ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

MPI/OpenMP Hybrid Parallelism and OpenMP Optimisation

Why MPI/OpenMP ?

Simple cases where MPI/OpenMP might help

- If you have only partially parallelised your algorithm using MPI
  - E.g. 3D Grid-based problem using domain decomposition only in 2D
    - For strong scaling this leads to long thin domains for each MPI process
    - The same number of nodes using MPI/OpenMP hybrid might scale better
  - If there are a hierarchy of elements in your problem and you have only parallelised over the outer ones
- Where your problem size is limited by memory per process on a node
  - For example you might currently have to leave some cores idle on a node in order to increase the memory available per process
- Where compute time is dominated by halo exchanges or similar communication mechanism
- When your limiting factor is scalability and you wish to reduce your time to solution by running on more processors
  - In many cases you won't be faster than an MPI application whilst it is still in its scaling range
  - You might be able to scale more than with the MPI application though …

… in these cases and more, we can take advantage of the changes in architecture of the nodes that make up our modern HPC clusters …

Preparing MPI Code for OpenMP

MPI Codes - Do We Need to Add Threading ?

- Advantages from “pure” MPI
  - No need to worry about cache-coherence and race conditions
  - You have to think about every explicit data transfer
    - You are concerned about the performance impact of every data transfer!
- Disadvantages of “pure” MPI
  - You aren’t able to take advantage of memory speed data transfers
  - You need to make calls to MPI library for all explicit copies of data
  - MPI calls can have a large overhead
- Advantages of threading
  - Reduce the communications overhead
  - Reduced memory usage due to operating system process overheads
  - Reduced memory usage due to replicated data
  - Potential to overlap communication and computation
  - Potential to get better load balance

Simple Changes to your Code and Job

- In the most simple of cases you need only change your MPI initialisation routine
  - ~~**MPI_Init**~~ is replaced by **MPI_Init_thread**
  - **MPI_Init_thread** has two additional parameters for the level of thread support required, and for the level of thread support provided by the library implementation
- You are then free to add threading (e.g. with OpenMP this would be directives and runtime calls) as long as you stick to the level of thread safety you specified in the call to MPI_Init_thread
- *Note that MPI threading support is not tied to one threading model*
  - *Support is provided for any threading which could be OpenMP, TBB etc.*

C: **int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)**

Fortran: **MPI_Init_Thread(required, provided, ierror)**
Integer : required, provided, ierror

required specifies the requested level of thread support, and the actual level of support is then returned into **provided**
You should check the value of provided after the call

The 4 Options for Thread Support

User Guarantees to the MPI Library

1. MPI_THREAD_SINGLE
   - Only one thread will execute
     - Standard MPI-only application
2. MPI_THREAD_FUNNELED
   - Only the Master Thread will make calls to the MPI library
     - The thread that calls `MPI_Init_thread` is henceforth the master thread
     - A thread can determine whether it is the master thread by a call to the routine `MPI_Is_thread_main`
3. MPI_THREAD_SERIALIZED
   - Only one thread *at a time* will make calls to the MPI library, but all threads are eligible to make such calls as long as they do not do so at the same time

The MPI Library is responsible for Thread Safety

1. MPI_THREAD_MULTIPLE
   - Any thread may call the MPI library at any time
   - The MPI library is responsible for thread safety within that library, and for any libraries that it in turn uses
   - *Codes that rely on the level of MPI_THREAD_MULTIPLE may run significantly slower than the case where one of the other options has been chosen*
   - You might need to link in a separate library in order to get this level of support (e.g. Cray MPI libraries are separate)

In most cases MPI_THREAD_FUNNELED provides the best choice for hybrid programs

Hybrid programming on Cray systems

- In order to select a thread level higher than MPI_THREAD_SINGLE on Cray systems you also need to set the environment variable MPICH_MAX_THREAD_SAFETY
  - If you do not set this variable then "provided" will return MPI_THREAD_SINGLE
- The maximum value of this variable with the default MPI library is MPI_THREAD_SERIALIZED
- If you need MPI_THREAD_MULTIPLE then you need to add "-lmpich_threadm" to your link line

Thread level	Environment setting (export/setenv)	Library
MPI_THREAD_SINGLE	export MPICH_MAX_THREAD_SAFETY=single	
MPI_THREAD_FUNNELED	export MPICH_MAX_THREAD_SAFETY=funneled	
MPI_THREAD_SERIALIZED	export MPICH_MAX_THREAD_SAFETY=serialized	
MPI_THREAD_MULTIPLE	export MPICH_MAX_THREAD_SAFETY=multiple	-lmpich_threadm

Changing Job Launch for MPI/OpenMP

- Check with your batch system how to launch a hybrid job
  - Set the correct number of processes per node and find out how to specify space for OpenMP Threads
- Set OMP_NUM_THREADS in your batch script
- You may have to repeat information on the `mpirun/aprun/srun` line
- Find out whether your job launcher has special options to enable cpu and memory affinity
  - If these are not available then it may be worth your time looking at using the Linux system call `sched_setaffinity` within your code to bind processes/threads to processor cores
  - On the Cray you should look at the “-cc” option to `aprun`

