High Performance Scientific Computing with OpenMP, part 2

Ramses van Zon

PHY1610 Winter 2026

Parallel DAXPY performance

```
$ debugjob -n 40 # if on the Teach cluster
$ cd ~/omp
$ git pull
$ source setup # loads compiler & python
$ make daxpy daxpy-parallel
```

```
$ ./daxpy
Tock registers 2.39 sec
```

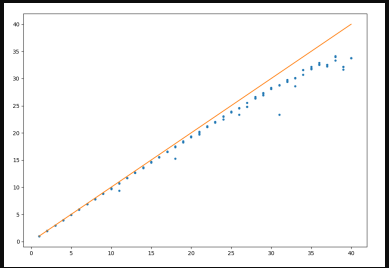
```
$ export OMP_NUM_THREADS=40
$ ./daxpy-parallel
Tock registers 0.07446 sec
```

32 times faster!

Parallel DAXPY Scaling

```
$ for P in {1..40}; do
   for r in {1..3}; do
    export OMP_NUM_THREADS=$P
    printf "$P "
    ./daxpy-parallel
   done
  done > data0.txt
```

```
$ ipython --pylab
>>> P,_,_,T,_=genfromtxt("data0.txt").T
>>> S = T[0]/T
>>> plot(P,S,'.',P,P,'-')
>>> savefig("plotdata0.png")
>>> exit
```



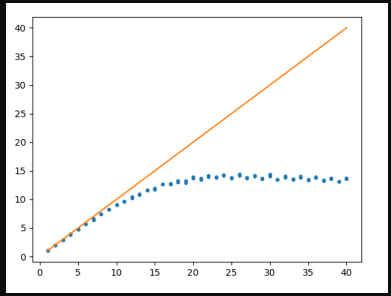
Your mileage may vary...

On purpose, we compiled with -O0.

The reason may become obvious when we look at the same graph with -O3:

```
$ make CXXFLAGS="-O3 ..."
$ make clean && make
$ for P in {1..40}; do
   for r in {1..3}; do
    export OMP_NUM_THREADS=$P
    printf "$P "
    ./daxpy-parallel
   done
  done > data3.txt
```

```
>>> P,_,_,T,_=genfromtxt("data3.txt").T
>>> S = T[0]/T
>>> plot(P,S,'.',P,P,'-')
>>> savefig("plotdata3.png")
>>> exit
```



Memory bandwidth

The issue here is that it takes time for data to be read or written to memory.

Memory has a maximum bandwidth per node. I.e., it does not increase with the number of cores.

Thus, the bandwidth may be sufficient to 'feed' a computation of single core, but not of multiple cores.

The result is that at large number of threads, memory becomes the bottleneck, and the speedup saturates.

Why does this hit optimized code more?

With -O3, the compiler generates machine code that uses the cores better, i.e., that code can do more computations in the same time.

So optimized code is more "memory hungry".

In real computations, you always want to compile with optimization.

The memory bandwidth often limits the scalabilities of shared memory programming.

Reductions


SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Dot Product

- Dot product of two vectors
- Start from a serial implementation, then will add OpenMP
- Program tells answer, correct answer, time.

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

Dot Product Code

```cpp
// dot_main.cpp
#include <print>
#include <rarray>
#include "ticktock.h"
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y);
int main()
{
  int n = 40*1000*1000;
  rvector<dbldbl> x(n), y(n);
  for (int i=0; i<n; i++)
    x[i]=y[i]=i;
  dbldbl nn = dbldbl(n);
  dbldbl ans=(nn*(nn-1)*(2*nn-1))/6;
  TickTock tt;
  tt.tick();
  dbldbl dotproduct = dot(x,y);
  tt.tock("Took");
  std::println("Dot product:  {:e}", dotproduct);
  std::println("Exact answer: {:e}", ans);
}
```

```cpp
// serial_dot.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl tot=0;
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ debugjob -n 16
$ cd ~/omp && git pull && source setup
$ make serial_dot
$ ./serial_dot
Took    1.432 sec
Dot product:  2.133333e+22
Exact answer: 2.133333e+22
$
```

Towards A Parallel Dot Product

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

```cpp
// omp_dot_race.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl 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_dot_race
$ export OMP_NUM_THREADS=16
$ ./omp_dot_race
Took    3.405 sec
Dot product:  2.663333e+21
Exact answer: 2.133333e+22
$ ./omp_dot_race
Took    3.229 sec
Dot product:  2.218010e+21
Exact answer: 2.133333e+22
```

