ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

DAY 2: Parallel Programming with MPI and OpenMP

Preparing MPI Code for OpenMP

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 OpenMP directives and runtime calls as long as you stick to the level of thread safety you specified in the call to MPI_Init_thread

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**

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`

What architecture are we targeting ?

Processors have become Multi-core

- HPC Processors
  - IBM Power7 processor – 8 cores per processor
  - Intel Nehalem-EX – 8 cores per processor
  - AMD Magny-Cours – 12 cores per processor
    - Actually 2 x 6-core processors internally
    - Next generation Interlagos will be 16 cores per processor
  - IBM BlueGene/Q – will have 16 cores per processor
  - Fujitsu Sparc VIIIfx (for RIKEN next generation supercomputer [NGSC]) – will have 8 cores per processor

Some processors have become Multi-threaded

- HPC Processors
  - IBM Power7 processor – 8 cores per processor
    - ***Power7 supports up to 4 hardware threads***
  - Intel Nehalem-EX – 8 cores per processor
    - ***Intel Nehalem processors support 2 hardware threads***
  - AMD Magny-Cours – 12 cores per processor
    - Actually 2 x 6-core processors internally
    - Next generation Interlagos will be 16 cores per processor
      - Actually it will be 8 modules, each of which have 2 cores ***that share the floating point SIMD unit***
  - IBM BlueGene/Q – will have 16 cores per processor
    - ***BG/Q PowerPC processor supports up to 4 hardware threads***
  - Fujitsu Sparc VIIIfx (for RIKEN next generation supercomputer [NGSC]) – will have 8 cores per processor
- Hardware support for multi-threading (SMT – symmetric multi-threading) doesn't add any functional units to the processor
  - It allows the processor to do useful work if on thread is stalled due to some I/O
  - It allows better hiding of memory latency
- Note that hardware *threads* are not the same as operating system threads, each hardware *thread* could be a separate process

Many nodes have become Multi-core, multi-socket

- HPC Nodes
  - BlueGene/Q node will be one multi-core processor
  - Fujitsu RIKEN NGSC node will be one multi-core processor
  - Some nodes that are made up of a multi-core chip and accelerator may have only one multi-core x86 processor
  - *All other nodes are likely to be multi-socket and multi-core*
  - *Cray XE6 nodes are multi-core, multi-socket*

OpenMP Parallelisation Strategies

Examples of pure OpenMP

- Introduction yesterday given by Matthew Cordery showed a taster of OpenMP
- The best place to look is the OpenMP 3.0 specification!
  - Contains hundreds of examples
  - Available from http://www.openmp.org/

The basics of running a parallel OpenMP job

- The simplest form of parallelism in OpenMP is to introduce a parallel region
- Parallel regions are initiated using "`!$omp Parallel`" or "`#pragma omp parallel`"
- All code within a parallel region is executed unless other work sharing or tasking constructs are encountered
- Once you have changed your code, you simply need to
  - Compile the code
    - Check your compiler for what flags you need (if any) to recognise OpenMP directives
  - Set the OMP_NUM_THREADS environment variable
  - Run your application

```
Program OMP1
Use omp_lib
Implicit None
Integer :: threadnum
!$omp parallel
Write(6,'("I am thread num ",I3)') &
  & omp_get_thread_num()
!$omp end parallel
End Program OMP1
```

```
I am thread num   1
I am thread num   0
I am thread num   2
I am thread num   3
```

Parallel regions and shared or private data

- For anything more simple than “Hello World” you need to give consideration to whether data is to be *private* or *shared* within a parallel region
- The declaration of data visibility is done when the parallel region is declared
- Private data can only be viewed by one thread and is undefined upon entry to a parallel region
  - Typically temporary and scratch variables will be private
  - Loop counters need to be private
- Shared data can be viewed by all threads
  - Most data in your program will typically be shared if you are using parallel work sharing constructs at anything other than the very highest level
- There are other options for data such as *firstprivate* and *lastprivate* as well as declarations for *threadprivate* copies of data and *reduction* variables

