OpenMP for Computational Scientists

4: Combining MPI and OpenMP

Tom Deakin
University of Bristol

Outline

University of BRISTOL

- Quick recap
- Calculating memory bandwidth for the 5-point stencil code

Programming beyond a single multi-core CPU:

- Non-uniform Memory Access
- Thread affinity in OpenMP
- Combining MPI with OpenMP

Recap

University of BRISTOL

We've already come a long way!

- Parallelise loops with OpenMP: `!$omp parallel do`.
- Data sharing clauses.
- Synchronisation with barriers, atomics and `critical` regions.
- Reductions with the `reduction` clause.
- The cache hierarchy.
- Performance analysis and the Roofline model.
- Vectorisation along with the OpenMP `simd` construct.
- Optimisations for memory access.

Previous exercise

Vectorise and optimise memory access patterns of your parallel 5-point stencil code:

```
!$omp parallel do reduction(+:total)
do j = 1, ny
  !$omp simd
  do i = 1, nx
    Atmp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1) + A(i,j+1)) *
    ↪  0.2
    total = total + Atmp(i,j)
  end do
  !$omp end simd
end do
!$omp end parallel do
```

- Swapped loops to ensure stride-1 access pattern.
- Removed division!
- Use `simd` construct on inner loop (removing `collapse` clause).
- Checked vectorisation report: assume sizes arrays cause issue, so move kernel into `subroutine`.

Calculating memory bandwidth

Is your 5-point stencil code *fast*?

Calculating memory bandwidth

Is your 5-point stencil code *fast*?
Calculate memory bandwidth of the *kernel* as a whole:

- Assume a "perfect cache" model: once you read a memory location, it's been cached and further reads are "free" within the kernel.

Calculating memory bandwidth

Is your 5-point stencil code *fast*?
Calculate memory bandwidth of the *kernel* as a whole:

- Assume a "perfect cache" model: once you read a memory location, it's been cached and further reads are "free" within the kernel.
- All of `A` array is read: $nx \times ny$ reads.

Calculating memory bandwidth

Is your 5-point stencil code *fast*?
Calculate memory bandwidth of the *kernel* as a whole:

- Assume a "perfect cache" model: once you read a memory location, it's been cached and further reads are "free" within the kernel.
- All of `A` array is read: $nx \times ny$ reads.
- All of `Atmp` array is written: $nx \times ny$ reads.

Calculating memory bandwidth

Is your 5-point stencil code *fast*?
Calculate memory bandwidth of the *kernel* as a whole:

- Assume a "perfect cache" model: once you read a memory location, it's been cached and further reads are "free" within the kernel.
- All of `A` array is read: $nx \times ny$ reads.
- All of `Atmp` array is written: $nx \times ny$ reads.
- Total memory moved: $2 \times nx \times ny \times 8$ bytes data moved (double precision) *per iteration*.

Calculating memory bandwidth

University of BRISTOL

Is your 5-point stencil code *fast*?
Calculate memory bandwidth of the *kernel* as a whole:

- Assume a "perfect cache" model: once you read a memory location, it's been cached and further reads are "free" within the kernel.
- All of `A` array is read: $nx \times ny$ reads.
- All of `Atmp` array is written: $nx \times ny$ reads.
- Total memory moved: $2 \times nx \times ny \times 8$ bytes data moved (double precision) *per iteration*.
- Memory bandwidth: $\frac{ntimes \times 2 \times nx \times ny \times 8}{runtime}$ bytes/second.

Achieved memory bandwidth

Results on dual-socket Intel Xeon E5-2680 v4 @ 2.40GHz, 14 cores/socket. Compiled with Intel 2018 compiler, `-O3 -xHost`.

Set $nx = ny = 20,000$ so arrays are 3.2 GB. Set $ntimes = 30$. Removed `write` statement. Taken best of 5 runs.

