

Shared Memory Programming with OpenMP

OpenMP Performance



A common scenario.....



“So I wrote my OpenMP program, and I checked it gave the right answers, so I ran some timing tests, and the speedup was, well, a bit disappointing really. Now what?”.

Most of us have probably been here.

Where did my performance go?

It disappeared into overheads.....

The six (and a half) evils...

- There are six main sources of overhead in OpenMP programs:
 - sequential code
 - idle threads
 - synchronisation
 - scheduling
 - communication
 - hardware resource contention
- and another minor one:
 - compiler (non-)optimisation
- Let's take a look at each of them and discuss ways of avoiding them.



Sequential code

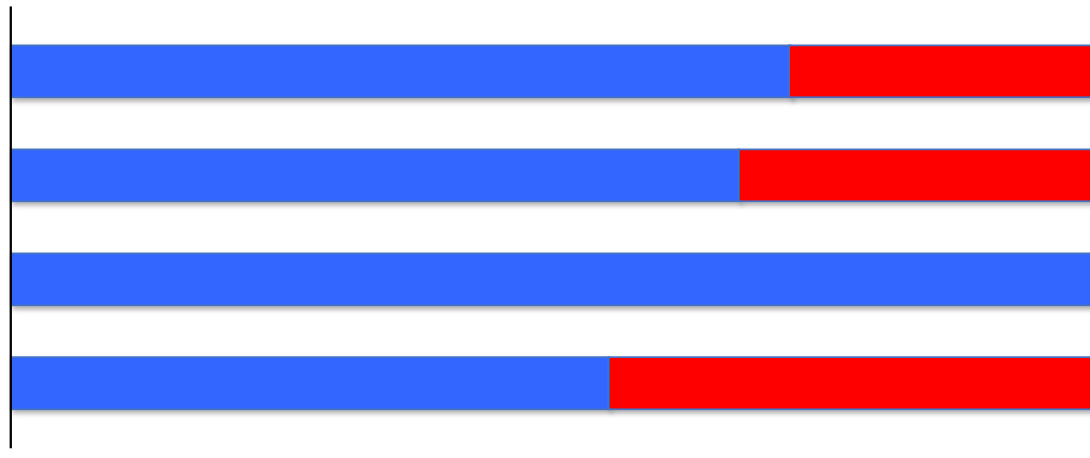


- In OpenMP, all code outside parallel regions, or inside MASTER and SINGLE directives is sequential.
- Time spent in sequential code will limit performance (that's Amdahl's Law).
- If 20% of the original execution time is not parallelised, I can never get more than 5x speedup.
- Need to find ways of parallelising it!

Idle threads



- Some threads finish a piece of computation before others, and have to wait for others to catch up.
- e.g. threads sit idle in a barrier at the end of a parallel loop or parallel region.



Time →

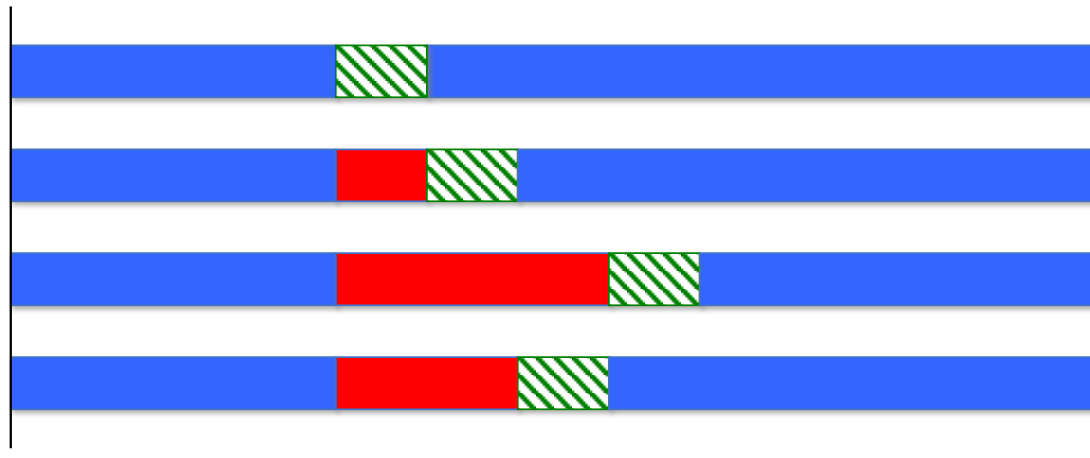
Avoiding load imbalance



- It's a parallel loop, experiment with different schedule kinds and chunk sizes
 - can use **SCHEDULE (RUNTIME)** to avoid recompilation.
- For more irregular computations, using tasks can be helpful
 - runtime takes care of the load balancing
- Note that it's not always safe to assume that two threads doing the same number of computations will take the same time.
 - the time taken to load/store data may be different, depending on if/where it's cached.