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
```

Other work sharing options

- The COLLAPSE addition to the DO/for directive allows you to improve load balancing by collapsing loop iterations from multiple loops
  - Normally a DO/for directive only applies to the immediately following loop
  - With collapse you specify how many loops in a nest you wish to collapse
  - Pay attention to memory locality issues with your collapsed loops
- Fortran has a set a WORKSHARE construct that allows you to parallelise over parts of your code written using Fortran90 array syntax
  - In practice these are rarely used as most Fortran programmers still use Do loops
- For a fixed number of set tasks it is possible to use the SECTIONS constructs
  - The fact that a set number of sections are defined in such a region makes this too restrictive for most people
- Nested loop parallelism
  - Some people have shown success with nested parallelism
  - The collapse clause can be used in some circumstances where nested loop parallelism appeared to be attractive

Collapse Clause

```
Program Test_collapse
Use omp_lib
Implicit None
Integer, Parameter :: wp=Kind(0.0D0)
Integer, Parameter :: arr_size=1000
Real(wp), Dimension(:,:), Allocatable :: a, b, c
Integer :: i, j, k
Integer :: count
Allocate(a(arr_size,arr_size),b(arr_size,arr_size),&
     &c(arr_size,arr_size))
a=0.0_wp
b=0.0_wp
c=0.0_wp
!$omp parallel private(i,j,k) shared(a,b,c) private(count)
count=0
!$omp do
Do i=1, omp_get_num_threads()+1
   Do j=1,arr_size
      Do k=1,arr_size
         c(i,j)=c(i,j)+a(i,k)*b(k,j)
         count=count+1
      End Do
   End Do
End Do
!$omp end do
!$ print *, "I am thread ",omp_get_thread_num()," and I did
",count," iterations"
!$omp end parallel
Write(6,'("Final val = ",E15.8)') c(arr_size,arr_size)
End Program Test_collapse
```

```
Program Test_collapse
Use omp_lib
Implicit None
Integer, Parameter :: wp=Kind(0.0D0)
Integer, Parameter :: arr_size=1000
Real(wp), Dimension(:,:), Allocatable :: a, b, c
Integer :: i, j, k
Integer :: count
Allocate(a(arr_size,arr_size),b(arr_size,arr_size),&
     &c(arr_size,arr_size))
a=0.0_wp
b=0.0_wp
c=0.0_wp
!$omp parallel private(i,j,k) shared(a,b,c) private(count)
count=0
!$omp do collapse(3)
Do i=1, omp_get_num_threads()+1
   Do j=1,arr_size
      Do k=1,arr_size
         c(i,j)=c(i,j)+a(i,k)*b(k,j)
         count=count+1
      End Do
   End Do
End Do
!$omp end do
!$ print *, "I am thread ",omp_get_thread_num()," and I did
",count," iterations"
!$omp end parallel
Write(6,'("Final val = ",E15.8)') c(arr_size,arr_size)
End Program Test_collapse
```

```
I am thread 0  and I did 2000000  iterations
I am thread 2  and I did 1000000  iterations
I am thread 1  and I did 2000000  iterations
I am thread 3  and I did 0  iterations
Final val =  0.00000000E+00
```

