# OpenMP Reduction Performance

Dr. Christopher Marcotte — *Durham University*

In a previous exercise, you learned the basics of data races in OpenMP, and how to avoid them using a manual approach, critical section, atomic update, or compiler-assisted reduction. Unfortunately, that problem wasn't compute-intensive enough to make the parallelism worthwhile. A calculation which is more likely to be worthwhile is a dot-product between two length-$N$ vectors:

$$a \cdot b \equiv \sum_{n=1}^{N} a_n b_n.$$

### *Hint*

You should complete the exercise about race conditions in OpenMP, first!

### *Exercise*

You should first initialize $a$ and $b$ – in parallel! – where $a_i = i + 1$ and $b_i = -i \bmod (i, 2)$, cf.

***reduction-template.c***

```c
void init(double *a, double *b, size_t N){
  /* Intialise with some values */
  // Parallelize this loop!
  for (size_t i = 0; i < N; i++) {
    a[i] = i+1;
    b[i] = (-1)*(i%2) * i;
  }
}
```

Then parallelize the function `double dot(double *a, double *b, size_t N)` using the manual, critical, atomic, and reduction approaches.

### *Solution*

A listing of the parallelized reduction clause is:

***reduction-solution.c***

```c
  #pragma omp parallel default(none) shared(a, b, N, dotabparallel)
  {
    #pragma omp for reduction(+:dotabparallel)
     for (size_t i = 0; i < N; i++) {
      dotabparallel += a[i] * b[i];
     }
  }
```

If we look at some data collected on an earlier version of Hamilton with much worse per-core performance, we see that the parallelism is *very* worthwhile, at least for $p \leq 16$ cores, and $N = 5 \times 10^8$.

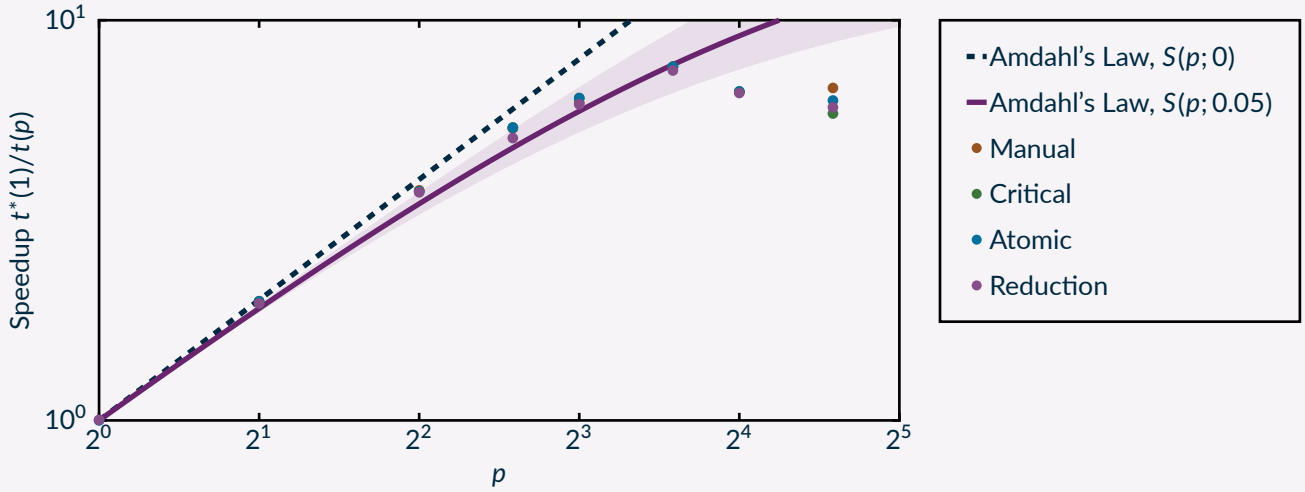


Figure 1: Speedup of reductions against the fastest serial code.

Focusing specifically on the reduction clause, the speedup for my laptop is shown in Figure 2 with $N = 100000000$. The story is the same – it is, indeed, worth parallelizing the reduction, at least up to the number of performance cores on the machine (8).