Critical sections

- Threads can be idle waiting to access a critical section
 - In OpenMP, critical regions, atomics or lock routines



Time →

Avoiding waiting

- Minimise the time spent in the critical section
- OpenMP critical regions are a global lock
 - but can use critical directives with different names
- Use atomics if possible
 - allows more optimisation, e.g. concurrent updates to different array elements
- ... or use multiple locks

Synchronisation



- Every time we synchronise threads, there is some overhead, even if the threads are never idle.
 - threads must communicate somehow.....
- Many OpenMP codes are full of (implicit) barriers
 - end of parallel regions, parallel loops
- Barriers can be very expensive
 - depends on no. of threads, runtime, hardware, but typically 1000s to 10000s of clock cycles.
- Criticals, atomics and locks are not free either.
- ...nor is creating or executing a task

Avoiding synchronisation overheads



- Parallelise at the outermost level possible.
 - Minimise the frequency of barriers
 - May require reordering of loops and/or array indices.
- *Careful* use of NOWAIT clauses.
 - easy to introduce race conditions by removing barriers that are required for correctness
- Atomics *may* have less overhead than critical or locks
 - quality of implementation problem

Scheduling



- If we create computational tasks, and rely on the runtime to assign these to threads, then we incur some overheads
 - some of this is actually internal synchronisation in the runtime
- Examples: non-static loop schedules, task constructs

```
#pragma omp parallel for schedule(dynamic,1)
for (i=0;i<10000000;i++){
    .....
}
```

- Need to get granularity of tasks right
 - too big may result in idle threads
 - too small results in scheduling overheads

Communication



- On shared memory systems, communication is “disguised” as increased memory access costs - it takes longer to access data in main memory or another processors cache than it does from local cache.
- Memory accesses are expensive! ($O(100)$ cycles for a main memory access compared to 1-3 cycles for a flop).
- Communication between processors takes place via the cache coherency mechanism.
- Unlike in message-passing, communication is fine-grained and spread throughout the program
 - much harder to analyse or monitor.

Cache coherency in a nutshell



- If a thread writes a data item, it gets an exclusive copy of the data in its local cache
- Any copies of the data item in other caches get invalidated to avoid reading of out-of-date values.
- Subsequent accesses to the data item by other threads must get the data from the exclusive copy
 - this takes time as it requires moving data from one cache to another

(Caveat : this is a *highly* simplified description!)

Data affinity

- Data will be cached on the processors which are accessing it, so we must reuse cached data as much as possible.
- Need to write code with good *data affinity* - ensure that the same thread accesses the same subset of program data as much as possible.
- Try to make these subsets large, contiguous chunks of data
- Also important to prevent threads migrating between cores while the code is running.
 - use `export OMP_PROC_BIND=true`

Data affinity example 1

```
#pragma omp parallel for schedule(static)
for (i=0;i<n;i++){
    for (j=0; j<n; j++){
        a[i][j] = i+j;
    }
}
```

Balanced loop

```
#pragma omp parallel for schedule(static,16)
for (i=0;i<n;i++){
    for (j=0; j<i; j++){
        b[j] += a[i][j];
    }
}
```

Unbalanced loop

Different access patterns
for **a** will result in extra
communication

Data affinity example 2

```
#pragma omp parallel for  
for (i=0;i<n;i++){  
    ... = a[i];  
}
```

a will be spread across
multiple caches

```
for (i=0;i<n;i++){  
    a[i] = 23;  
}
```

Sequential code!
a will be gathered into
one cache

```
#pragma omp parallel for  
for (i=0;i<n;i++){  
    ... = a[i];  
}
```


a will be spread across
multiple caches again

Data affinity (cont.)



- Sequential code will take longer with multiple threads than it does on one thread, due to the cache invalidations
- Second parallel region will scale badly due to additional cache misses
- May need to parallelise code which does not appear to take much time in the sequential program!

