High Performance Scientific Computing with OpenMP, part 2

Ramses van Zon

PHY1610 Winter 2024

Loops

Loops in OpenMP

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

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
 #pragma omp parallel default(none) shared(std::cout)
 {
  int t = omp_get_thread_num();
  for (int i=0; i<16; i++)
   std::cout << "Thread " + std::to_string(t)
             + " gets i=" + std::to_string(i) + "\n";
 }
}
```

What would you expect this to do with e.g. 2 threads?

This is what it does:

```
$ make omp-loop1
$ export OMP_NUM_THREADS=2
$ ./omp-loop1
Thread 0 gets i=0
Thread 0 gets i=1
Thread 0 gets i=2
Thread 1 gets i=0
Thread 0 gets i=3
Thread 1 gets i=1
Thread 0 gets i=4
Thread 1 gets i=2
Thread 0 gets i=5
Thread 1 gets i=3
Thread 0 gets i=6
Thread 1 gets i=4
Thread 0 gets i=7
Thread 1 gets i=5
Thread 0 gets i=8
Thread 1 gets i=6
Thread 0 gets i=9
Thread 1 gets i=7
Thread 0 gets i=10
Thread 1 gets i=8
Thread 0 gets i=11
```

- Every thread executes all 16 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() {
  #pragma omp parallel default(none) shared(std::cout)
  {
   int t = omp_get_thread_num();
   #pragma omp for
   for (int i=0; i<16; i++)
     std::cout << "Thread " + std::to_string(t)
               + " gets i=" + std::to_string(i) + "\n";
   }
}
```

Worksharing constructs in OpenMP

- 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.

```
$ make omp_loop2
$ export OMP_NUM_THREADS=2
$ ./omp_loop2
Thread 0 gets i=0
Thread 0 gets i=1
Thread 0 gets i=2
Thread 1 gets i=8
Thread 0 gets i=3
Thread 1 gets i=9
Thread 0 gets i=4
Thread 0 gets i=5
Thread 0 gets i=6
Thread 0 gets i=7
Thread 1 gets i=10
Thread 1 gets i=11
Thread 1 gets i=12
Thread 1 gets i=13
Thread 1 gets i=14
Thread 1 gets i=15
```

Less trivial example: DAXPY

```
#include <rarray>
#include "ticktock.h"

void init(rvector<double>& x, rvector<double>& y, rvector<double>& z);

void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z);

int main()
{
   int n = 10'000'1000;
   rvector<double> x(n), y(n), z(n);
   double a = 5./3.;
   TickTock tt;
   tt.tick();
   init(x,y,z);
   mydaxpy(a,x,y,z);
   tt.tock("Tock registers");
}
```

DAXPY - Function definitions

```
#include <algorithm>

// Initialize arrays x and y with i^2 and i^2-1, respectively
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++) {
      x[i] = double(i)*double(i);
      y[i] = double(i+1)*double(i-1);
      z[i] = 0.0;
   }
}
// Add a*x+y to z.  x, y, and z are arrays and a is a scalar.
void mydaxpy(double a, const rvector<double>& x,
           const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++)
      z[i] += a * x[i] + y[i];
}
```

How would you OpenMP-parallelize this?

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`?

Parallel DAXPY

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(x,y,z,n)
   {
      #pragma omp for
      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }
  }
}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(x,y,a,z,n)
   {
      #pragma omp for
      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
   }
}
```

For your convenience

short-hand/combined pragmas

```
#pragma omp parallel
```

and

```
#pragma omp for
```

may be combined to

```
#pragma omp parallel for
```

Also note that instead of a code block with curly braces, a single line or a single loop with a single lines can be a parallel region.

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(n,x,y)
   {
      #pragma omp for
      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }
  }
}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(n,x,y,a,z)
   {
      #pragma omp for
      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
   }
}
```

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y)


      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }

}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y,a,z)


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

}
```

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y)
   for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
   }
}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y,a,z)
   for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
}
```

Parallel DAXPY performance

```
$ debugjob --clean -n 16  # if on the Teach cluster
$ module load gcc/13
$ make mydaxpy
$ ./mydaxpy
Tock registers 0.3936 sec
$ make mydaxpy-parallel
$ export OMP_NUM_THREADS=16
$ ./mydaxpy-parallel
Tock registers 0.07156 sec
```

5.5 times faster!

Submitting OpenMP jobs to the scheduler

- OpenMP uses shared memory, so you need to stay on one node.
- The application is a single process, so one task.
- That application needs multiple CPUs for its threads.
- But that application still needs to be told how many threads openmp should use.
- You probably want to know how long it took.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
#SBATCH --time=1:00:00
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/13

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

time ./mydaxpy-parallel
```

Reductions

Dot Product

- Dot product of two vectors

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

- Start from a serial implementation, then will add OpenMP

- Program tells answer, correct answer, time.

Dot Product Code

```
// ndot_main.cc
#include <iostream>
#include <rarray>
#include "ticktock.h"
double ndot(const rvector<double>& x,
            const rvector<double>& y);
int main()
{
   int n = 20'000'000;
   rvector<double> x(n), y(n);
   for (int i=0; i<n; i++)
       x[i]=y[i]=i;
   double nn  = n;
   double ans = (nn-1)*nn*(2*nn-1)/6;
   TickTock tt;
   tt.tick();
   double dot = ndot(x,y);
   std::cout << "Dot product: " << dot << "\n"
             << "Exact answer: " << ans << "\n";
   tt.tock("Took");
}
```

```
// serial_ndot.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   for (int i=0; i<n; i++)
      tot += x[i] * y[i];
   return tot;
}
```

```
$ make serial_ndot
$ ./serial_ndot
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took    0.1055 sec
$
```

Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make tot shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_race
Dot product:  2.64925e+20
Exact answer: 2.66667e+21
Took    0.5431 sec
$ ./omp_ndot_race
Dot product:  2.62621e+20
Exact answer: 2.66667e+21
Took    0.5383 sec
```

Wrong answer!

