Introduction to Practical Parallel Programming

Course Overview, and The ‘Big Picture’
Today’s Main Goal

• Students arriving with scientific computing background should be able to leave and immediately start parallelizing codes, understand concepts involved
## Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9am-10:30</td>
<td>Basic Concepts</td>
</tr>
<tr>
<td>10:30-10:45</td>
<td>Break</td>
</tr>
<tr>
<td>10:45-12:45</td>
<td>Intro to OpenMP</td>
</tr>
<tr>
<td>12:45-1:45</td>
<td>Lunch</td>
</tr>
<tr>
<td>1:45-3:30</td>
<td>Intro to MPI</td>
</tr>
<tr>
<td>3:30-3:45</td>
<td>Break</td>
</tr>
<tr>
<td>3:45-5:00</td>
<td>Intro to MPI 2</td>
</tr>
</tbody>
</table>
What will we be doing here

• This is a short course on parallel *programming*

• You will be doing a lot of typing and programming to help build skills with OpenMP, MPI.
Parallel Computing
I: Concurrency, Amdahl’s Law, and Locality
Why Parallel Computing?

*Faster:*  
At any given time, there is a limit as to **how fast** one computer can compute.  
So use more computers!
Why Parallel Computing?

Bigger:
At any given time, there is a limit as to how much memory, disk space, etc can be put on one computer.
So use more computers!
Why Parallel Computing?

*More:* You have a program that runs in reasonable time on one processor but you want to run it *thousands of times.* So use more computers!
Concurrency

• Must be something for the ‘more computers’ to do.
• Must be able to find concurrency in your problems
  • Many Tasks
  • Order Unimportant

http://flickr.com/photos/splorp/
Data Dependencies Limit Concurrency

\[ \sqrt{x^2 + y^2 + z^2} \]
**Parameter Study: Ideal case**

- Want to know all results as model parameter varies
- Can run serial code on up to as many processors as parameter sets
- ‘More’
Throughput = Tasks/Time

How long it takes to process the $N$ tasks you want done

$$\text{throughput} = \frac{N}{\text{time}}$$

For completely independent tasks, $P$ processors can increase throughput by factor $P$!
Scaling with $P$

How a problem **scales**: how throughput behaves as processor number increases. In this case, the throughput scales linearly with the number of processors.

This is the best case: ‘Perfect scaling’
Scaling with $P$

Another way to look at it: time it takes to get some fixed amount of work done

More usual (and more important!)

Perfect scaling: time to completion $\sim 1/P$

$P$ processors - $P$ times faster
Scaling with P

Another way to look at it: time it takes to get some fixed amount of work done

More usual (and more important!)

Perfect scaling: time to completion $\sim 1/P$

$P$ processors - $P$ times faster
Parameter Study: ‘Embarrassingly Parallel’

- **Scales** perfectly up to $P=N$
- Speedup = $P$:'linear scaling', ideal case.
Problems Differ in amount of Concurrency

Integrate (or some other simple processing) tabulated experimental data

Integration of different regions can be summed by each processor

But first need to get data to processor, then bring together all the sums
Parallel Portion: Perfectly Parallel (as long as there is enough work) \[ T \sim \frac{1}{P} \]
Serial Portion:
Sum has to be done; if done on one processor, just same as serial: $T \sim const$
Parallel Overhead:
Data has to be sent to appropriate processor, a cost of the parallel implementation

\[ T \text{ const (best case) or increasing fn of } P \]
Total Time: Serial + Parallel

Ignoring data-moving costs (for now):

\[ \text{time}(N, P) = \left\lceil \frac{N}{P} \right\rceil T_{\text{work}} + T_{\text{reduction}}(P) \]

Typically linear in P (sum)
Eventually, as problem becomes increasingly scaled up, serial term dominates
Timing of simple case

Ignore data transfer costs; say:
100 s in integration work
5 s in assembling the parts
How does this behave on many processors?
More processors per run don’t always help

Given timing data, how do we choose P to run on if we have N programs to run?