OpenMP Parallelisation Strategies

Fine-grained Loop-level Work sharing

- This is the simplest model of execution
- You introduce “!$omp parallel do” or “#pragma omp parallel for” directives in front of individual loops in order to parallelise your code
- You can then incrementally parallelise your code without worrying about the unparallelised part
  - This can help in developing bug-free code
  - … but you will normally not get good performance this way
- Beware of parallelising loops in this way unless you know where your data will reside

```
Program OMP2
Implicit None
Integer :: I
Real :: a(100), b(100), c(100)
Integer :: threadnum
!$omp parallel do private(i) shared(a,b,c)
Do i=1, 100
   a(i)=0.0
   b(i)=1.0
   c(i)=2.0
End Do
!$omp end parallel do
!$omp parallel do private(i) shared(a,b,c)
Do i=1, 100
   a(i)=b(i)+c(i)
End Do
!$omp end parallel do
Write(6,’(“I am no longer in a parallel
region”)’)
!$omp parallel do private(i) shared(a,b,c)
Do i=1,100
   c(i)=a(i)-b(i)
End Do
!$omp end parallel do
End Program OMP2
```

Coarse-grained approach

- Here you take a larger piece of code for your parallel region
- You introduce “!$omp do” or “#pragma omp for” directives in front of individual loops within your parallel region
- You deal with other pieces of code as required
  - !$omp master or !$omp single
  - Replicated work
- Requires more effort than fine-grained but is still not complicated
- Can give better performance than fine grained

```
Program OMP2
Implicit None
Integer :: I
Real :: a(100), b(100), c(100)
Integer :: threadnum
!$omp parallel private(i) shared(a,b,c)
!$omp do
Do i=1, 100
   a(i)=0.0
   b(i)=1.0
   c(i)=2.0
End Do
!$omp end do
!$omp do
Do i=1, 100
   a(i)=b(i)+c(i)
End Do
!$omp end do
!$omp master
Write(6,’(“I am **still** in a parallel region”)’)
!$omp end master
!$omp do
Do i=1,100
   c(i)=a(i)-b(i)
End Do
!$omp end do
!$omp end parallel
End Program OMP2
```

The Task Model

- More dynamic model for separate task execution
- More powerful than the SECTIONS worksharing construct
- Tasks are spawned off as “!$omp task” or “#pragma omp task” is encountered
- Threads execute tasks in an undefined order
  - You can’t rely on tasks being run in the order that you create them
- Tasks can be explicitly waited for by the use of TASKWAIT

… the task model shows good potential for overlapping computation and communication …

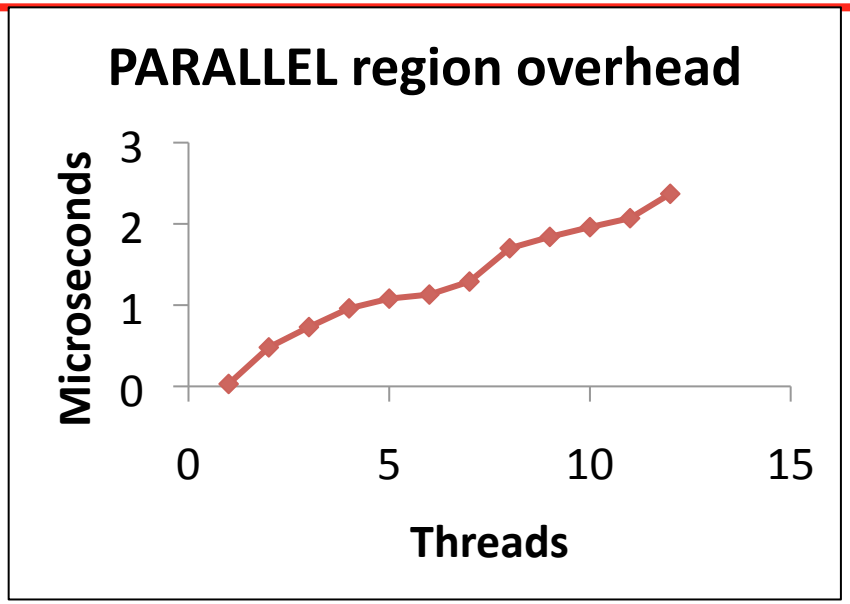
… or overlapping I/O with either of these …