```
I am thread 2  and I did 1250000  iterations
I am thread 1  and I did 1250000  iterations
I am thread 0  and I did 1250000  iterations
I am thread 3  and I did 1250000  iterations
Final val =  0.00000000E+00
```

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 …

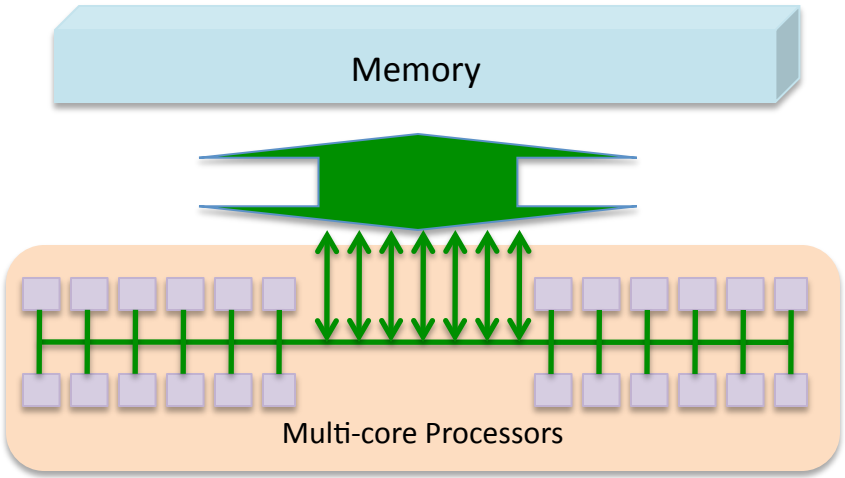
```
Program Test_task
Use omp_lib
Implicit None
Integer :: i
Integer :: count
!$omp parallel private(i)
!$omp master
Do i=1, omp_get_num_threads()+3
   !$omp task
   Write(6,'("I am a task, do you like tasks ?")')
      !$omp task
      Write(6,'("I am a subtask, you must like
me !")')
      !$omp end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

Multi-socket, multi-core nodes

Architecture of a Multi-core Multi-socket Node

- A multi-socket node consists of a number of multi-core processors and a global memory that all processors can access
- From an application point of view a single process or thread sees the memory and interconnect as shared resources
- In order to allocate memory, a single thread doesn't need to know where the memory is coming from
- For a single-socket multi-core node this should present no problems
  - But some single-socket processors are actually multiple processors fused
    - E.g. AMD Magny-Cours