Data affinity: NUMA effects

- Very evil! 
- On multi-socket systems, the location of data in main memory is important.
 - Note: all current multi-socket x86 systems are NUMA!
- OpenMP has no support for controlling this.
- Common default policy for the OS is to place data on the processor which first accesses it (first touch policy).
- For OpenMP programs this can be the worst possible option
 - data is initialised in the master thread, so it is all allocated one socket
 - having all threads accessing data on the same socket becomes a bottleneck

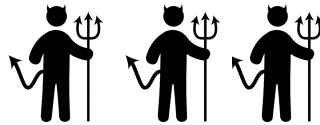
Avoiding NUMA effects



- In some OSs, there are options to control data placement
 - e.g. in Linux, can use `numactl` change policy to round-robin
- First touch policy can be used to control data placement indirectly by parallelising data initialisation
 - even though this may not seem worthwhile in view of the insignificant time it takes in the sequential code
- Don't have to get the distribution exactly right
 - some distribution is usually much better than none at all.
- Remember that the allocation is done on an OS page basis
 - typically 4KB to 16KB
 - beware of using large pages!

False sharing

- Very very evil!



- The units of data on which the cache coherency operations are done (typically 64 or 128 bytes) are always bigger than a word (typically 4 or 8 bytes).
- Different threads writing to *neighbouring words* in memory may cause cache invalidations!
 - still a problem if one thread is writing and others reading

False sharing patterns

- Worst cases occur where different threads repeatedly write neighbouring array elements.

```
count[omp_get_thread_num()]++;
```



```
#pragma omp parallel for schedule(static,1)
for (i=0;i<n;i++){
    for (j=0; j<i; j++){
        b[i] += a[j][i];
    }
}
```

Hardware resource contention



- The design of shared memory hardware is often a cost vs. performance trade-off.
- There are shared resources which if all cores try to access at the same time, do not scale.
 - or, put another way, an application running on a single core can access more than its fair share of the resources
- In particular, cores (and hence OpenMP threads) can contend for:
 - memory bandwidth
 - cache capacity
 - functional units (if using SMT)

Memory bandwidth



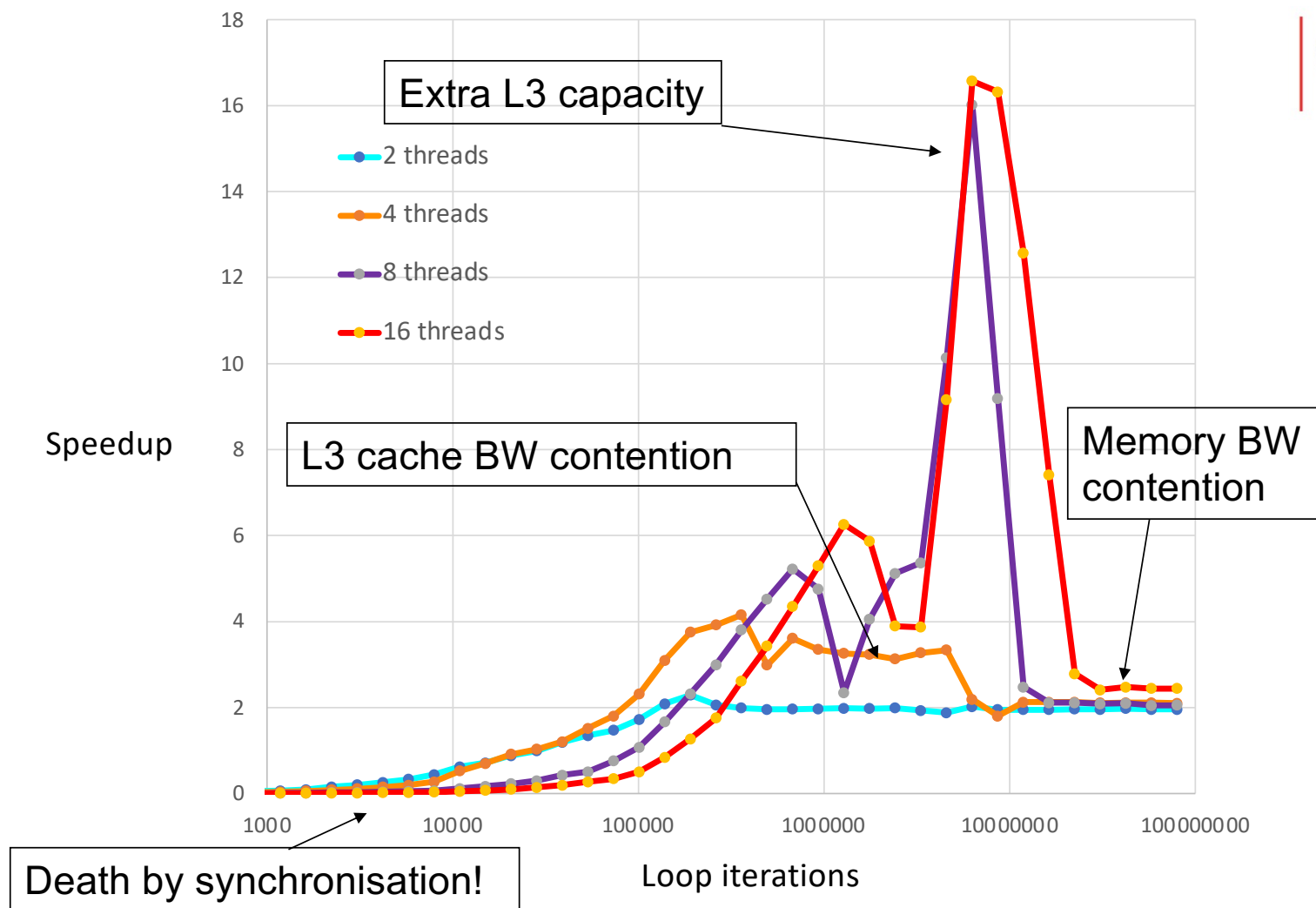
- Codes which are very bandwidth-hungry will not scale linearly on most shared-memory hardware.
- Try to reduce bandwidth demands by improving locality, and hence the re-use of data in caches
 - will benefit the sequential performance as well.