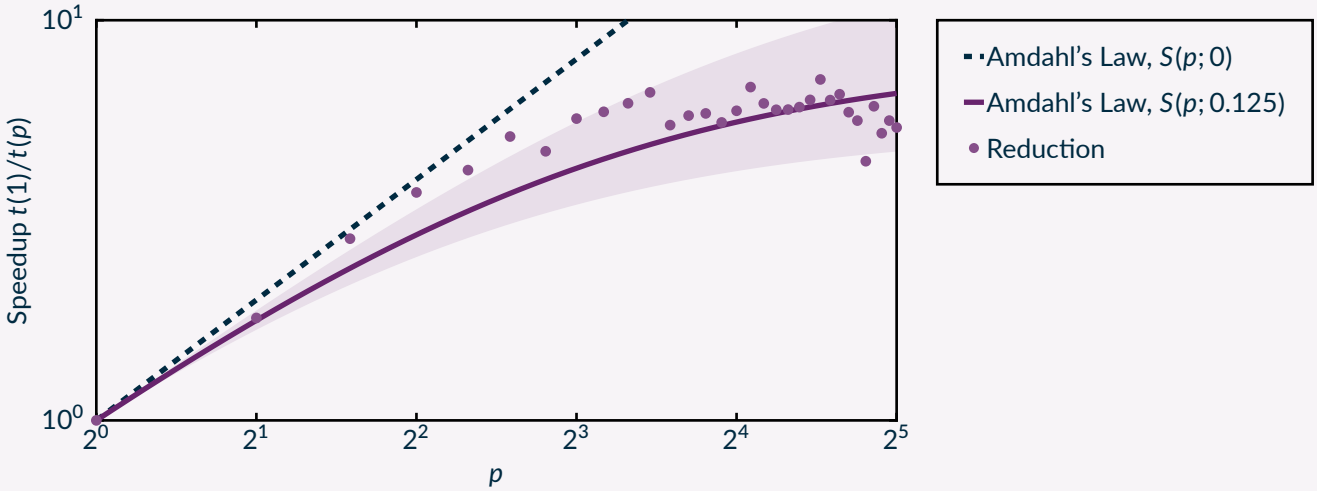


Figure 2: Speedup of reduction clause against the serial code.

### *Exercise*

Investigate the weak scaling of the code. Estimate the serial fraction, and compare in a plot to Gustafson's law.

#### *Hint*

Since we supply the size of the vectors at the command line, it is pretty easy to perform a weak scaling experiment using bash scripting:

```
gcc -O1 -fopenmp reduction-template.c -o r
for (( n = 1 ; n <= 32 ; n+=1 )); do
  OMP_NUM_THREADS=${n} ./r $((1000000 * n))
done > weak_timing.dat
```

***Solution***

We will first just plot the execution time for the $p$-scaled problem, with and without $p$ threads, in Figure 3.*