Ideal case, timing goes down 1/P - doesn’t matter

Serial part (5%!) becomes a bottleneck

Can improve **throughput** by running on fewer processors

**Note:**

\[
\begin{align*}
t(50) &= 7s \\
t(25) &= 9s
\end{align*}
\]

Can run 2 jobs on 25 procs each in about same time as one on 50!
Speedup: How much faster with P procs?

An important concept is the speedup of a given parallel implementation

$$\text{speedup}(P) = \frac{t(N, P = 1)}{t(N, P)}$$
Efficiency: Speedup should be $\sim P$

Related concept: Parallel Efficiency (compared to serial code)

\[
\text{Efficiency}(P) = \frac{t(N, P = 1)}{Pt(N, P = 1)} \cdot \frac{\text{speedup}(P)}{P}
\]
Amdahl’s Law

Any serial part of computation will eventually dominate. If serial fraction is $f$, even if parallel component goes to zero, speedup can only be $1/f$.

$$\text{time}(N, P) \sim \left(f + \frac{1-f}{P}\right)^{1/f}$$

$$\text{Speedup} = \frac{1}{\left(f + \frac{1-f}{P}\right)}$$

$$\lim_{P \to \infty} \text{Speedup} = \frac{1}{f}$$

$$\lim_{P \to \infty} \text{Efficiency} = 0$$
Amdahl’s Law

- Any serial part of computation will eventually dominate
- If serial fraction is $f$, even if parallel component goes to zero, speedup can only be $1/f$
Avoiding Amdahl

In some cases, may not matter. If will run in reasonable time on some small number of processor, asymptotic arguments may not matter.
Trying to Beat Amdahl, #1

Rewrite serial portions to take into account parallelism. Eg, many reductions can be done in parallel that will cost $\log_2(P)$ (not 1, but much better than serial = $P$...)

![Diagram showing partitioning and data flow with regions labeled 1 to 4 and an arrow to an answer circle.](image-url)
Big Lesson #1

Optimal **Serial** Algorithm for your problem may not be the P→1 limit of your optimal **Parallel** algorithm
Trying to Beat Amdahl, #2 - Upsize

Desktop problem isn’t a supercomputer problem!

Reason to run on big machines is size as well as speed

Amdahl’s law assumes constant size problem

More work; f goes down.

Gustafson’s law: any sufficiently large problem can be efficiently parallelized.
Weak Scaling

How does problem behave if you expand problem size as number of processors?

Strong Scaling - on how many processors can you efficiently run given problem

Weak Scaling - how large a problem can you efficiently run
More on Concurrency

Most problems are not pure concurrency

Some level of synchronization, exchange of information needed between tasks

This needs to be minimized

Increases Amdahl’s ‘f’

Are themselves costly
Concurrency

Makes possible lots of wasted time (‘load balancing’, about which more later)
Locality

Information needed by the task should be as local as possible.

When tasks do need to interact, best that those interactions be as local as possible, and with as few others as possible.

Communications cost lower

Fewer processes have are locked up during the necessary synchronization.
Big Lesson #2

Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.
Finding Concurrency

Identify tasks that can be done independently, order doesn’t matter

PDEs: parts of domain

N-body: particles (or interactions)
Maintaining Locality

Now have to lump the concurrent bits into tasks.
Choosing that re-aggregation can greatly affect locality.
Example: 1d integration

Integrate a 1d function with (say) Simpson’s rule, with \( N \) points.
Concurrency: can do each of the points independently, then sum.
Locality: have each do a chunk
Example: 1d integration

Each processor gets N/P points to do
Total compute time for one process:

\[ T_{\text{comp}} = \left( \frac{N}{P} \right) N_{SRC_{\text{comp}}} \]

Now how to do sums?
Example: 1d integration

Each processor sends partial sums to others, then all can do total
Each processor sends its result (P-1) times and receives (P-1) results

\[ T_{\text{comm}} = 2(P - 1)C_{\text{comm}} \]
Integration with parallel costs:

Can actually get worse with P!