Memory bandwidth example

- AMD Rome processor
 - 1 NUMA region - 16 cores
 - L1 and L2 caches per core
 - 16 MB shared L3 cache per 4 cores
 - Cray compiler

```
#pragma omp parallel for reduction(+:sum)
for (i=0;i<n;i++){
    sum += a[i];
}
```

- Measure speedup for varying values of n and no. of threads



Cache space contention



- On systems where cores share some level of cache (e.g. L3), codes may not appear to scale well because a single core can access the whole of the shared cache.
- Beware of tuning block sizes for a single thread, and then running multithreaded code
 - each thread will try to utilise the whole cache

Hardware threads



- When using hardware threads, OpenMP threads running on the same core contend for functional units as well as cache space and memory bandwidth.
- Tends to benefit codes where threads are idle because they are waiting on memory references
 - code with non-contiguous/random memory access patterns
- Codes which are bandwidth-hungry, or which saturate the floating point units (e.g. dense linear algebra) may not benefit from this
 - may actually run slower

Oversubscription



- Running more threads than hardware execution units (cores or hardware threads) is generally a bad idea.
- OS tries to give each thread a fair share of execution units
- Cost of stopping one thread and starting another is high (1000s of clock cycles)
- Ruins data locality!

Compiler (non-)optimisation



- Very rarely, the addition of OpenMP directives can inhibit the compiler from performing sequential optimisations.
- Symptoms: 1-thread parallel code has longer execution time than sequential code.
- Can be hard to find a workaround
- Can sometimes be cured by making shared data private, or making local copies of variables.

Minimising overheads

My code is giving poor speedup. I don't know why.

What do I do now?

1.

- Say “OpenMP is a heap of junk”.
- Give up.

2.

- Try to *classify* and *localise* the sources of overhead.
- What type of problem is it, and where in the code does it occur?
- Use any available tools to help you (e.g. timers, hardware counters, profiling tools).
- Fix problems which are responsible for large overheads first.
- Iterate.



Profilers



- Standard profilers (gprof, IDE profilers) can be confusing
 - they typically accumulate the time spent in functions across all threads.
- You can get a lot out of using timers (`omp_get_wtime()`)
- Add timers round every parallel region, and round the whole code.
 - work out which parallel regions have the worst speedup
 - don't assume the time spent outside parallel regions is independent of the number of threads.

Performance tools



- Vampir
 - timeline traces can be very useful for visualising load balance
- Intel Vtune
- TAU
- Arm MAP
- CrayPAT
- Scalasca
 - breaks down overheads into different categories
- ParaTools Threadspotter
 - very good for finding cache/memory problems, including false sharing.

Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.