[1] `https://ark.intel.com/products/91754/Intel-Xeon-Processor-E5-2680-v4-35M-Cache-2_40-GHz`

Achieved memory bandwidth

Results on dual-socket Intel Xeon E5-2680 v4 @ 2.40GHz, 14 cores/socket. Compiled with Intel 2018 compiler, `-O3 -xHost`.

Set $nx = ny = 20,000$ so arrays are 3.2 GB. Set $ntimes = 30$. Removed `write` statement. Taken best of 5 runs.

Theoretical peak bandwidth[1]: $2 \times 76.8\text{GB/s} = 153.6\text{GB/s}$.
STREAM Triad: 129.0 GB/s (84% theoretical peak).

[1] `https://ark.intel.com/products/91754/Intel-Xeon-Processor-E5-2680-v4-35M-Cache-2_40-GHz`

Achieved memory bandwidth

Results on dual-socket Intel Xeon E5-2680 v4 @ 2.40GHz, 14 cores/socket. Compiled with Intel 2018 compiler, `-O3 -xHost`.

Set $nx = ny = 20,000$ so arrays are 3.2 GB. Set $ntimes = 30$. Removed `write` statement. Taken best of 5 runs.

Theoretical peak bandwidth[1]: $2 \times 76.8\text{GB/s} = 153.6\text{GB/s}$.
STREAM Triad: 129.0 GB/s (84% theoretical peak).

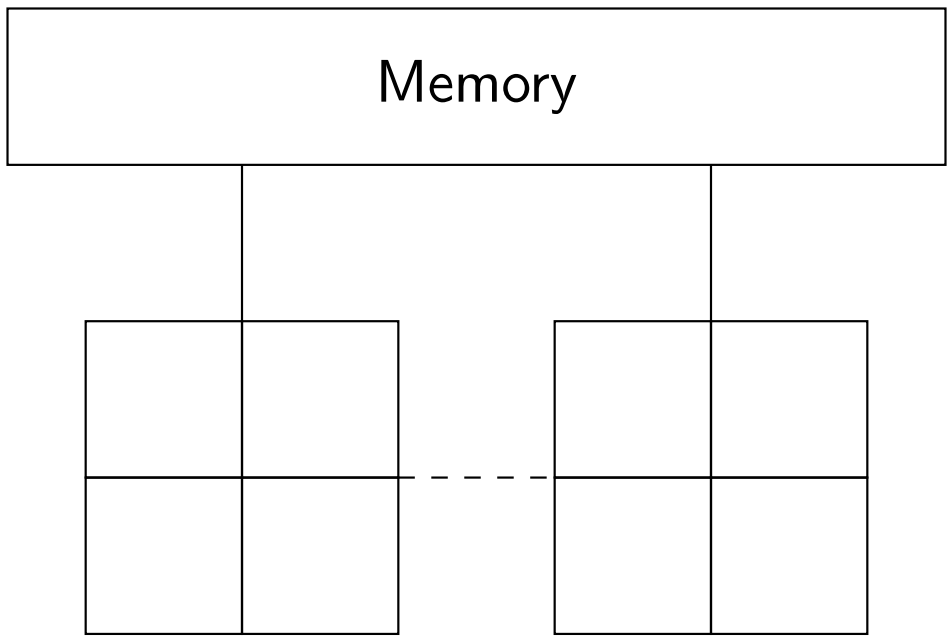
Version	Runtime (s)	Memory bandwidth (GB/s)
Initial parallel reduction	25.667	7.48
Swap loops + vectorise	4.876	39.38

Achieving 30.5% of STREAM memory bandwidth.