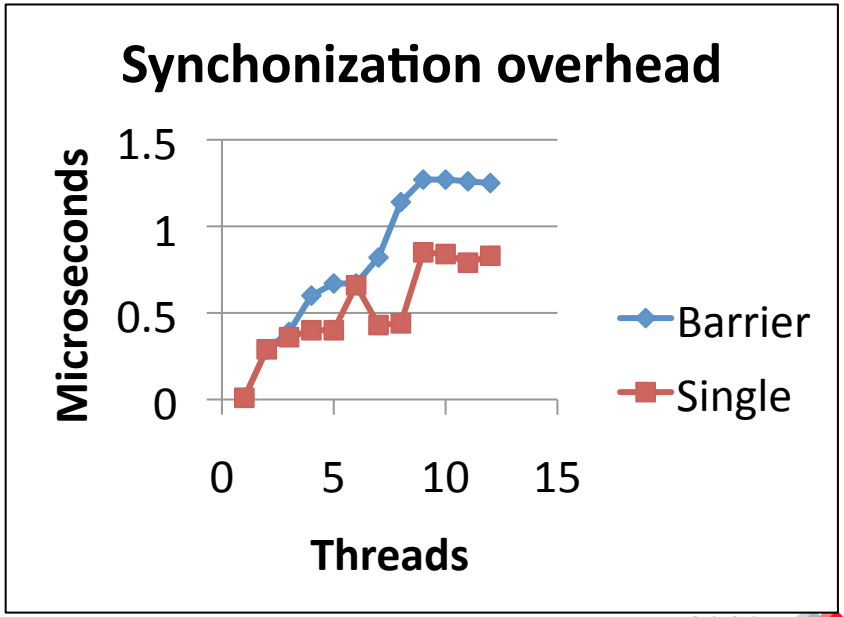
```
Program Test_task
Use omp_lib
Implicit None
Integer :: i
Integer :: count
!$omp parallel private(i)
!$omp master
Do i=1, omp_get_num_threads()+3
   !$omp task
   Write(6,'("I am a task, do you like tasks ?")')
      !$omp task
      Write(6,'("I am a subtask, you must like
me !")')
      !$omp end task
   !$omp end task
End Do
!$omp end master
!$omp end parallel
End Program Test_task
```

```
I am a task, do you like tasks ?
I am a task, do you like tasks ?
I am a task, do you like tasks ?
I am a task, do you like tasks ?
I am a subtask, you must like me !
I am a subtask, you must like me !
I am a subtask, you must like me !
I am a subtask, you must like me !
I am a task, do you like tasks ?
I am a task, do you like tasks ?
I am a task, do you like tasks ?
I am a subtask, you must like me !
I am a subtask, you must like me !
I am a subtask, you must like me !
```

Strategies for Speed

- Avoid creating too many separate parallel regions
  - Thread creation overhead is costly
- Avoid too many synchronisation points
  - Look to see where you can use `nowait`
- Investigate different scheduling strategies (static, dynamic, guided)
- Look at what loop reorderings your compiler carries out in serial optimisations
  - When you add a "!$omp do" or "#pragma omp do" directive to a loop, the compiler *must* parallelise that loop





The Dangers of OpenMP !!!

Warning - It's not as simple as it first appears…

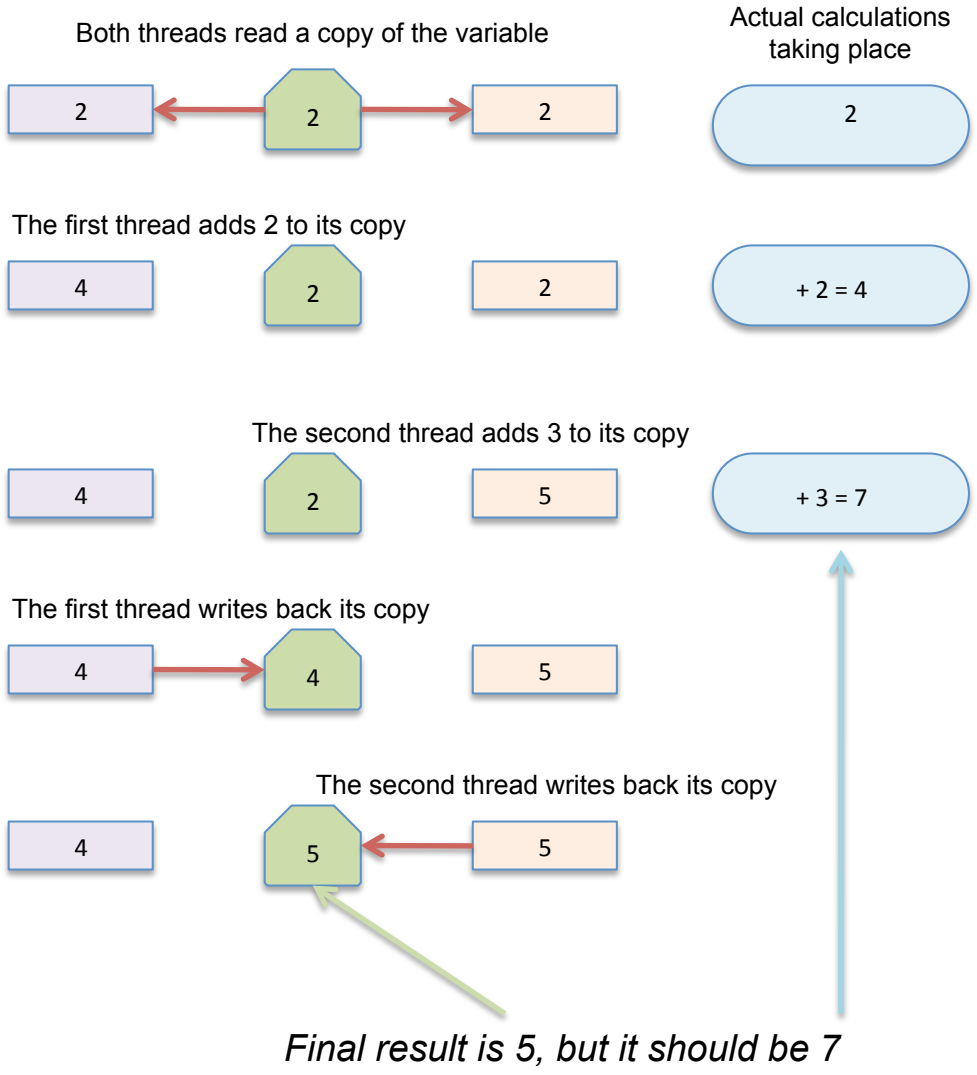
- To write an MPI code you need to work on the whole code, distributing the data and the work completely
- With OpenMP you can develop the code incrementally, but you must work on the whole application in order to get speedup
- OpenMP parallelism is not just about adding a few directives
  - You don't have to think as deeply as with MPI in order to get your code working
  - You *do* have to think as deeply as with MPI if you want to get your code *performing*
- There are many performance issues that need to be considered …