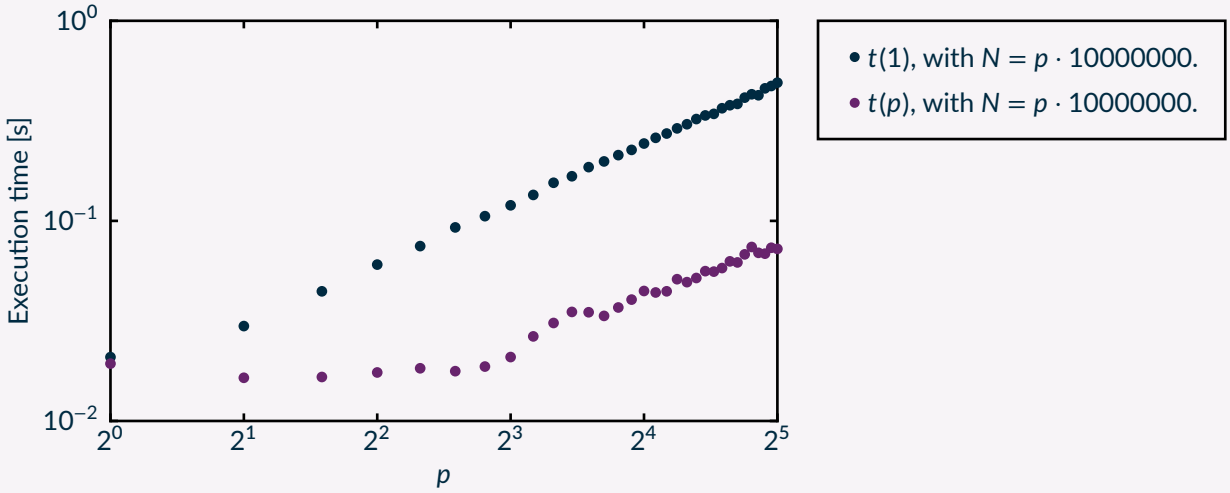


Figure 3: Execution time for the reduction clause on the $p$-scaled problem, both with and without additional threads.

It's pretty clear that this is really fast for $N = p \cdot 10000000$, with and without scaling the number of threads by $p$. In fact, you would be hard-pressed to identify a difference if you weren't measuring things carefully. Plotting the weak scaling in Figure 4 reveals that our serial fraction is pretty small; I estimate close to $f = 0.2$.

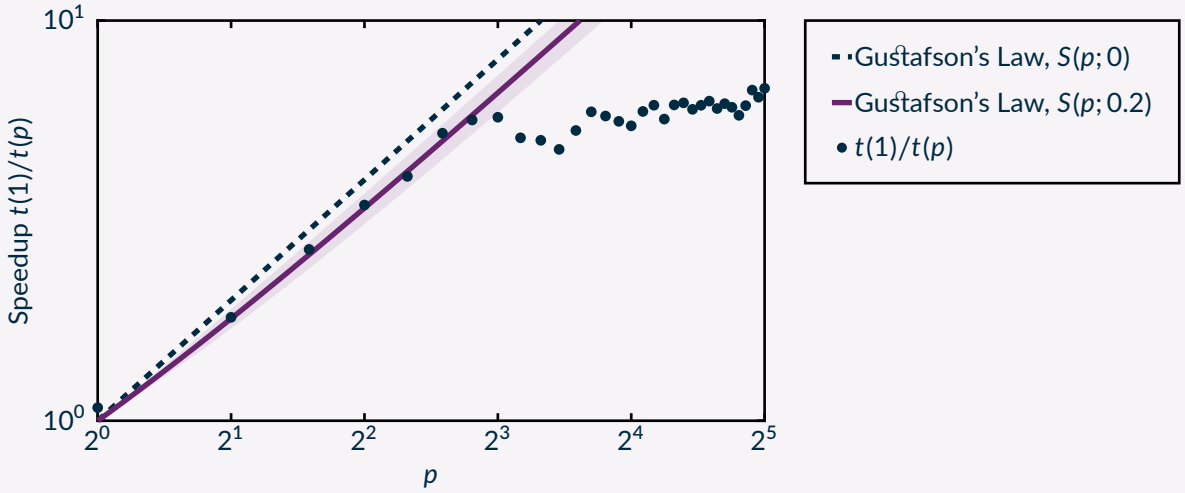


Figure 4: Speedup of reduction clause in the weak scaling sense with $N = p \cdot 10000000$.

On my laptop, the speedup stops at around $p = 8$, which corresponds to where we fall out of both L2 and L3 cache on my processor (28 MB + 24 MB, respectively), but also where we run out of high-performance cores (4P+4P+2E) so we may become bandwidth-limited, or compute-limited. Doing this experiment on Hamilton should tell you clearly one way or the other.

## Core Localisation

The Hamilton compute nodes are dual-socket. That is, they have two chips in a single motherboard, each with its own memory attached. Although OpenMP treats this logically as a single piece of shared memory, the performance of the code

*Note I am compiling with optimization level 1; this is to disadvantage the compiler enough to hopefully see some benefit.

depends on where the memory accessed is relative to where the threads are. OpenMP exposes some extra environment variables that control where threads are physically placed on the hardware. We'll look at how these variables affect the performance of our reduction. Use the implementation you determined to be the fastest above.

### *Hint*

The relevant environment variables for controlling placement are `OMP_PROC_BIND` and `OMP_PLACES`. We need to set `OMP_PLACES=cores`. *Don't forget to do this in your submission script*.