Communication cost increases with P

\[ N = 10000, N_{\text{sr}}=4, \]
\[ \frac{C_{\text{comm}}}{C_{\text{comp}}} = 100 \]
Integration with parallel costs:

Can actually get worse with $P$!

Communication cost increases with $P$

\[ N = 10000, \ N_{\text{sr}}=4, \quad \frac{C_{\text{comm}}}{C_{\text{comp}}} = 100 \]
Integration with parallel costs:

Can actually get worse with $P$!
Communication cost increases with $P$

$N = 10000$, $N_s = 4$, $C_{comm}/C_{comp} = 100$
We want this to be (ideally) constant in $P$, or at least grow slowly; otherwise as we scale up, we spend more time sending messages than computing.

If $N_{SR} \sim 4$, $C_{comm} \sim 1000 \ C_{comp}$, $N = 10000$, then

$$T_{comm}/T_{comp} \sim 1.2 \text{ for } P=16$$

(Advanced: this even matters for serial computation, due to memory bandwidth limitations. “Arithmetic Intensity”)

$$\frac{T_{comm}}{T_{comp}} = \frac{2(P - 1)C_{comm}}{N_{SR}C_{comp}}$$

$$= \frac{2P(P - 1)}{N_{SR}C_{comp}}$$

$$\sim P^2$$
Better Summing

Pairs of processors; send partial sums
Max messages received $\log_2(P)$
Can repeat to send total back

$$T_{\text{comm}} = 2 \log_2(P) C_{\text{comm}}$$

Reduction; works for a variety of operators $(+,*\text{,min,max...})$
Speedup with reduction

Very good! Efficiency still falling off past 20 or so processors (But integrating 10,000 numbers...)
Speedup with reduction with 1,000,000 numbers...
Communication-to-Computation ratio

Much better!

As number of processors goes up, relative cost of communications goes up only logarithmically.

If $N_{SR} \sim 4$, $C_{comm} \sim 100 \ C_{comp}$, $N = 10000$, then

$T_{comm}/T_{comp} \sim 0.08$ for $P=16$

$$\frac{T_{comm}}{T_{comp}} = \frac{2 \log_2(P)C_{comm}}{\frac{N}{P} N_{SR} C_{comp}}$$

$$= \frac{2P \log_2(P)}{N} \frac{1}{N_{SR}} \frac{C_{comm}}{C_{comp}}$$

$$\sim P \log_2(P)$$
Parallel Computing

II: Parallel Computers
Top500.org:
List updated every 6 months of the worlds 500 largest supercomputers.

Info about architecture, ...

1 Petaflop ($10^{15}$ flop/s);
126,600 cores
Computer Architectures

How the computers work shape how best to program them.
Shared Memory vs Distributed Memory.
Vector computers...
Distributed Memory: Clusters

Simplest type of parallel computer to build

• Take existing powerful standalone computers

• And network them

http://flickr.com/photos/eurleif/
Each Node is Independent

Parallel code consists of programs running on separate computers, communicating with each other. *Could* be entirely different programs.
Each node has independent memory

Locally stores its own portion of problem
Whenever it needs information from another region, requests it from appropriate CPU
Usual model: ‘message passing’
Clusters + Message Passing

HW: Easy to build (harder to build well)
HW: Can build larger and larger clusters relatively easily
SW: Every communication has to be hand coded -- hard to program
<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>GigE</td>
<td>~10 µs</td>
<td>1 Gb/s</td>
</tr>
<tr>
<td></td>
<td>(10,000 ns)</td>
<td>(~60 ns/double)</td>
</tr>
<tr>
<td>Infiniband</td>
<td>~2 µs</td>
<td>2-10 Gb/s</td>
</tr>
<tr>
<td></td>
<td>(2,000 ns)</td>
<td>(~10 ns/double)</td>
</tr>
</tbody>
</table>

Processor speed: 1 FLOP ~ few ns or less
Shared Memory

One large bank of memory, different computing cores acting on it. All ‘see’ same data

Any coordination done through memory.

