NATURAL ENVIRONMENT RESEARCH COUNCIL

EPSRC

ARCHER Single Node Optimisation

Optimising multi-threaded code

Slides contributed by Cray and EPCC

epcc

THE UNIVERSITY OF EDINBURGH

Sources of overhead

- There are 6 main causes of poor performance in threaded 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
- Consider the special case of MPI + threads

Sequential code

- 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 as small as possible.

Communication

- 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 affinity

- 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)
- Note: MPI programs have good data affinity by default!

Data affinity (cont)

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

Data affinity (cont)

Example:

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

        a(:) = 26.0

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

a will be spread across multiple caches

Sequential code!
a will be gathered into one cache

a will be spread across multiple caches again

epcc

THE UNIVERSITY OF EDINBURGH

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

- 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!
- 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 huge pages!

False sharing

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

Load imbalance

- 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

Synchronisation

- 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.

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 them 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, 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 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.

Cache space contention

- 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

SMT

- 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

SMT on ARCHER

- Ivy Bridge processors supports 1 or 2 SMT threads (hyperthreads) per core
- Default is to use 1 hyperthread per core
- Can enable 2 hyperthreads per core with `aprun -j 2`
- Run 48 processes/threads per node
- Need to take some care with thread placement
- Benefits often do not outweigh the overheads of doubling the number of MPI processes, or threads
  - especially if you are already running close to the limit of scalability

Compiler (non-)optimisation

- 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.

Hybrid MPI + threads

- Many applications use hybrid parallelism for improved scalability and/or reducing memory usage.
- Usually MPI + OpenMP, sometimes MPI + Posix threads
- Introduces its own set of single node optimisation problems

Styles of mixed-mode programming

- Master-only
  - all MPI communication takes place in the sequential part of the OpenMP program (no MPI in parallel regions)
- Funneled
  - all MPI communication takes place through the same (master) thread
  - can be inside parallel regions
- Serialized
  - only one thread makes MPI calls at any one time
  - distinguish sending/receiving threads via MPI tags or communicators
  - be very careful about race conditions on send/recv buffers etc.
- Multiple
  - MPI communication simultaneously in more than one thread
  - some MPI implementations don't support this
  - …and those which do mostly don't perform well

OpenMP Master-only

```
!$OMP parallel
 work…
!$OMP end parallel

call MPI_Send(…)

!$OMP parallel
 work…
!$OMP end parallel
```

```
#pragma omp parallel
{
    work…
}
ierror=MPI_Send(…);
#pragma omp parallel
{
    work…
}
```

OpenMP Funneled

```
!$OMP parallel
… work
!$OMP barrier
!$OMP master
  call MPI_Send(…)
!$OMP end master
!$OMP barrier
.. work
!$OMP end parallel
```

```
#pragma omp parallel
{
   … work
  #pragma omp barrier
  #pragma omp master
  {
    ierror=MPI_Send(…);
  }
 #pragma omp barrier
   … work
}
```

OpenMP Serialized

```
!$OMP parallel
… work
!$OMP critical
   call MPI_Send(…)
!$OMP end critical
… work
!$OMP end parallel
```

```
#pragma omp parallel
{
   … work
   #pragma omp critical
   {
      ierror=MPI_Send(…);
   }
   … work
}
```

OpenMP Multiple

```
!$OMP parallel
… work
call MPI_Send(…)
… work
!$OMP end parallel
```

```
#pragma omp parallel
{
   … work
  ierror=MPI_Send(…);
   … work
}
```

Pitfalls

- The OpenMP implementation may introduce additional overheads not present in the MPI code (e.g. synchronisation, false sharing, sequential sections).
- The mixed implementation may require more synchronisation than a pure OpenMP version, if non-thread-safety of MPI is assumed.
- Implicit point-to-point synchronisation may be replaced by (more expensive) barriers.

- In the pure MPI code, the intra-node messages will often be naturally overlapped with inter-node messages
  - harder to overlap inter-thread communication with inter-node messages.
- NUMA effects can limit the scalability of OpenMP: it may be advantageous to run one MPI process per NUMA domain, rather than one MPI process per node.
  - process placement becomes very important
  - On ARCHER each socket (12 cores) is a NUMA domain

Master-only

- Advantages
  - simple to write and maintain
  - clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - no concerns about synchronising threads before/after sending messages

Master-only

- Disadvantages
  - threads other than the master are idle during MPI calls (sequential code at the threading level)
  - all communicated data passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication do not overlap.
  - only way to synchronise threads before and after message transfers is by parallel regions which have a relatively high overhead.
  - packing/unpacking of derived datatypes is sequential.

Example

```
!$omp parallel do
      DO I=1,N * nthreads
         A(I) = B(I) + C(I)
      END DO

      CALL MPI_BSEND(A(N),1,.....)
      CALL MPI_RECV(A(0),1,.....)

!$omp parallel do
      DO I = 1,N * nthreads
         D(I) = A(I-1) + A(I)
      END DO
```

Implicit barrier added here

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Funneled

- Advantages
  - relatively simple to write and maintain
  - cheaper ways to synchronise threads before and after message transfers
  - possible for other threads to compute while master is in an MPI call
- Disadvantages
  - less clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - all communicated data still passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication still do not overlap.

OpenMP Funneled with overlapping (1)

```
#pragma omp parallel
{
   … work
  #pragma omp barrier
  if (omp_get_thread_num() == 0)   {
    ierror=MPI_Send(…);
  }
  else {
    do some computation
  }
 #pragma omp barrier
   … work
}
```

Can't using worksharing here!

OpenMP Funneled with overlapping (2)

```
#pragma omp parallel num_threads(2)
{
if (omp_get_thread_num() == 0)   {
    ierror=MPI_Send(…);
  }
  else {
#pragma omp parallel
    {
       do some computation
    }
  }
}
```

Higher overheads and harder to synchronise between teams

Serialised

- Advantages
  - easier for other threads to compute while one is in an MPI call
  - can arrange for threads to communicate only their “own” data (i.e. the data they read and write).
- Disadvantages
  - getting harder to write/maintain
  - more, smaller messages are sent, incurring additional latency overheads
  - need to use tags or communicators to distinguish between messages from or to different threads in the same MPI process.

Distinguishing between threads

- By default, a call to MPI_Recv by any thread in an MPI process will match an incoming message from the sender.
- To distinguish between messages intended for different threads, we can use MPI tags
  - if tags are already in use for other purposes, this gets messy
- Alternatively, different threads can use different MPI communicators
  - OK for simple patterns, e.g. where thread N in one process only ever communicates with thread N in other processes
  - more complex patterns also get messy

Multiple

- Advantages
  - Messages from different threads can (in theory) overlap
    - many MPI implementations serialise them internally.
  - Natural for threads to communicate only their “own” data
  - Fewer concerns about synchronising threads (responsibility passed to the MPI library)
- Disdavantages
  - Hard to write/maintain
  - Not all MPI implementations support this – loss of portability
  - Most MPI implementations don’t perform well like this
    - Thread safety implemented crudely using global locks.