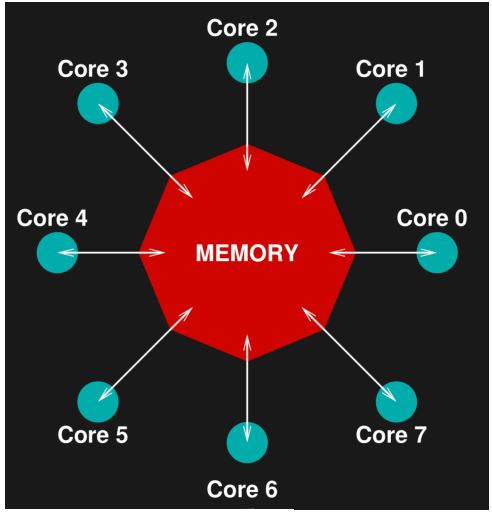
Introduction to High Performance Computing

Alejandro Cárdenas-Avendaño

OpenMP

Shared Memory

- One large blob of memory, different computing cores acting on it. All ‘see’ the 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

```
conda install -c intel openmp
brew install libomp
```

- Open Multi-Processing (OpenMP)
- For shared memory systems.
- Add parallelism to functioning serial code.
- http://openmp.org
- Compiler, run-time environment does a lot of work for us (divides up work)
- But we have to tell it how to use variables, where to run in parallel, . . .
- Works by adding **compiler directives** to code.

OpenMP basic operations

In code:

- In C++, you add lines starting with **#pragma omp**
  - This parallelizes the subsequent code block.

When compiling:

- To turn on OpenMP support in g++, add the **-fopenmp** flag to the compilation and link commands

When running:

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

OpenMP basic operations

```
#include <iostream>
#include <omp.h>
#include <string>

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

```
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program
Hello world from thread 0!
```

```
$ export OMP_NUM_THREADS=4
$ ./omp-hello-world
At start of program
Hello world from thread 2!
Hello world from thread 1!
Hello world from thread 0!
Hello world from thread 3!
```

```
$ g++-8 -O2 -o omp-hello-world omphelloworld.cpp -fopenmp
```

What happened precisely?

- Threads were launched.
- Each prints ‘Hello, world . . . ’
- In seemingly random order.

```
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program
Hello world from thread 2!
Hello world from thread 1!
Hello world from thread 4!
Hello world from thread 0!
Hello world from thread 3!
Hello world from thread 6!
Hello world from thread 5!
Hello world from thread 7!
```

Language extension + a library

- #pragma omp give the language extensions
- #include <omp.h> give access to library functions such as

```
int omp_get_num_threads();          // number of threads currently running
int omp_get_thread_num();           // index of the current threads (starts at 0)
void omp_set_num_threads(int n);    // number of threads to be used at the next parallel section
int omp_get_num_procs();            // get maximum number of processors
```

Example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {

  std::cout << "At start of program\n";

  #pragma omp parallel
  std::cout << "Hello world from thread "
     + std::to_string(omp_get_thread_num()) +
"!\n";

    std::cout << "There were "
     + std::to_string(omp_get_num_threads())
+ " threads.\n";
}
```

```
$ export OMP_NUM_THREADS=4
$ ./omp2
At start of program
Hello world from thread 0!
Hello world from thread 1!
Hello world from thread 3!
Hello world from thread 2!
There were 1 threads.
```

- Strange, says: ‘There were 1 threads.’. Why?
- Because that is true outside the parallel region!

```
$ g++-8 -O2 -o omp2 ompnumthreads.cpp -fopenmp
```

Variables to the rescue!

- **omp_get_num_threads** only returns the number of threads in a parallel region inside said region.
- Let's try to store the result of **omp_get_num_threads** to a variable then.

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

int main() {

  int t, nthreads;

  #pragma omp parallel default(none) shared(nthreads) private(t)
  {
    t = omp_get_thread_num();
    if (t == 0)
      nthreads = omp_get_num_threads();
  }
  std::cout<<"There were "<<nthreads<<" threads.\n";
}
```

```
$ export OMP_NUM_THREADS=4
$ ./omp3
There were 4 threads.
```

- What are these extra clauses?
  - **shared**: read/write access to the variable for each thread
  - **private**: separate instance of the variable for each thread