Thread creation overhead and synchronisation

- Creation and destruction of threads is an overhead that takes time
- In theory each entry and exit of a parallel region could lead to thread creation and destruction
  - in most OpenMP implementations threads are not destroyed at the end of a parallel region but are merely put to sleep
- In any case, entering and exiting a parallel region requires barriers to be called between a team of threads
  - This is often what is referred to as thread creation/destruction overhead
- Staying within a parallel region, and having multiple worksharing constructs within it reduces the overhead associated with entering and exiting parallel regions
- The best performance might be produced by duplicating work across multiple threads for some trivial activities
  - You would probably do this duplication in MPI as well in most cases, rather than have one process calculate a value and then issue a MPI_Bcast
- For best performance avoid unnecessary synchronisation and consider using NOWAIT with DO/for loops wherever possible

A common problem – race conditions

- Race conditions occur when two threads both want to update the same piece of data
  - Related to cache coherency, but this time it's dangerous
  - Is concerned with data read into processor registers
    - Once data is in a register it can no longer be looked after by cache coherency protocols
- One thread reads in a piece of data and updates it in one of its registers
- The second thread reads the data and updates it in one of its registers
- The first thread writes back the new data
- The second thread writes back its new data
- Both updates have not been accounted for !!!!
- In OpenMP, you need to use the `atomic` or `critical` directives wherever there is a risk of a race condition



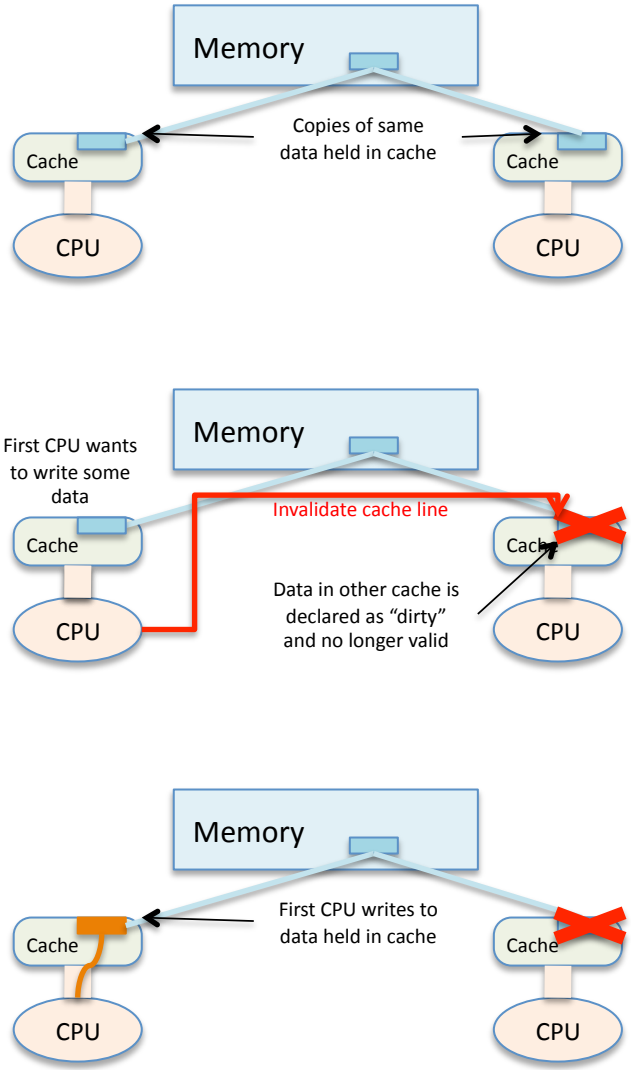
Cache Coherence

- As caches are local copies of global memory, multiple cores can hold a copy of the same data in their caches
  - For separate MPI processes with distinct memory address spaces multiple cores are not likely to hold copies of the same user data
    - Unless data is copied during process migration from one core to another
      - This can be avoided by using cpu affinity
  - For OpenMP codes where multiple threads share the same address space this could lead to problems
- Before accessing memory, a processor core will check its own cache *and* the cache of the other socket to ensure consistency between cache and memory
  - This is referred to as **cache-coherency**
- Ensuring that a node is cache coherent does not mean that problems associated with multiple copies of data are completely removed
  - Data held in processor registers are not covered by coherence
    - This lack of coherence can lead to a *race condition*

