



Multicore Workshop

Performance Tuning

Mark Bull

David Henty

- There are 6 main causes of poor performance in shared memory parallel programs:
 - sequential code
 - communication
 - load imbalance
 - synchronisation
 - hardware resource contention
 - compiler (non-)optimisation
- We will take a look at each and discuss ways to address them

- Amount of sequential code will limit performance (Amdahl's Law)
- Need to find ways of parallelising it!
- In OpenMP, all code outside parallel regions, and inside MASTER, SINGLE and CRITICAL directives is sequential - this code should be as small as possible.

- On shared memory machines, 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! (~300 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 spread throughout the program. This makes it much harder to analyse or monitor.

- Data will be cached on the processors which are accessing it, so we must reuse cached data as much as possible.
- Try to write code with good *data affinity* - ensure that the same thread accesses the same subset of program data as much as possible.
- Also try to make these subsets large, contiguous chunks of data (avoids false sharing)

Example:

```
!$OMP DO PRIVATE (I)
```

```
  do j = 1,n
```

```
    do i = 1,n
```

```
      a(i,j) = i+j
```

```
    end do
```

```
  end do
```

```
!$OMP DO SCHEDULE (STATIC,16) PRIVATE (I)
```

```
  do j = 1,n
```

```
    do i = 1,j
```

```
      b(j) = b(j) + a(i,j)
```

```
    end do
```

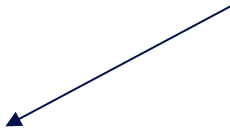
```
  end do
```

Different access patterns
for **a** will result in
additional cache misses

Example:

```
!$OMP PARALLEL DO
  do i = 1,n
    ... = a(i)
  end do
```

a will be spread across multiple caches



```
a(:) = 26.0
```

Sequential code!
a will be gathered into one cache



```
!$OMP PARALLEL DO
  do i = 1,n
    ... = a(i)
  end do
```

a will be spread across multiple caches again



- 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.

- On distributed shared memory (cc-NUMA) systems, the location of data in main memory is important.
 - Note: all current multi-socket x86 systems are cc-NUMA!
- OpenMP has no support for controlling this (and there is still a debate about whether it should or not!).
- 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 node
 - having all threads accessing data on the same node become a bottleneck

- 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!

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

Cures:

1. Padding of arrays. e.g.:

```
integer count(maxthreads)
```

```
!$OMP PARALLEL
```

```
. . .
```

```
count(myid) = count(myid) + 1
```

becomes

```
parameter (linesize = 16)
```

```
integer count(linesize,maxthreads)
```

```
!$OMP PARALLEL
```

```
. . .
```

```
count(1,myid) = count(1,myid) + 1
```

2. Watch out for small chunk sizes in unbalanced loops e.g.:

```
!$OMP DO SCHEDULE (STATIC,1)
  do j = 1,n
    do i = 1,j
      b(j) = b(j) + a(i,j)
    end do
  end do
```

may induce false sharing on **b**.

- Note that load imbalance can arise from imbalances in communication as well as in computation.
- Experiment with different loop scheduling options - use **SCHEDULE (RUNTIME)** .
- If none of these are appropriate, don't be afraid to use a parallel region and do your own scheduling (it's not that hard!). e.g. an irregular block schedule might be best for some triangular loop nests.
- For more irregular computations, using tasks can be helpful
 - runtime takes care of the load balancing

```
!$OMP PARALLEL DO SCHEDULE (STATIC,16) PRIVATE (I)
  do j = 1,n
    do i = 1,j
      . . .
```

becomes

```
!$OMP PARALLEL PRIVATE (LB,UB,MYID,I)
  myid = omp_get_thread_num()
  lb = int(sqrt(real(myid*n*n)/real(nthreads)))+1
  ub = int(sqrt(real((myid+1)*n*n)/real(nthreads)))
  if (myid .eq. nthreads-1) ub = n
  do j = lb, ub
    do i = 1,j
      . . .
```

- Barriers can be very expensive (typically 1000s to 10000s of clock cycles).
- *Careful* use of NOWAIT clauses.
- Parallelise at the outermost level possible.
 - May require reordering of loops and/or array indices.
- Choice of CRITICAL / ATOMIC / lock routines may have performance impact.

- The NOWAIT clause can be used to suppress the implicit barriers at the end of DO/FOR, SECTIONS and SINGLE directives.

Syntax:

Fortran: **!\$OMP DO**

do loop

!\$OMP END DO NOWAIT

C/C++: **#pragma omp for nowait**

for loop

- Similarly for SECTIONS and SINGLE.

Example: Two loops with no dependencies

```
!$OMP PARALLEL
!$OMP DO
    do j=1,n
        a(j) = c * b(j)
    end do
!$OMP END DO NOWAIT
!$OMP DO
    do i=1,m
        x(i) = sqrt(y(i)) * 2.0
    end do
!$OMP END PARALLEL
```

- Use with **EXTREME CAUTION!**
- All too easy to remove a barrier which is necessary.
- This results in the worst sort of bug: non-deterministic behaviour (sometimes get right result, sometimes wrong, behaviour changes under debugger, etc.).
- May be good coding style to use NOWAIT everywhere and make all barriers explicit.

Example:

```
!$OMP DO SCHEDULE (STATIC,1)
  do j=1,n
    a(j) = b(j) + c(j)
  end do

!$OMP DO SCHEDULE (STATIC,1)
  do j=1,n
    d(j) = e(j) * f
  end do

!$OMP DO SCHEDULE (STATIC,1)
  do j=1,n
    z(j) = (a(j)+a(j+1)) * 0.5
  end do
```

Can remove the first barrier, *or* the second, but not both, as there is a dependency on **a**

- 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 code can access more than its fair share of the resources
- In particular, OpenMP threads can contend for:
 - memory bandwidth
 - cache capacity
 - functional units (if using SMT)

- Codes which are very bandwidth-hungry will not scale linearly of 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.

- On systems where cores share some level of cache, 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

- When using SMT, threads running on the same core contend for functional units as well as cache space and memory bandwidth.
- SMT 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 SMT
 - might run slower

- Sometimes the addition of parallel directives can inhibit the compiler from performing sequential optimisations.
- Symptoms: 1-thread parallel code has longer execution time and higher instruction count than sequential code.
- Can sometimes be cured by making shared data private, or local to a routine.

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

What do I do now?

1.

- Say “this machine/language is a heap of junk”.
- Give up and go back to your workstation/PC.

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.