```
$ g++-8 -O2 -o omp3 ompvariables.cpp -fopenmp
```

Shared and Private Variables

Shared Variables:

- A variable designated as shared can be accessed by all threads.
- For reading variable values, this is very convenient.
- For assigning to variables, this introduces potential **race conditions**.

Private Variables:

- If a variable is designated as private, each thread gets its own separate version of the variable.
- Different threads **cannot** see other threads' versions.
- Thread-private versions do not have the value of the variable outside the parallel loop.
- The thread-private versions **cease to exists** after the parallel region.

If a variable is not designated as either **shared** or **private**, the compiler chooses:

- That may seem like a nice feature, but try not to rely on this!
- With default(none), compilation fails if undesignated variables are used in parallel regions.

Variables to the rescue!

- We can also declare a local declaration will became private

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

int main() {

  int nthreads;

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

Single Execution

- We do not care which thread sets nthreads.
- Might as well be the first thread that gets to it.
- OpenMP has a construct for this:

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

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

```
$ export OMP_NUM_THREADS=4
$ ./omp4
There were 4 threads.
```

```
$ g++-8 -O2 -o omp4 ompsingle.cpp -fopenmp
```

Loops in OpenMP

- Lots of loops in scientific code. Let's add a senseless loop:

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  int i, t;
  #pragma omp parallel default(none) private(i,t) shared(std::cout)
  {
    t = omp_get_thread_num();
    for (i=0; i<4; i++)
    std::cout << "Thread " + std::to_string(t)
              + " gets i=" + std::to_string(i) + "\n";
  }
}
```

```
$ g++-8 -O2 -o omploop omploop.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploop
```

```
Thread 1 gets i=0
Thread 0 gets i=0
Thread 1 gets i=1
Thread 1 gets i=2
Thread 1 gets i=3
Thread 0 gets i=1
Thread 0 gets i=2
Thread 0 gets i=3
```

Loops in OpenMP

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  int i, t;
  #pragma omp parallel default(none) private(i,t) shared(std::cout)
  {
    t = omp_get_thread_num();
    for (i=0; i<4; i++)
    std::cout << "Thread " + std::to_string(t)
              + " gets i=" + std::to_string(i) + "\n";
  }
}
```

```
Thread 1 gets i=0
Thread 0 gets i=0
Thread 1 gets i=1
Thread 1 gets i=2
Thread 1 gets i=3
Thread 0 gets i=1
Thread 0 gets i=2
Thread 0 gets i=3
```

- Every thread executes all 4 cases!
- Probably not what we want.

Worksharing in OpenMP

- We don't generally want tasks to do exactly the same thing.
- Want to divide a problem into pieces that threads works on.
- OpenMP has a worksharing construct: omp for.

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  int i, t;
  #pragma omp parallel default(none) private(i,t) shared(std::cout)
  {
    t = omp_get_thread_num();
  #pragma omp for
    for (i=0; i<8; i++)
    std::cout << "Thread " + std::to_string(t)
              + " gets i=" + std::to_string(i) + "\n";
  }
}
```

```
Thread 0 gets i=0
Thread 1 gets i=4
Thread 0 gets i=1
Thread 0 gets i=2
Thread 0 gets i=3
Thread 1 gets i=5
Thread 1 gets i=6
Thread 1 gets i=7
```

```
$ g++-8 -O2 -o omploop2 omploop2.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploop2
```

Worksharing constructs in OpenMP

```
Thread 0 gets i=0
Thread 1 gets i=4
Thread 0 gets i=1
Thread 0 gets i=2
Thread 0 gets i=3
Thread 1 gets i=5
Thread 1 gets i=6
Thread 1 gets i=7
```

- omp for construct breaks up the iterations by thread.
- If doesn't divide evenly, does the best it can.
- Allows easy breaking up of work!
- Code need not know how many threads there are; OpenMP does the work division for you.

Parallelizing the loops

Things to consider when parallelizing:

- Where is the concurrency?

  I.e. what loops have independent iterations, so they may be done in parallel?
- If we divide the work over threads, which variables do the threads need to know about?
- Which ones are shared, which ones are to be private?

For your convenience:

- Constants are forced to be automatically shared
- **#pragma omp parallel** and **#pragma omp for** may be combined to

  #pragma omp parallel for

Example

- A very weird calculation…
- Start from a serial implementation, then will add OpenMP

$$result = \sum_{ij}^{n} \sqrt{a\left[i\right]^2 + b\left[i\right]^2}$$

$$a\left[i\right] = \log\left(i+1\right)\left[\cos\left(i\right)\tan\left(j\right)\right]^2 \qquad b\left[i\right] = \sqrt{j}\left[\sin\left(i\right)\tan\left(j\right)\right]^2$$