[1] `https://ark.intel.com/products/91754/Intel-Xeon-Processor-E5-2680-v4-35M-Cache-2_40-GHz`

NUMA Architecture

Recall this cartoon of a dual-socket, shared memory system:

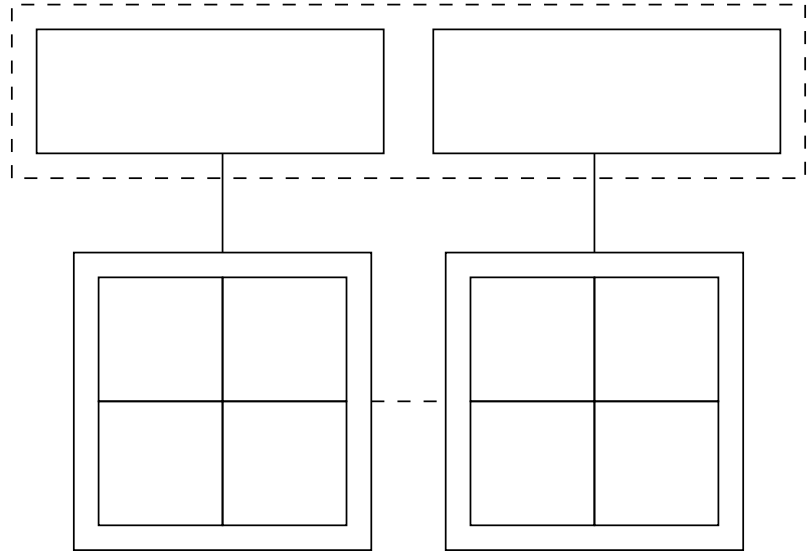


All threads (each running on a core) can access the same memory.

NUMA Architecture

University of BRISTOL

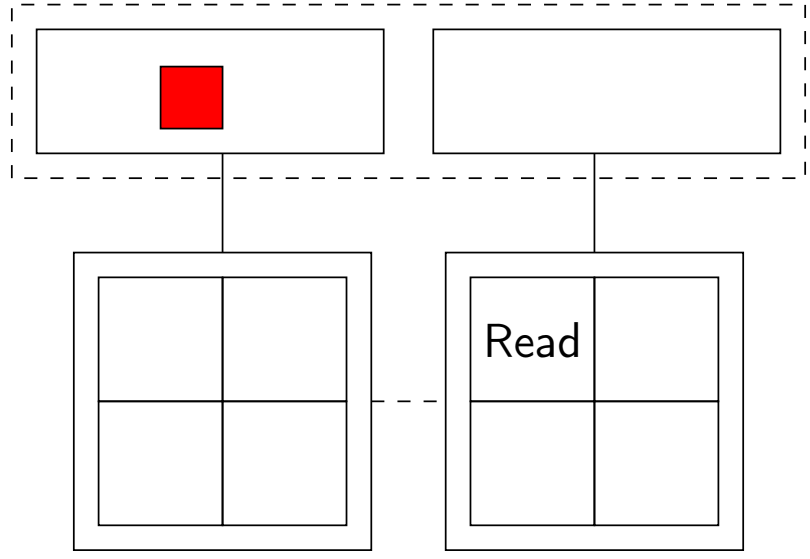
- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
  - Go via the socket-to-socket interconnect.
  - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



NUMA Architecture

University of BRISTOL

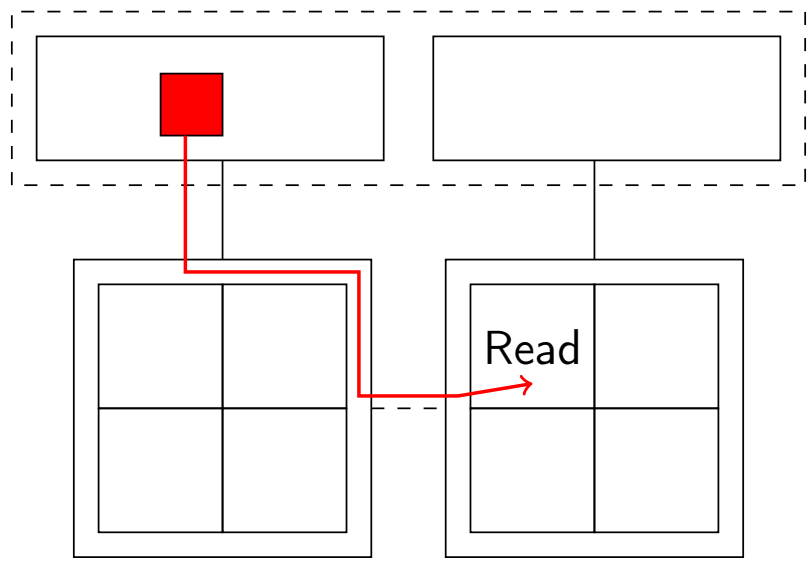
- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
  - Go via the socket-to-socket interconnect.
  - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