Cache coherence amongst multiple cores

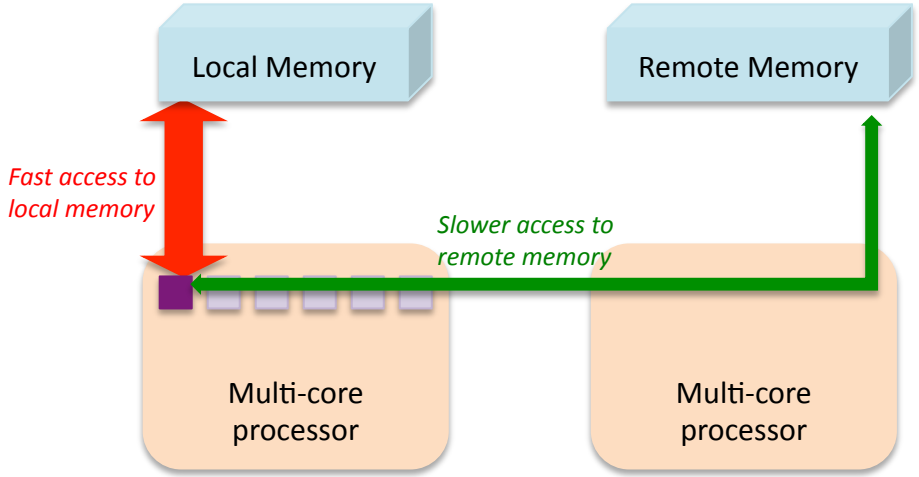
Example with two single-core sockets

- Assume a model with two processors each with one level of cache
- Both processors have taken a copy of the same data from main memory
- If one of them wants to write to this data, then the local copy will be affected, but the main memory does not change
  - The reason for having fast cache is to avoid slower main memory accesses
- On a **cache coherent** system, the rest of the node needs to be told about the update
  - The other processor's cache needs to be told that its data is "bad" and that it needs a fresh copy
  - On a multi-processor system other cores need to be aware that main memory is now "tainted" and does not have the most up-to-date copy of this data
  - Then the new data can be written into the local copy held in cache
- If CPU 2 wishes to read from or write to the data it needs to get a fresh copy



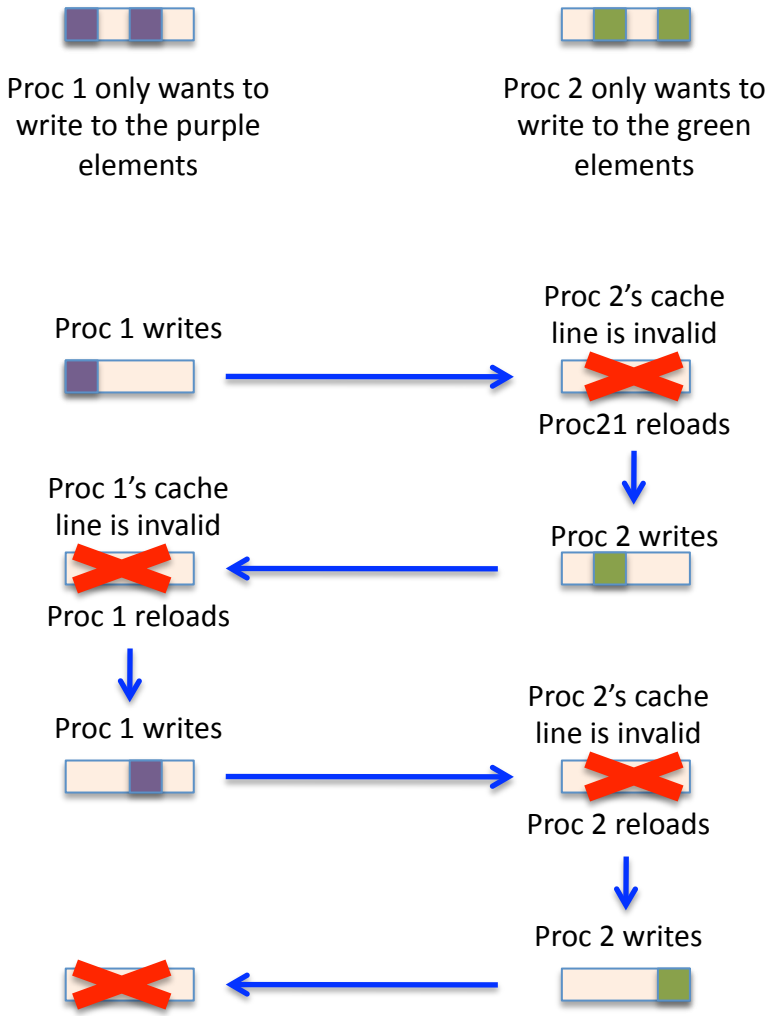
Cache-coherent NUMA node

- Each compute node typically has several gigabytes of memory directly attached
- An processor core can access the other socket's memory by crossing the inter-socket link between the two processors
- Accessing remote memory is slower than accessing local memory, so this is referred to as a **Non-uniform memory access** (**NUMA**) node
- A node that has NUMA characteristics and that guarantees cache-coherence is referred to as a **cache-coherent NUMA node**