There are two options for `OMP_PROC_BIND`: `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. The former places threads on cores that are physically close to one another, first filling up the first socket, and then the second. The latter spreads threads out between cores: thread 0 to the first socket, thread 1 to the second, and so on. We'll now look at the difference in scaling performance when using different values for `OMP_PROC_BIND`.

### *Exercise*

Which value for `OMP_PROC_BIND` works better? Is there a difference at all?

Look up the other options for `OMP_PLACES`; why might `OMP_PLACES=threads` or `OMP_PLACES=sockets` be less immediately useful for measuring performance?

#### *Solution*

Running this experiment on an earlier iteration of Hamilton, we get timing results like Figure 5. Here, again $N = 5 \times 10^8$, so we are providing each thread with a considerable amount of work to do. It is clear that the spread option is more performant for intermediate numbers of threads (i.e., when $p \ll n_{\text{cores}}$). This is because it places threads on both sockets on the compute node, maximising the amount of memory bandwidth that each additional new thread can obtain (i.e., thread $p + 1$ only competes for bandwidth with the other cores on the same socket, which is $p/2$.) Once $p \approx n_{\text{cores}}$, the two options have only marginal differences.

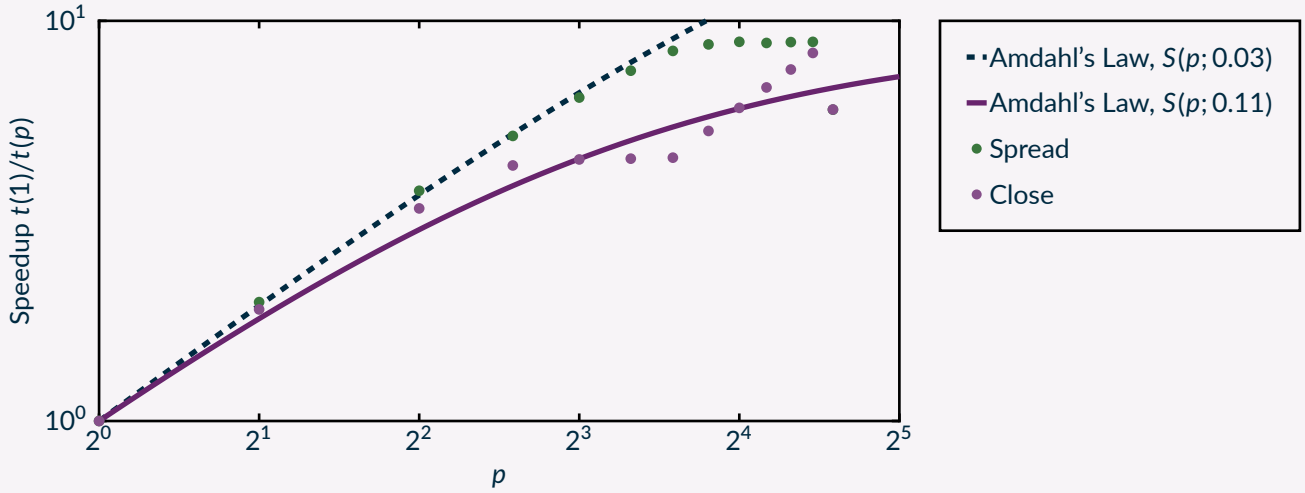


Figure 5: Speedup of reduction clause for close and spread OpenMP process binding options; $N = 5 \times 10^8$.

On a typical modern x86_64 CPU with simultaneous multi-threading (SMT), the binding to *threads* means each physical core will eventually be allocated two threads as $p$ increases, which can harm the performance of a properly-designed computation-bound program. Likewise, binding to a *socket* may hit memory bandwidth bottlenecks before a binding to cores, due to non-uniform memory access (NUMA). This can be very hard to

predict from the technical specifications of a processor; in practice, one tests for these things with programs not entirely unlike those you've written here.

***Aims***

- Practice with practial implementation of level 1 BLAS routines in OpenMP
- Practice with parallel performance scaling and diagnostics
- Practice with the specification of OpenMP environment variables and their impact on performance