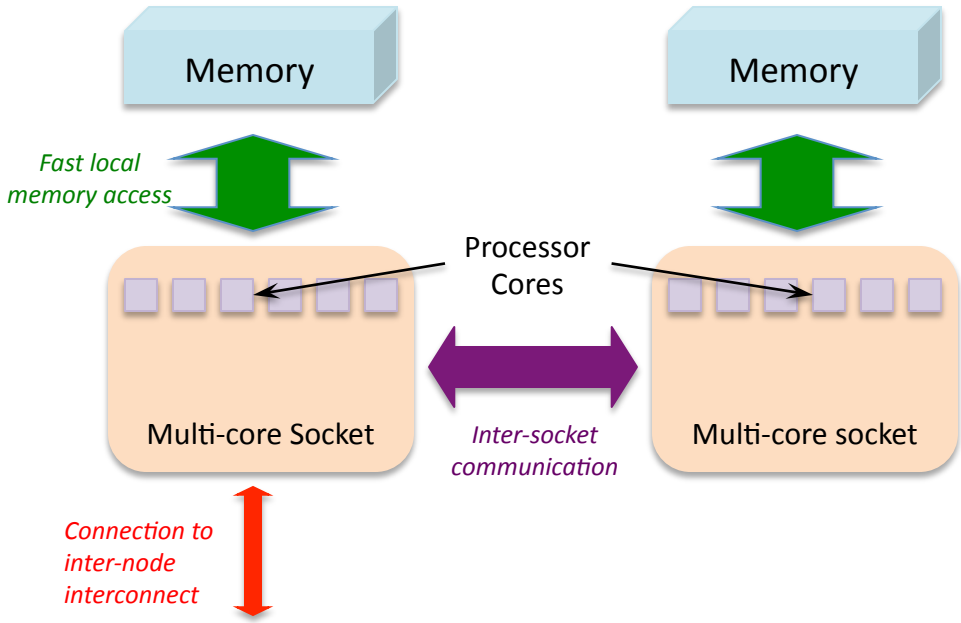


For multi-socket multi-core nodes, we cannot treat the memory in this way

NUMA – Non-Uniform Memory Access

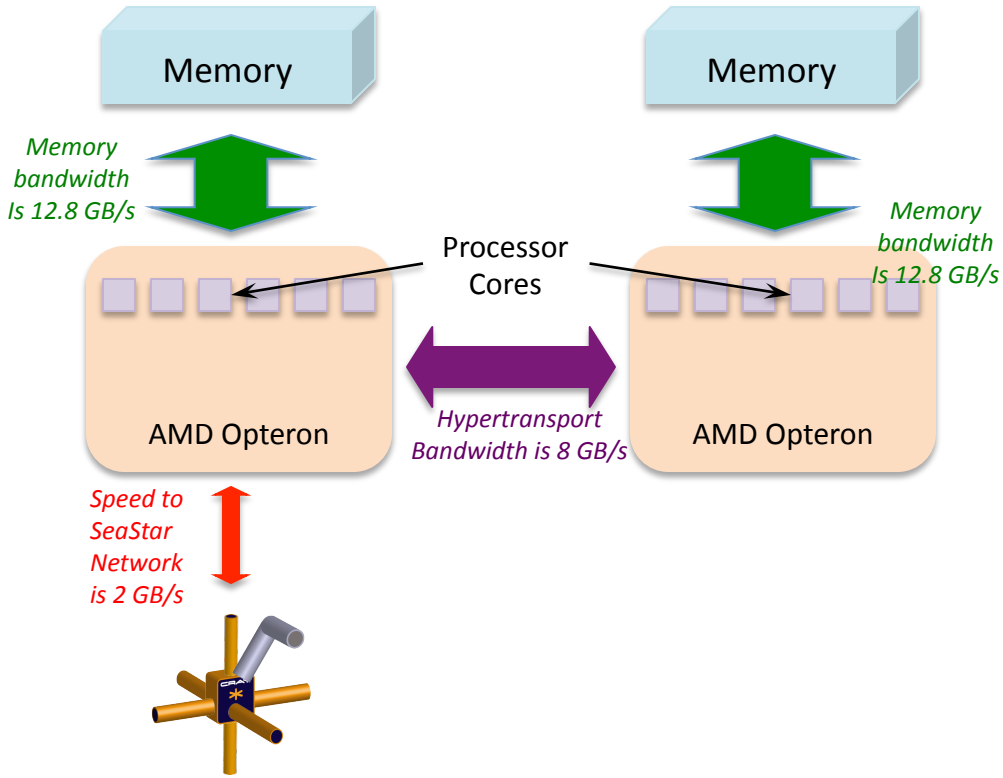
Example two-socket node

- 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

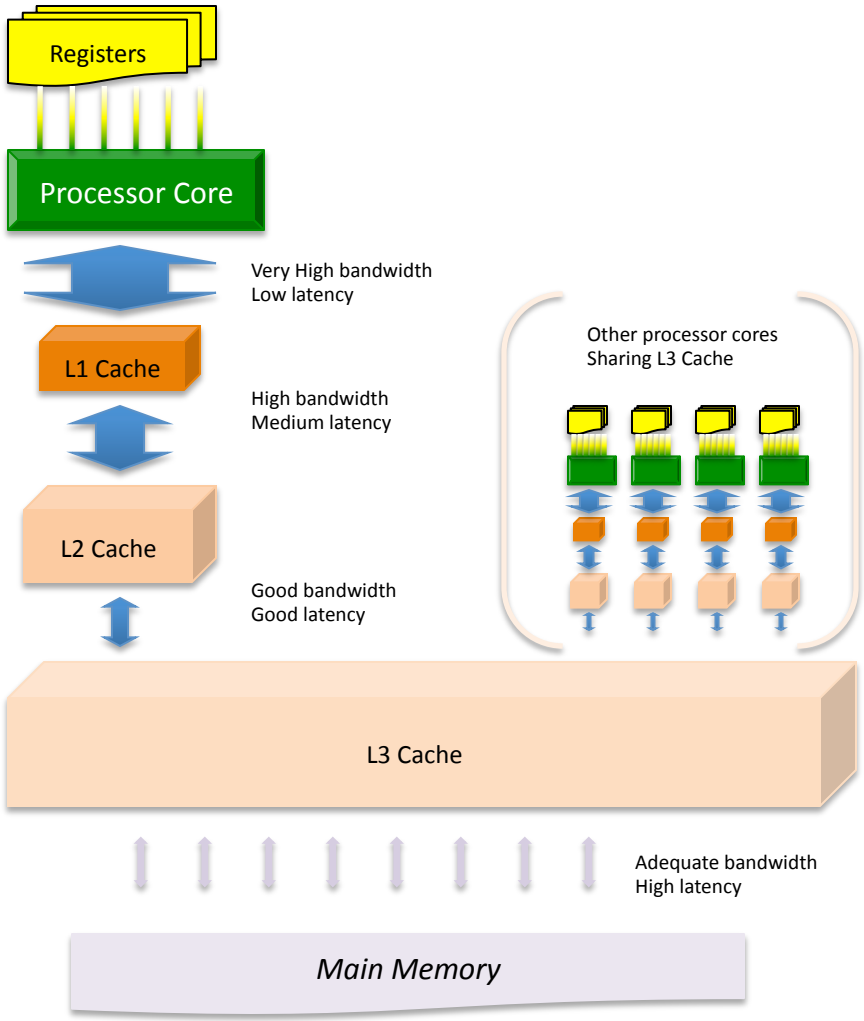
- The memory bandwidth to local memory is 12.8 GB/s
  - If all accesses are to local memory then theoretical peak node bandwidth is 25.6 GB/s
- The inter-socket bandwidth is 8 GB/s
  - If all memory accesses are to remote memory then theoretical peak node bandwidth is reduced to 8 GB/s
- The latency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

Caches Hierarchies and Locality

- Since accessing main memory is slow, modern processors provide fast local memory (**cache**) to speed up memory accesses
  - Caches are only effective for data that is being reused
- Data that has been used recently may have a high likelihood of being used again (**temporal locality**)
  - Recently used data sits in the cache in case it is required again soon
- Data is fetched from main memory to the cache in blocks called *cache lines* as there is a high likelihood that data nearby will be used together (**spatial locality**)
  - Often an algorithm will step through adjacent locations in memory
- There may be multiple levels of cache, each with different characteristics
  - Most modern processors have 3 levels of cache
  - Third level cache (L3 cache) is often shared amongst several processor cores