NUMA Architecture

University of BRISTOL

- In reality on a dual-socket system each *socket* is physically connected to half of the memory.
- Still shared memory: all cores can access all the memory.
- A core in the first socket wanting memory attached to the other socket must:
  - Go via the socket-to-socket interconnect.
  - Access memory via the other socket's memory controllers.
- Accessing memory from other socket is slower than access from own socket.



Memory allocation

- What happens when you run `allocate(A(1:N))`?

Memory allocation

University of BRISTOL

- What happens when you run `allocate(A(1:N))`?
- Allocating memory does not necessarily allocate memory!
- Memory is allocated when it's first used (i.e. `A(i) = 1.0`), one *page* at a time.
- OS tends to use a *first touch policy*.
- Memory is allocated in the closest NUMA region to the thread that first touches the data.
- Ideally want threads to use data in local NUMA region to reduce socket-to-socket interconnect transfers.

Taking advantage of first touch

Parallelising your data initialisation routine might mean your main loops go faster!

```
1  ! Allocate and initialise vectors
2  allocate(A(N), B(N), C(N))
3  !$omp parallel do
4  do i = 1, N
5    A(i) = 1.0
6    B(i) = 2.0
7    C(i) = 0.0
8  end do
9  !$omp end parallel do
10
11 ! Vector add
12 !$omp parallel do
13 do i = 1, N
14   C(i) = A(i) + B(i)
15 end do
16 !$omp end parallel do
```

NUMA-aware

University of BRISTOL

- Parallelise your initialisation routines the same way you parallelise the main loops.
- This means each thread touches the same data in initialisation and compute.
- Should reduce the number of remote memory accesses needed and improve run times.
- But, OS is allowed to move threads around cores, and between sockets.
- This will mess up your NUMA aware code!

Pinning threads

University of BRISTOL

- OpenMP gives you the controls to pin threads to specific cores.
- Exposed as *places* and *thread pinning policy* to those places.
- By default there is one place consisting of all the cores.
- Use the `OMP_PROC_BIND` environment variable to set pinning for all `parallel` regions.
- Can use the `proc_bind` clause for control of specific regions, but advise against this.

OMP_PROC_BIND

University of BRISTOL

- `OMP_PROC_BIND=false`: Often the default; threads may move! `proc_bind` clauses ignored.
- `OMP_PROC_BIND=true`: Threads won't move, and follow `proc_bind` clauses or else the implementation default pinning.
- `OMP_PROC_BIND=master`: Threads pinned to same place as master thread.
- `OMP_PROC_BIND=close`: Threads are assigned to places close to the master thread. If `OMP_NUM_THREADS.eq.ncores`: thread 0 will pin to core 0; thread 1 will pin to core 1; etc
- `OMP_PROC_BIND=spread`: Threads are assigned to places "sparsely". If `OMP_NUM_THREADS.eq.ncores`: thread 0 will pin to socket 0 core 0; thread 1 will pin to socket 1 core 0; thread 2 will pin to socket 0 core 1; etc.