Answer varies!

Slower computation!

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.

How does this issue arise?

Non-atomic adding and updating

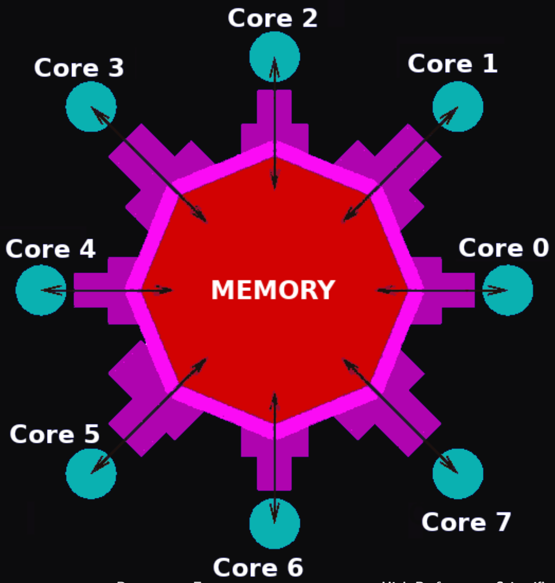
Thread 0: add 1	Thread 1: add 2
read tot=0 to reg0	.
reg0 = reg0+1	read tot=0 to reg1
store reg0(=1) in tot	reg1 = reg1 + 2
.	store reg1(=2) in tot

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.

Fixing the race condition

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.

```
// omp_ndot_critical.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp critical
      tot += x[i] * y[i];
   return tot;
}
```

Critical Construct Timing

```
// omp_ndot_critical.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,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=16
$ ./omp_ndot_critical
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 4.6697 sec
```

Correct, but 44× slower than serial version!

OpenMP atomic construct

- Most hardware has support for atomic instructions (indivisible so cannot get interrupted)
- Small subset, but load/add/store usually in it.
- Not as general as critical
- Much lower overhead.
- `#pragma omp atomic [read|write|update|capture]`

```
// omp_ndot_atomic.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp atomic update
      tot += x[i] * y[i];
   return tot;
}
```

Atomic Construct Timing

```
// omp_ndot_atomic.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,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=16
$ ./omp_ndot_atomic
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 2.177 sec
```

About twice faster than critical, but still not great.

Local Sums

The issue we have not resolved is that we're still updating tot, which causes copies to main memory at every iteration.

What if we accumulated tot for each core, and sum them up later?

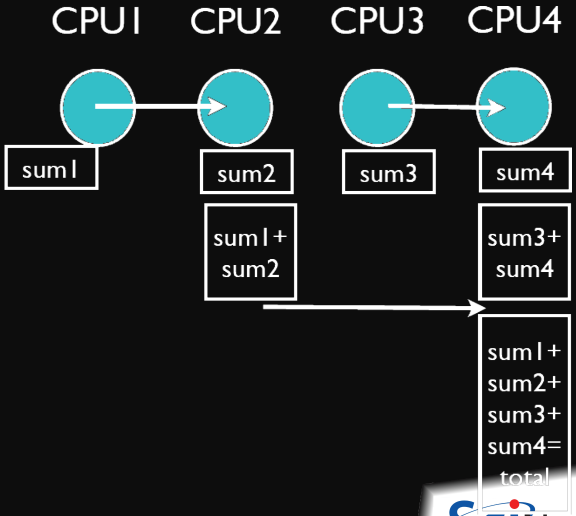
```
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel default(none) shared(n,tot,x,y)
 {
   double localtot=0;
   #pragma omp for
   for (int i=0; i<n; i++)
     localtot += x[i] * y[i];
   #pragma omp atomic update
   tot += localtot;
 }
 return tot;
}
```

```
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_local
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 0.01715 sec
```

Correct answer, 6x faster!

OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.
- When declaring a variables as reduction variables, private copies are made (much as for private variables), which are combined at the end of a parallel region through some operation (+, *, min, max).
- `omp_ndot_reduction.cc`



Reduction Timing

```
// omp_ndot_reduction.cc
#include <rarray>
#include <algorithm>
double ndot(const rvector<double>& x,
            const rvector<double>& y)
{
  int n = std::min(x.size(), y.size());
  double tot=0;
  #pragma omp for default(none) shared(n,x,y) reduction(+:tot)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_reduction
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_reduction
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 0.01691 sec
$
```

Correct, same timing as local sums, but simpler code.

Load Balancing

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 parallel 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.

More. . .

There are many more features to OpenMP we have not discussed.

- Collapsed loops
- Tasks
- Tasks with dependencies
- Nested OpenMP parallelism
- Locks
- SIMD
- Thread affinities
- Compute devices (e.g. NVIDIA/AMD graphics cards, Intel Xeon Phi)