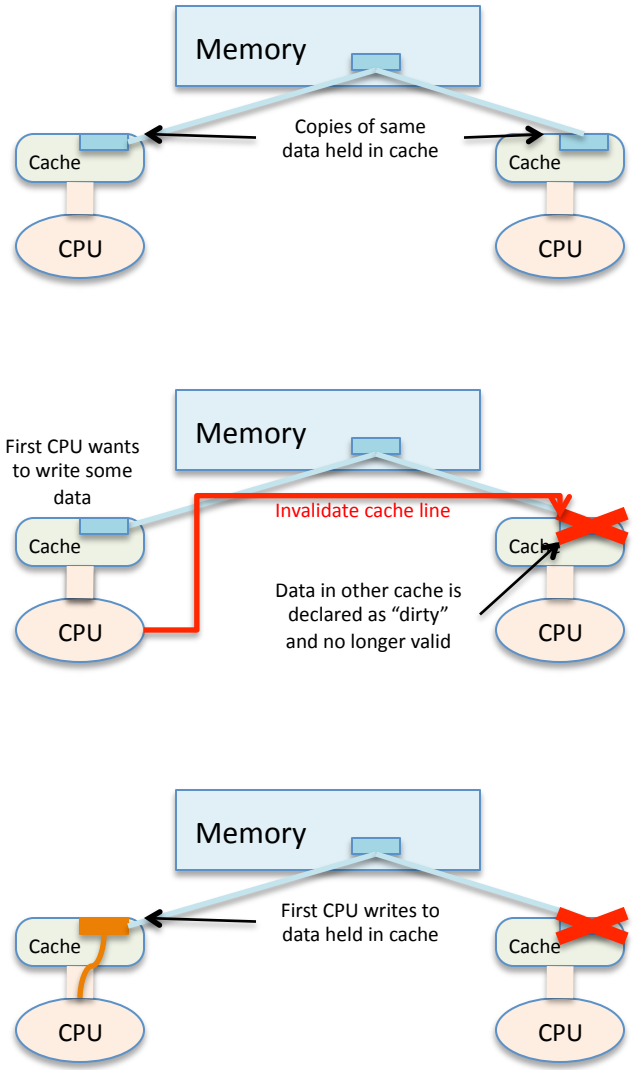
Cache Coherence

- 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

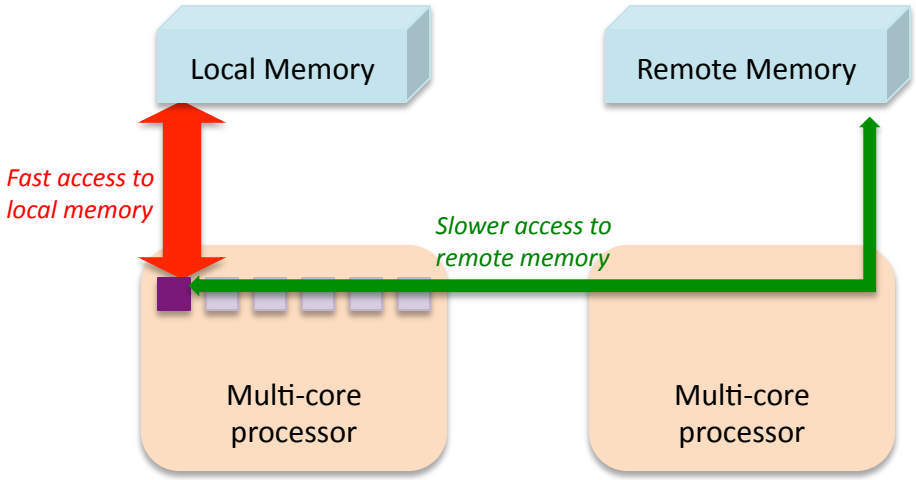
Example with two single-core sockets

- 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**



Super-scalar out-of-order pipelined with SIMD

- Most modern server processor core can issue multiple instructions per clock cycle such as a load, a floating point instruction and an integer operation
  - A **super-scalar** processor issues multiple instructions per cycle
- Most modern server processor cores can issue instructions **out-of-order** if the next instruction in line to be issued is stalled
  - E.g. If you are waiting to load variable *A* from memory, and the processor wants to use *A* for a floating-point calculation, then the processor will issue subsequent instructions if there are no dependencies
- Typically an instruction such as floating-point multiply will take several clock cycles to complete. A **pipelined** processor can feed new data into the floating-point unit each clock cycle rather than stalling on completion of each operation
  - This is like feeding new material into a factory production line
- The SIMD unit (SSE on x86 cores or AltiVec/"double Hummer" on Power processor cores) provide small vector units for enhanced floating point performance
  - If you are not using the SIMD units then you will get worse performance

Do We Need to Add Threading as Well ?