Contention - Cache Thrashing, False Sharing

- Cache coherency protocols update data based on cache lines
- Even if two threads want to write to different data elements in an array, if they share the same cache line then cache coherency protocols will mark the other cache line as dirty
- In the best case false sharing leads to serialisation of work
- In the worst case it can lead to *slowdown* of parallel code compared to the serial version

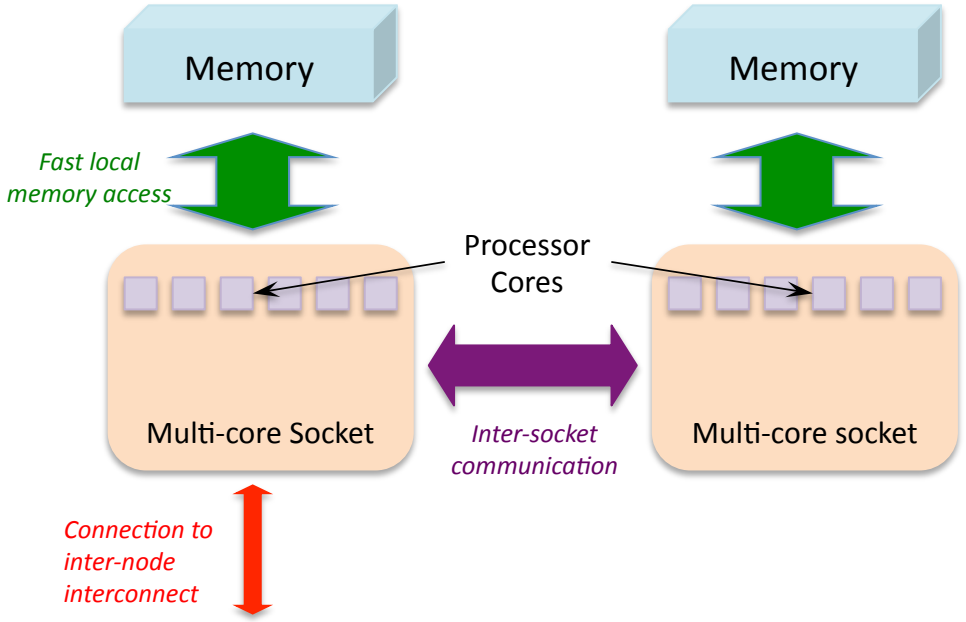


Multi-socket, multi-core nodes

NUMA – Non-Uniform Memory Access

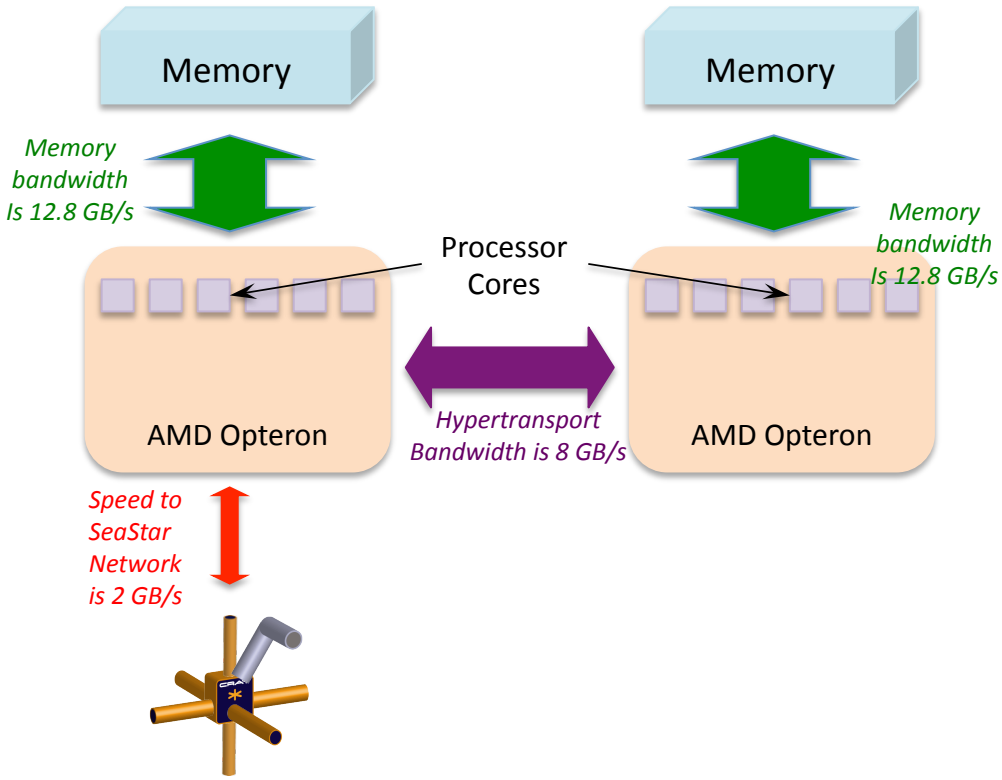
Example two-socket node

- Local memory accesses have higher bandwidth and lower latency than remote accesses
- If all memory accesses from all cores are to one memory then the effective memory bandwidth is reduced across all processes/threads
- If all accesses are to remote memory then “memory bandwidth” will actually be dominated by inter-socket bandwidth