Places

University of BRISTOL

- The affinity (policy) defines how threads are assigned to places.
- Places allow you to divide up the hardware resource, so that threads can be assigned to them.
- Default: one place with all cores.
- Use `OMP_PLACES` environment variable to control.
- `OMP_PLACES=thread`: each place is a single hardware thread.
- `OMP_PLACES=cores`: each place is a single core (containing one or more hardware threads).
- `OMP_PLACES=sockets`: each place contains the cores of a single socket.
- Can also use list notation:
  `OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"`

Thread pinning summary

University of BRISTOL

- In general, going to want to just use `OMP_PROC_BIND=true`.
- Sometimes `spread` or `close` gets better performance.
- Pinning rules can get complicated when there are multiple places, so prefer to use the predefined values.
- Most effective with a NUMA-aware implementation.
- Also helps reduce run-to-run timing variability.
- But must be careful with MPI+OpenMP pinning: more on this later...

Why combine MPI+OpenMP

University of BRISTOL

- Supercomputers are often constructed with a hierarchical structure:
  - Shared memory nodes connected with a network.
- Need MPI (or similar) to communicate between distributed nodes.
- With multi-core, could just run MPI everywhere (flat MPI).
- But there are advantages to running *hybrid* MPI and OpenMP:
  - Larger fewer messages to take advantage of network bandwidth.
  - Fewer MPI ranks to manage (fewer to synchronise and for collectives).
  - Can avoid memory copies for intra-node communication.
  - Reduced memory footprint.
  - Parallelise other problem dimensions not decomposed with MPI.

Scaling

University of BRISTOL

- Strong scaling:
  - Take a fixed problem and add more compute resource.
  - Would hope runtime reduces with more resource.
- Weak scaling:
  - Take a fixed problem *per compute resource*, and add more resource.
  - Problem gets bigger with more resources.
  - Would hope runtime stays constant.
- In both cases, typically see scaling of MPI-only codes tail off at high node counts.
- Hybrid MPI+OpenMP codes often continue scaling.

MPI programs

University of BRISTOL

What happens when you run an MPI program?

```
mpirun -np 16 ./a.out
```

- 16 processes are spawned on one (or more) nodes according to the hostname list file given by the queuing system.
  - E.g. with PBS (`qsub`, etc.) set by `$PBS_NODEFILE`.
- There is no reason why these processes have to be serial:
  - Each MPI rank could spawn OpenMP threads and run in parallel.
  - Each MPI rank could use a GPU.

Compiling OpenMP and MPI code

University of BRISTOL

- Remember building MPI code just uses the wrapper commands.
- Just pass in the OpenMP flag as usual:
  - GNU: `mpif90 -fopenmp`
  - Intel: `mpiifort -qopenmp`
  - Cray: `ftn`
- Set the number of OpenMP threads *per rank*.
- E.g 2 MPI ranks, 8 threads per rank:

```
OMP_NUM_THREADS=8 mpirun -np 2 ./a.out
```

Combining OpenMP and MPI

University of BRISTOL

- MPI assumes that each MPI process does not spawn anything else.
- Must initialise MPI differently if using threads!

```
call MPI_Init_thread(required, provided, ierr)
```

- You specify a required thread support level, and it returns the level it could support.
- A good idea to check `provided .ge. required`.

Thread support levels

University of BRISTOL

- `MPI_THREAD_SINGLE`
  Only one thread will execute (no threads allowed).
- `MPI_THREAD_FUNNELED`
  May spawn threads, but only the original process may call MPI routines: the one that called `MPI_Init`.
- `MPI_THREAD_SERIALIZED`
  May spawn threads and any thread can make MPI calls, but only one at a time. *Your* responsibility to synchronise.
- `MPI_THREAD_MULTIPLE`
  May spawn threads and any thread can make MPI calls. The MPI library has to deal with being called in parallel.

Remember to make sure ranks still match the MPI communications to avoid deadlock.

