
> ./example2
host_threads = 16
trgt_threads = 224
```

OpenMP Device Example using `target`

```
program example2
  use omp_lib
  integer host_threads, trgt_threads
  host_threads = omp_get_max_threads()
  !$omp target map(from:target_threads)
  trgt_threads = omp_get_max_threads();
  !$omp end target
  print *, "host_threads =", host_threads
  print *, "trgt_threads =", trgt_threads
end program example2
```

```
> $FC -fopenmp example2.f90 -o example2
> ./example2
    host_threads = 16
    trgt_threads = 224
```

OpenMP Device Example using teams, distribute

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int ntprocs;
   #pragma omp target map(from:ntprocs)
   ntprocs = omp_get_num_procs();
   int ncases=2240, nteams=4, chunk=ntprocs*2;

   #pragma omp target
   #pragma omp teams num_teams(nteams) thread_limit(ntprocs/nteams)
   #pragma omp distribute
   for (int starti=0; starti<ncases; starti+=chunk)
      #pragma omp parallel for
      for (int i=starti; i<starti+chunk; i++)
         printf("case i=%d/%d by team=%d/%d thread=%d/%d\n",
                i+1, ncases,
                omp_get_team_num()+1, omp_get_num_teams(),
                omp_get_thread_num()+1, omp_get_num_threads());
}
```

OpenMP Device Example using `teams, distribute`

```
program example3
 use omp_lib
 integer i, ntprocs, ncases, nteams, chunk
 !$omp target map(from:ntprocs)
 ntprocs = omp_get_num_procs()
 !$omp end target
 ncases=2240
 nteams=4
 chunk=ntprocs*2
 !$omp target
 !$omp teams num_teams(nteams) thread_limit(ntprocs/nteams)
 !$omp distribute
 do starti=0,ncases,chunk
  !$omp parallel do
  do i=starti,starti+chunk
   print *,"i=",i,"team=",omp_get_team_num(),"thread=",omp_get_thread_nu
  end do
  !$omp end parallel
 end do
 !$omp end target
end program example3
```

Summary of New Directives and Functions for Devices

- `omp target [map]`
  marks a region to execute on device
- `omp teams`
  creates a league of thread teams
- `omp distribute`
  distributes a loop over the teams in the league
- `omp declare target` / `omp end declare target`
  marks function(s) that can be called on the device

- `omp_get_team_num()`
- `omp_get_team_size()`
- `omp_get_num_devices()`

BUT...

- This is only really implemented by the Intel compiler for their Intel Xeon Phi co-processors (see *intro to HPC*).
- You would think OpenMP 4.0 would be a good platform to program GPUs as well, but there is a slightly more apt contented, called OpenACC.

OpenACC

- Also a directive-based parallel programming framework.
- Fits more closely with GPUs: gangs-workers-vectors.
- More explicit about data copying to the accellerator (vs. mapping in OpenMP).
- A bit more mature than accellerators in OpenMp.
- Supported by the PGI compiler for NVidia cards.
- See CUDA session for more info.

Open standard != Open source

But Gcc is starting/planning to support OpenACC for GPUs and OpenMP for Xeon Phis.

Useful references

- Chapman, Jost, Van der Pas: *Using OpenMP*
  (2008, MIT Press)
- openmp.org/wp/openmp-specifications
  (Strongly recommended – many good sample programs)
- *SciNet Wiki:* wiki.scinethpc.ca: Tutorials & Manuals