Example-Serial

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;

  for (i=0; i < n; i++)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o loopvec loopvec.cpp
$ ./loopvec
Final result= 128235433739.279927
real   0m2.387s
```

Example-OpenMP V1

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  #pragma omp parallel for
  for (i=0; i < n; i++)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec
omploopvec.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec
Final result= 122428728343.368360
real   0m1.266s
```

```
Serial One:
real   0m2.387s
```

Example-OpenMP V2

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec2
omploopvec2.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec2
Final result= nan
real   0m1.773s
```

```
Serial One:
real   0m2.387s
```

Example-Wait… What?

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;

  for (i=0; i < n; i++)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

OpenMP V1:
Final result= ***122428728343.368360***
real ***0m1.266s***

Serial One:
Final result= ***128235433739.279927***
real ***0m2.387s***

OpenMP V2:
Final result= ***nan***
real ***0m1.773s***

Our very first race condition!

- 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.
- Classical parallel bug.
- Multiple writers to some shared resource.

Race Condition Example

Say, initially, **tot=0**, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for **tot** is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

Non-atomic adding and updating:

Thread 0: add 1	**Thread 1: add 2**
read tot=0 to reg0	.
reg0 = reg0+1	read tot=0 to reg1
store reg0(=1) in tot	reg1 = reg1 + 2
.	store reg1(=2) in tot

Example-OpenMP V3

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for default(none) private(j,a,b) shared (i,n,result)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec3
omploopvec3.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec3
Final result= 125157708304.282732
real   0m1.545s
```

```
Serial One:
Final result= 128235433739.279927
real   0m2.387s
```

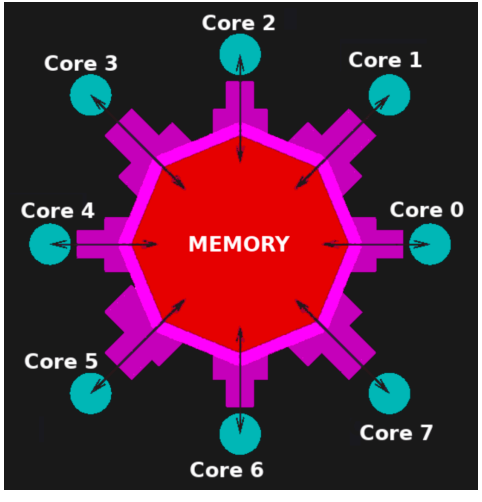
```
$ export OMP_NUM_THREADS=4
$ ./omploopvec3
Final result= 98607149253.141238
real   0m1.435s
```

So it’s wrong, but why is it slower?

You might thing the parallel version should at least still be faster, though it may be wrong. But even that’s not the case.

- Here, multiple cores repeatedly try to read, access and store the same variable in memory.
- This means the **shared variable** that is updated in a register, cannot stay in register: It has to be copied back to main memory, so the other threads see it correctly.
- The other threads then have to re-read the variable.
- This write-back would not be necessary if the variable was shared but not written to.

Memory hierarchy



- Memory is layered: between registers and shared main memory there are further layers called **caches**.
- Caches are faster but more expensive and therefore smaller. They are like private memory for each core.
- Main memory is the slowest part of the memory.
- Caches are automatically kept coherent between cores.

OpenMP critical construct

Our code get it wrong because different threads are updating the tot variable at the same time.

The **critical construct**:

- Defines a critical region.
- Only one thread can be operating within this region at a time.
- Keeps modifications to shared resources safe.

Example-OpenMP V4

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for default(none) private(j,a,b) shared (i,n,result)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          #pragma omp critical
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec4
omploopvec4.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec4
Final result= 128235433739.279927
real   0m5.568s
```

```
Serial One:
Final result= 128235433739.279927
real   0m2.387s
```

Reductions

Example-OpenMP V5

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for default(none) private(j,a,b) shared (i,n) reduction(+:result)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec5
omploopvec5.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec5
Final result= 128235433739.279927
real   0m1.406s
```

```
Serial One:
Final result= 128235433739.279927
real   0m2.387s
```

```
$ export OMP_NUM_THREADS=4
$ ./omploopvec5
Final result= 128235433739.279937
real   0m0.940s
```

```
$ export OMP_NUM_THREADS=1
$ ./omploopvec5
Final result= 128235433739.279937
real   0m2.626s
```

References

- SciNet Education (https://support.scinet.utoronto.ca/education/help/help_about.php)
- International HPC Summer School 2018 Lectures
- ARCHER UK National Supercomputing Service Training Courses (http://www.archer.ac.uk/training/)
- Muna, D & Price-Whelan, A. SciCoder Workshop. scicoder.org
- Peter Norvig, “What to demand from a Scientific Computing Language”, Mathematical Sciences Research Institute, 2010.