Example: MPI_THREAD_FUNNELED

Only the original process is allowed to call MPI routines.

```
!$omp parallel
... ! Parallel work
!$omp end parallel
call MPI_Sendrecv()
```

MPI_THREAD_SERIALIZED

The threads are allowed to call MPI, but you must program in synchronisation to ensure only one thread calls MPI at a time.

```
!$omp parallel
  ... ! Parallel work
  !$omp critical
  call MPI_Sendrecv()
  !$omp end critical
!$omp end parallel
```

MPI_THREAD_MULTIPLE

Any thread can call MPI whenever it likes. The `MPI_THREAD_MULTIPLE` guarantees the MPI library will be OK with this.

```
!$omp parallel
  ... ! Parallel work
  call MPI_Sendrecv()
!$omp end parallel
```

Thread pinning

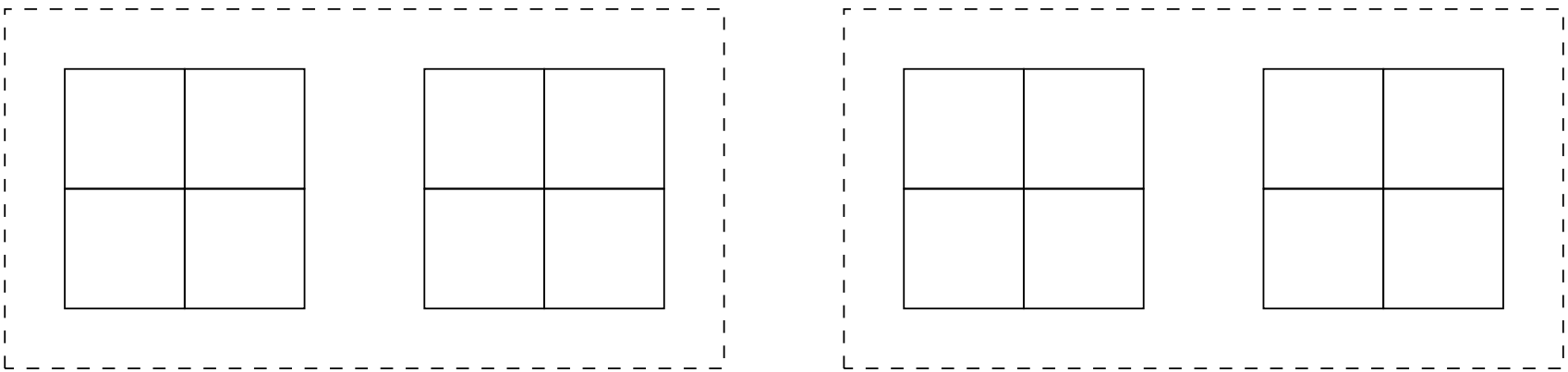
University of BRISTOL

- Need to be very careful how MPI ranks and OpenMP threads are mapped to the physical hardware.
- Imagine 2 dual-socket nodes: 4 sockets with (say) 16 cores per socket.
- Launch 64 MPI ranks: 1 per core.
  - This is flat MPI.
  - Launching OpenMP threads will over-allocate threads compared to hardware resource.
  - Warning: things will slow down.
- Launch 4 MPI ranks (one per socket).
  - Leaves 16 cores per MPI rank for OpenMP threads to run on.
  - But need to make sure processes *and* threads go to the right places!
  - Often close interaction with the queuing system — system dependant behaviour.

Example: default placement

Example MPI rank placement with standard PBS setup.
Job requested 2 nodes.