Could do like before, but why? Each core is assigned a thread of execution of a single program that acts on the data.
Thread Vs. Process

Processes: Independent tasks with their own memory, resources

Threads: Threads of execution within one process, ‘seeing’ the same memory, etc.
Shared Memory: NUMA

Complicating things: each core typically has some of its own memory
Non-Uniform Memory Access
Locality still matters
Cores have cache, too.
Keeping this memory coherent is extremely challenging
Coherency

The different levels of memory imply multiple copies of some regions.
Multiple cores mean can update unpredictably.
Very expensive hardware
Hard to scale up to lots of processors, very $$$
Very simple to program!!

\[ x[20] = 3 \]

\[ x[20] = ? \]
<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GigE</strong></td>
<td>~10 µs (10,000 ns)</td>
<td>1 Gb/s (~60 ns/double)</td>
</tr>
<tr>
<td><strong>Infiniband</strong></td>
<td>~2 µs (2,000 ns)</td>
<td>2-10 Gb/s (~10 ns/double)</td>
</tr>
<tr>
<td><strong>NUMA Shared Mem</strong></td>
<td>~0.1 µs (100 ns)</td>
<td>10-20 Gb/s (~4 ns/double)</td>
</tr>
</tbody>
</table>

Processor speed: 1 FLOP ~ ns or less
Big Lesson #3

The best approach to parallelizing your problem will depend on both details of your problem and of the hardware available.
Hybrid Architectures

Almost all of the biggest computers are now clusters of shared memory nodes.

Generally just use message passing across all cores, but as $P(1 \text{ node})$ goes up, hybrid approaches start to make sense.
Before we start with OpenMP:

- `cp -R ~/ljdursi/intro-ppp ~/`
- `source ~/intro-ppp/setup`
- `cd ~/intro-ppp/ gettingstarted/`
- `make omp_hello_world`
- `./omp_hello_world`

- `make mpi_hello_world`
- `mpirun -np 8 ./mpi_hello_world`

- `qsub -l -X` into your reserved node as per instruction sheet and ensure this works
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
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
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

```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;
}
```

F90: omp-hello-world.f90

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

```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
```
$ 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

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
.
At start of program
Hello, world, from thread 0!
export OMP_NUM_THREADS=32
.
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’
#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;
}

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
#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;
}

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)
#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;
}

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;
}
```

```
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
#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;
}

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

```bash
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
or
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp
```
```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 of %d!\n", omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}

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

```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());
    }
    printf("There were %d threads.\n", omp_get_num_threads());
    return 0;
}
```
```
#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;
}
```
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
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

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

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

```fortran
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
```
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.

```fortran
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
```
#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;
}

program omp_vars
use omp_lib
implicit none

integer :: nthreads

!$omp parallel default(none), shared(nthreads)
!$omp single
    nthreads = omp_get_num_threads()
    print *, 'Number of threads was ', nthreads, '.'
!$omp end single
!$omp end parallel

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
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)

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

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
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!

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

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

• \((a X + Y, \text{in double precision})\)

• Implement this, first serially, then with OpenMP

\[ \hat{z} = a\hat{x} + \hat{y} \]

• daxpy.c or daxpy.f90

• make daxpy or make fdaxpy
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

P=1

Get file1.c  Compile  Write file1.o  file2.c  Compile  file2.o

P=2

file2.c  Compile  Get file1.c  Compile  Write file1.o  file2.o
```c
#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;
}
```

Utilities for this course; NType is a numerical type which can be set to single 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);
    return 0;
}
```c
#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;
}
```
```c
#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;
}
```

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

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

```c
#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;
}
```
```c
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];
}
}
```

```c
!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$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. 3.69x speedup, 46% efficiency

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

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers 1.8619e-01 seconds. 1.86x speedup, 93% efficiency
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
- Implement this, first serially, then with OpenMP
- ndot.c or ndot.f90
- make ndot or make ndotf
- Tells time, answer, correct answer.