Specific Example – A Cray XT5 node

- The memory bandwidth to local memory is 12.8 GB/s
  - If all accesses are to local memory then theoretical peak node bandwidth is 25.6 GB/s
- The inter-socket bandwidth is 8 GB/s
  - If all memory accesses are to remote memory then theoretical peak node bandwidth is reduced to 8 GB/s
- The latency to local memory is also much lower than to remote memory



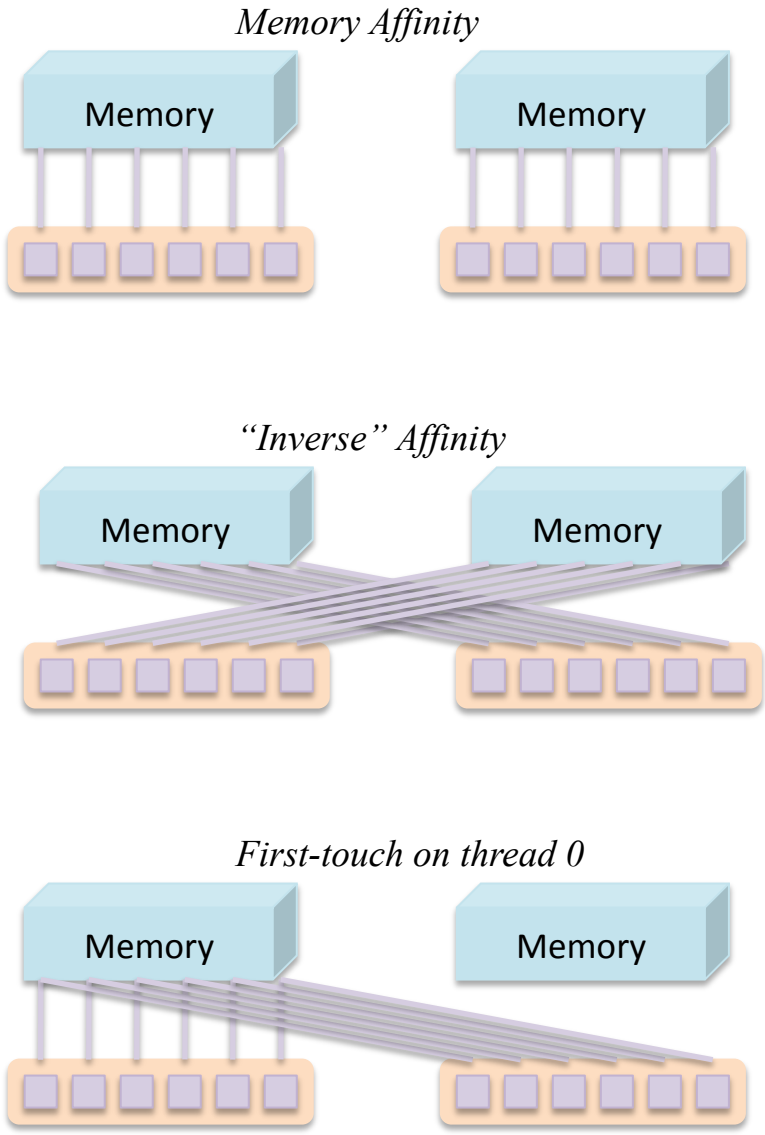
- For a Cray XE6 node this becomes 4 NUMA entities per node
  - There are 2 sockets, each of which has 2 NUMA nodes within it

Operating system memory allocation - affinity

- CPU affinity is the pinning of a process or thread to a particular core
  - If the operating system interrupts the task, it doesn't migrate it to another core, but waits until the core is free again
    - For most HPC scenarios where only one application is running on a node, these interruptions are short
- Memory affinity is the allocation of memory as close as possible to the core on which the task that requested the memory is running
  - It is not actually the allocation but the touching of the memory (reading/writing) that is important
    - This is referred to as a "first touch" policy
- Both CPU affinity and memory affinity are important if we are to maximise memory bandwidth on NUMA nodes
  - If memory affinity is not enabled then bandwidth will be reduced as we go off-socket to access remote memory
  - If CPU affinity is not enabled then allocating memory locally is of no use when the task that requested the memory might no longer be running on the same socket
- By default a NUMA-aware Linux kernel will try to use a "first touch" policy for allocating memory
- Tools and libraries are available to enforce CPU affinity
  - Some batch job launchers such as Slurm's `srun` and Cray's `aprun` can use CPU affinity by default
  - OpenMP has support for CPU affinity

Memory affinity (or not!) examples

- To demonstrate memory bandwidth changes we use the stream benchmark with 3 memory allocation policies
- First the default with cpu and memory affinity and each OpenMP thread using first-touch on its own memory
- Second we use a criss-cross pattern where a thread on a different socket touches the data to have it allocated on the remote memory
- Third we use the method where all of the memory is first touched by the master thread
  - For datasets that can fit in the local memory of one socket all the data will be allocated together
  - *Many people who are implementing OpenMP in their code do not take the trouble to put OpenMP directives into the routines where memory is first touched*
    - Be sure to put OpenMP directives around your first-touch routines before you out any other directives into your code – otherwise you might complain about the poor performance!