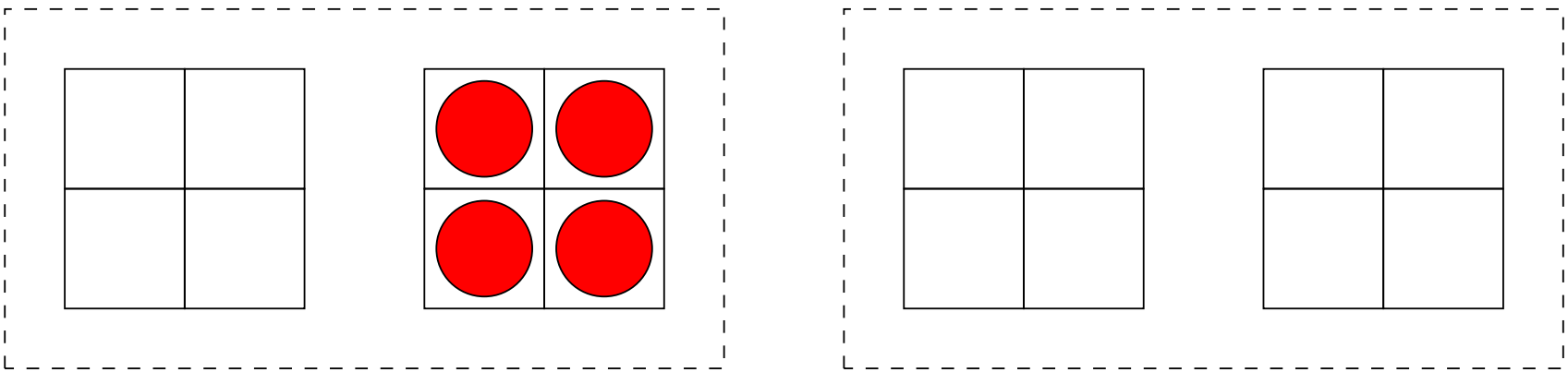
```
mpirun -np 4 ./a.out
```



Example: default placement

Example MPI rank placement with standard PBS setup.
Job requested 2 nodes.

```
mpirun -np 4 ./a.out
```



All ranks placed on the second socket of the first node.

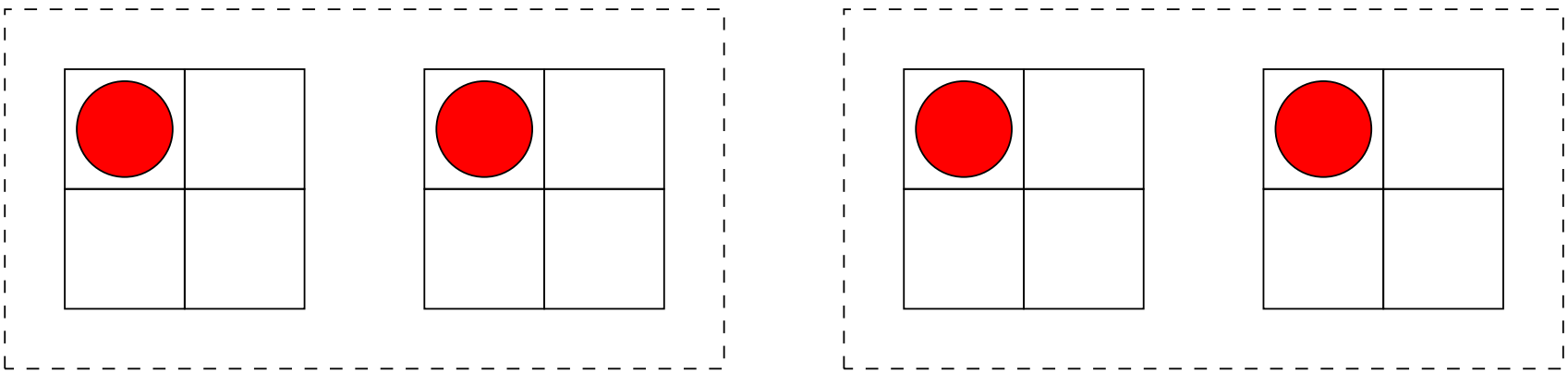
Example: pin MPI to one core per socket

University of BRISTOL

- Tell the OS and MPI runtime to pin each MPI to the first core in each socket.
- Then want to launch 4 OpenMP threads per process.
- For OpenMPI:

```
export OMP_NUM_THREADS=4
mpirun -np 4 --npersocket 1 ./a.out
```

- Where do the threads go?