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

$ ./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?
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
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  3.33333283333717098E+020  (vs  3.33333363469873840E+020 )
for n = 10000000 . Took  5.00000007E-02  sec.
$ export OMP_NUM_THREADS=8
$ ./fomp_ndot_race
  Dot product is  6.06898061003712922E+019  (vs  3.33333363469873840E+020 )
for n = 10000000 . Took  0.16300000  sec.

Wrong answer - and slower!
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

\[
\begin{align*}
\text{Thread 0:} & \quad \text{add 1} \\
\text{Thread 1:} & \quad \text{add 2}
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{read n} \dot{\text{ot}} (=0) & \text{read n} \dot{\text{ot}} (=0) \\
\text{into register} & \text{into register} \\
\hline
\text{reg} = \text{reg} + 1 & \text{reg} = \text{reg} + 1 \\
\hline
\text{store reg} (=1) & \text{store reg} (=2) \\
\text{into nd} \dot{\text{ot}} & \text{into nd} \dot{\text{ot}} \\
\hline
\end{array}
\]

\[\text{nd} \dot{\text{ot}} = 0.\]

\[\text{nd} \dot{\text{ot}} = 2\]
Memory contention - why it’s slow

- Multiple cores repeatedly trying to read, access, store the 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 or !$omp critical / !$omp end critical

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

ndot = 0.
$omp parallel default(none) shared(ndot)
$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
```
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.
```
How should we fix this?

\[ n = \hat{x} \cdot \hat{y} = \sum_{i} x_i y_i \]
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)

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

- cp one of the omp_ndot.c’s or fomp_ndot.c’s to omp_ndot_local.c (or fomp_ndot_local.f90)
Local variables:

```c
#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;
}
```

```c
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
```

```
$ ./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.
```
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
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.

[Diagram showing a fractal pattern with areas marked as "Lots of work" and "Little work"]]
Load-Balancing

- Default work sharing breaks $N$ iterations into $\sim N/nthreads$ 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
## Load-Balancing

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial</strong></td>
<td>0.63s</td>
</tr>
<tr>
<td><strong>Nthreads=8</strong></td>
<td>0.29s</td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
<td>2.2x</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>27%</td>
</tr>
</tbody>
</table>

- **800x800 pix; N/nthreads ~ 100x800**
Load-Balancing

- Can change the `chunk size' from $\sim N/nthreads$ to arbitrary number.
- In this case, more columns - work distributed a bit better.
- Now, for instance, chunk size $\sim 50$, and thread 7 gets both a big work chunk and a little work chunk.
Load-Balancing

#pragma omp for schedule(static,chunksize)

or

!$omp do schedule(static,chunksize)

Here, *chunksize* = 50.

Static scheduling
schedule(static,50)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>0.63s</td>
</tr>
<tr>
<td>Nthreads=8</td>
<td>0.15s</td>
</tr>
<tr>
<td>Speedup</td>
<td>4.2x</td>
</tr>
<tr>
<td>Efficiency</td>
<td>52%</td>
</tr>
</tbody>
</table>
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
# schedule(dynamic)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>0.63s</td>
</tr>
<tr>
<td>Nthreads=8</td>
<td>0.10</td>
</tr>
<tr>
<td>Speedup</td>
<td>6.3x</td>
</tr>
<tr>
<td>Efficiency</td>
<td>79%</td>
</tr>
</tbody>
</table>
• 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

<table>
<thead>
<tr>
<th>(static,4)</th>
<th>(dynamic,16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.084s</td>
<td>0.099s</td>
</tr>
<tr>
<td>7.6x</td>
<td>6.4x</td>
</tr>
<tr>
<td>95%</td>
<td>80%</td>
</tr>
</tbody>
</table>
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?

```c
#pragma omp for schedule(static,4)
for (int i=0; i<npix; i++)
    for (int j=0; j<npix; j++) {
        double x=((double)i)/((double)npix)*(xmax-xmin)+xmin;
        double y=((double)j)/((double)npix)*(ymax-ymin)+ymin;
        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