Benchmark data – AMD Magny-Cours

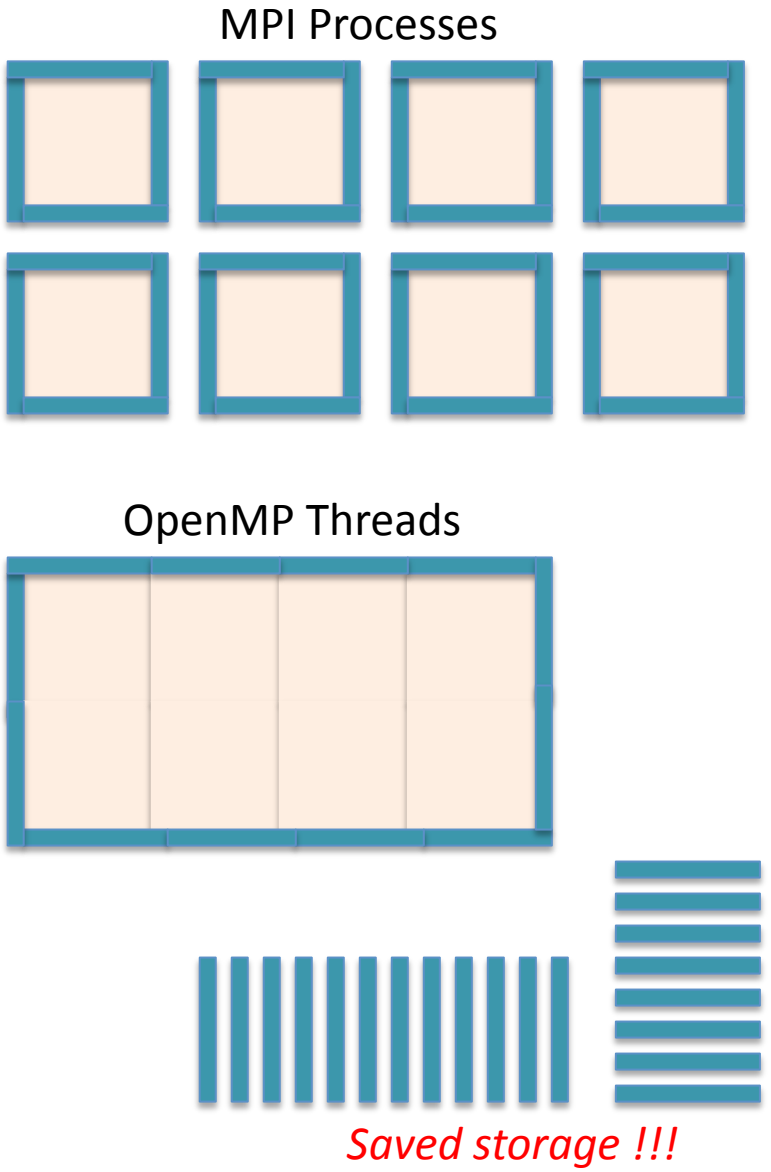
- AMD Opteron Magny-Cours 2-socket x 12-core processors running at 2.1 GHz
  - This was the *old (two cabinet)* Cray XE6 at CSCS
  - Cray's "aprun" will enforce thread cpu affinity by default
  - The hardware of the Magny-Cours processors meant that this was effectively 4-sockets x 6-cores

Numbers are aggregate bandwidth in GB/s	Memory Affinity	"Inverse" Affinity	Thread 0 First-touch
Copy	47.5	19.3	10.4
Scale	33.6	15.2	6.5
Add	36.5	15.4	6.8
Triad	36.5	15.4	6.8

- For the thread-0 first-touch, the memory from only one of the 4 mini-sockets is available, meaning that only one quarter of the real memory bandwidth is available !!

Halo regions and replicated data in MPI

- Halo regions are local copies of remote data that are needed for computations
  - Halo regions need to be copied fequently
- Using OpenMP parallelism reduces the size of halos region copies that need to be stored
- Other data structures than these might also lead to a benefit of MPI/OpenMP applications from reduced memory requirements
- Reducing halo region sizes also reduces communication requirements



Amdahl's law only tells part of the tale

- Amdahl's law for strong scaling
  - This states the ultimate limiting factor of parallel scaling is the part that cannot be parallelised
  - It only looks at parts of an application being either perfectly parallelisable or serial
- In reality scaling is a complex mix of components including
  - Computation: this is the part where we are trying to get linear scaling
    - Might already be efficiently parallelised in MPI code
  - Memory bandwidth limitations: the proportion of data that might need to be read for a given set of computations might increase with decreasing workload per task,
  - Communications: the amount of communication might not decrease linearly with the workload per task
    - OpenMP might be able to reduce this problem
  - Parallel processing overheads: some communication overheads may be fixed or not decrease significantly with decreasing workload per task
    - Some communications might be reduced here
  - I/O and other serial parts of the code