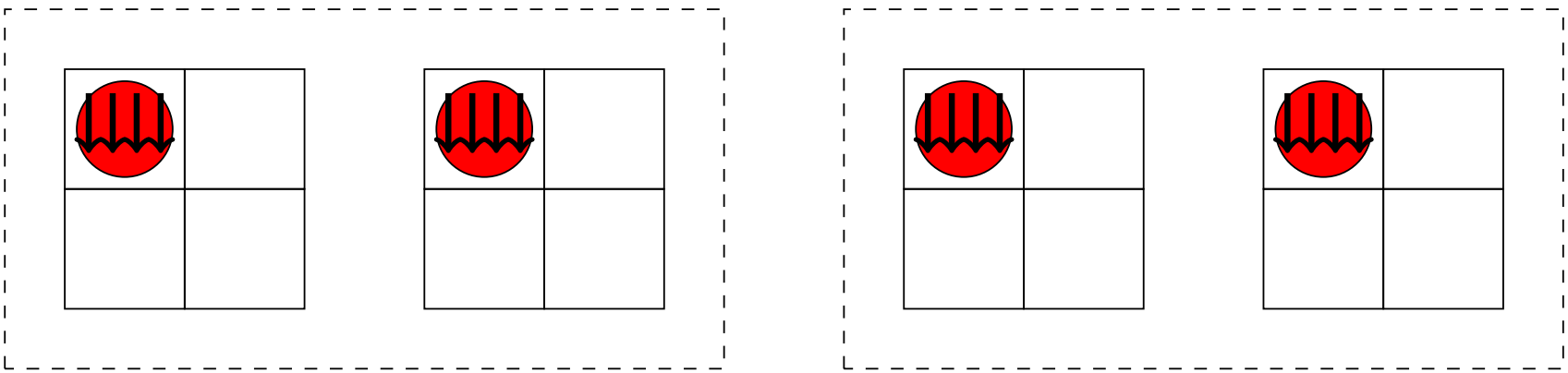
Example: pin MPI to one core per socket

University of BRISTOL

- Tell the OS and MPI runtime to pin each MPI to the first core in each socket.
- Then want to launch 4 OpenMP threads per process.
- For OpenMPI:

```
export OMP_NUM_THREADS=4
mpirun -np 4 --npersocket 1 ./a.out
```

- Where do the threads go?



Threads spawned inherit their parent's binding, which was one core.
Use `--report-bindings` flag to see what's being pinned where.

Example: pin MPI to socket

University of BRISTOL

- Pin each MPI process to the cores of a socket.
- MPI process *could* move around those cores.
- OpenMP threads can spawn across the socket.
- OpenMPI gives three ways to do this:
  - `--bind-to-socket`
  - `--bind-to-core --cpus-per-proc 8`
  - `--map-by socket:PE=8` (v1.10 and up)

Pinning with Intel and Cray

University of BRISTOL

- Intel MPI will need different flags and environment variables, but tends to do the right thing by default.
- Cray MPI (MVAPICH) can be controlled using `aprun`.
  - Use the `-d` flag to specify the threads per process.
  - Pinning usually happens correctly.
- Cray MPI with the Intel compiler needs a different set of `aprun` flags.
  - Default pinning is usually not what you expected.
  - Use the `-cc` flag to specify correct thread pinning.
- The `amask` tool from TACC is very useful for discovering the pinning[2].

[2] `https://github.com/TACC/amask`

Exercise

University of BRISTOL

- Make your parallel 5-point stencil code NUMA aware.
  - Parallelise the initialisation routine.
- Calculate improvements memory bandwidth.
  - Use a profiler to measure remote memory accesses before/after optimisation.
- Experiment with thread affinity.
- Extension: Add MPI to your OpenMP 5-point stencil to run it hybrid across multiple nodes.

Summary

University of BRISTOL

- Walked through memory bandwidth model calculation of 5-point stencil.
- NUMA issues and taking advantage of first touch policy.
- Controlling OpenMP thread affinity with `OMP_PROC_BIND` and `OMP_PLACES` environment variables.
- Programming a hybrid MPI+OpenMP code.
- Thread affinity of hybrid programs.

- Next sessions:
  5. GPU programming with OpenMP.
  6. Tasks and Tools.