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.
- Problem: 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

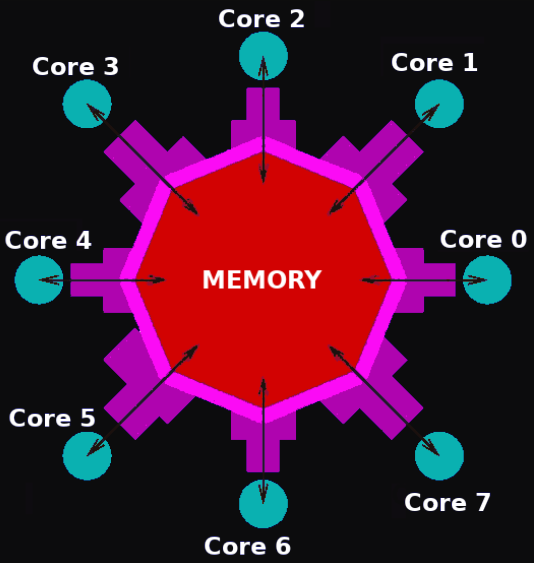
Thread 0: add 1	Thread 1: add 2
read tot=0 to reg0	.
reg0 = reg0+1	read tot=0 to reg1
store reg0(=1) in tot	reg1 = reg1 + 2
.	store reg1(=2) in tot

So it's wrong, but why is it slower?

You might think 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_dot_critical.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl 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;
}
```

Critical Construct Timing

```cpp
// omp_dot_critical.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl 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_dot_critical
$ ./omp_dot_critical
Took    36.31 sec
Dot product:  2.133333e+22
Exact answer: 2.133333e+22
```

Correct, but 25× 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_dot_atomic.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl 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;
}
```

Atomic Construct Timing

```
// omp_dot_atomic.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl 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_dot_atomic
$ ./omp_dot_atomic
Took    34.31 sec
Dot product:  2.133333e+22
Exact answer: 2.133333e+22
```

Correct, but not much faster than critical.

(It may be that `dbldbl` is not supported.)

Local Sums

The unresolved issue is that we're updating `tot`, which causes copies to main memory at every iteration.

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

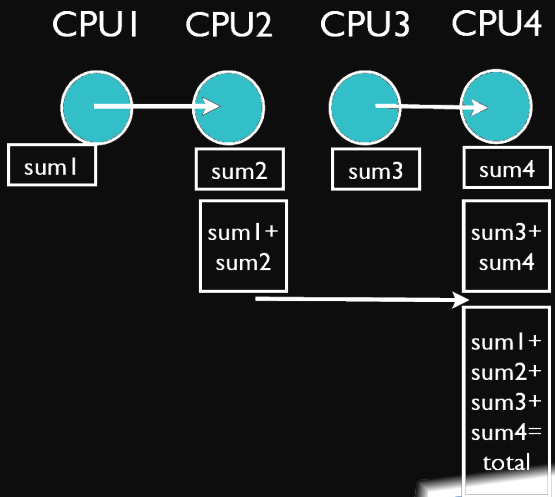
```
// omp_dot_local.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl tot=0;
  #pragma omp parallel default(none) shared(tot,n,x,y)
  {
    dbldbl localtot = 0.0;
    #pragma omp for
    for (int i=0; i<n; i++)
      localtot += x[i] * y[i];
    #pragma omp atomic update
    tot += localtot;
  }
  return tot;
}
```

```
$ make omp_dot_local
$ ./omp_dot_local
Took    0.1619 sec
Dot product:  2.133333e+22
Exact answer: 2.133333e+22
```

Correct answer, 9x 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_dot_reduction.cpp`



Reduction Timing

```cpp
// omp_dot_reduction.cpp
#include <rarray>
#include <algorithm>
using dbldbl = __float128; // compiler-extension
dbldbl dot(const rvector<dbldbl>& x,
           const rvector<dbldbl>& y)
{
  int n = std::min(x.size(), y.size());
  dbldbl tot=0;
  #pragma omp parallel default(none) shared(n,x,y) reduction(+:tot)
  #pragma omp for
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_dot_reduction
$ ./omp_dot_reduction
Took    0.1628 sec
Dot product:  2.133333e+22
Exact answer: 2.133333e+22
$
```

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)