- 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

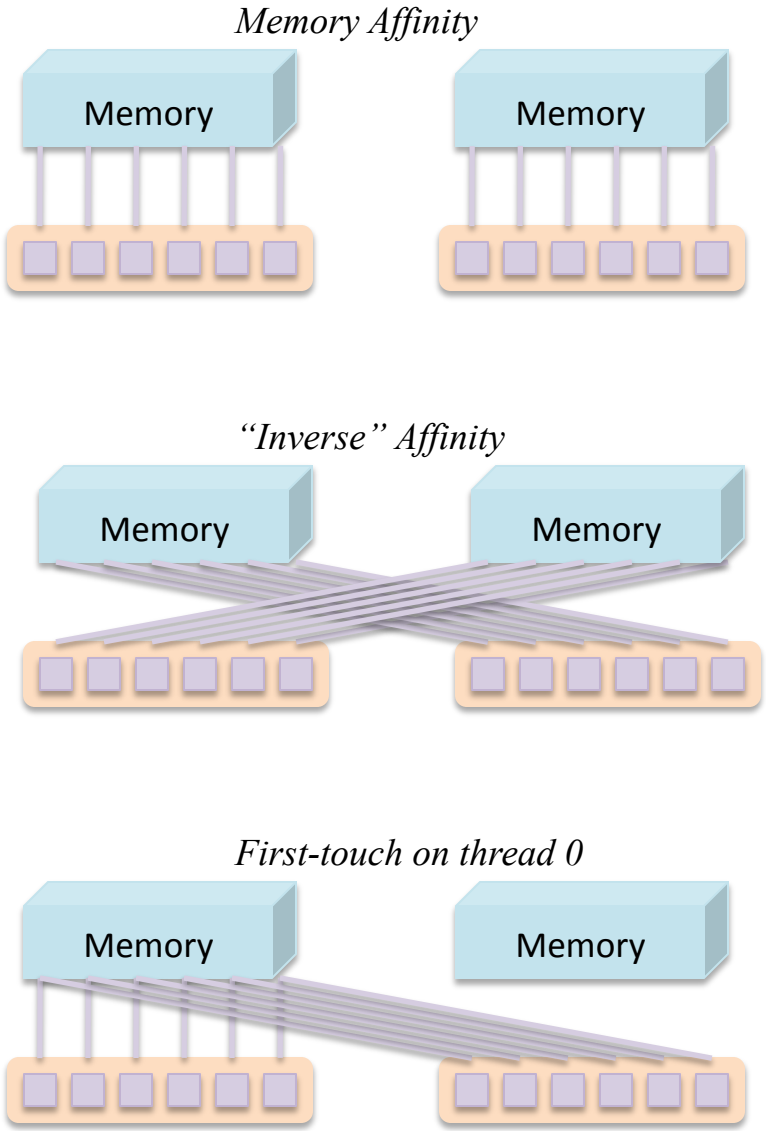
The Linux operating system

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 bandwidth change

- 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 – Intel Nehalem

- Intel Nehalem-EX 2-socket x 6-core processors running at 2.0 GHz
  - Machine in early testing, was not possible to use tools to obtain CPU affinity
    - Relying on Linux default to place threads appropriately
    - Benchmarks were run many, many, many, many, many, many, many, many, many … times and best numbers taken in each case

Numbers are aggregate bandwidth in GB/s	Memory Affinity	"Inverse" Affinity	Thread 0 First-touch
Copy	19.4	15.9	13.4
Scale	19.4	15.9	13.4
Add	24.8	19.7	16.2
Triad	24.8	19.7	16.2

Benchmark data – AMD Istanbul

- AMD Opteron Istanbul 2-socket x 6-core processors running at 2.6 GHz
  - This is a small version of the Cray XT5 at CSCS
  - Cray's "aprun" will enforce thread cpu affinity by default

Numbers are aggregate bandwidth in GB/s	Memory Affinity	"Inverse" Affinity	Thread 0 First-touch
Copy	20.1	6.9	9.5
Scale	13.4	6.7	6.7
Add	14.6	7.3	7.3
Triad	14.7	7.3	7.3

Benchmark data – AMD Magny-Cours

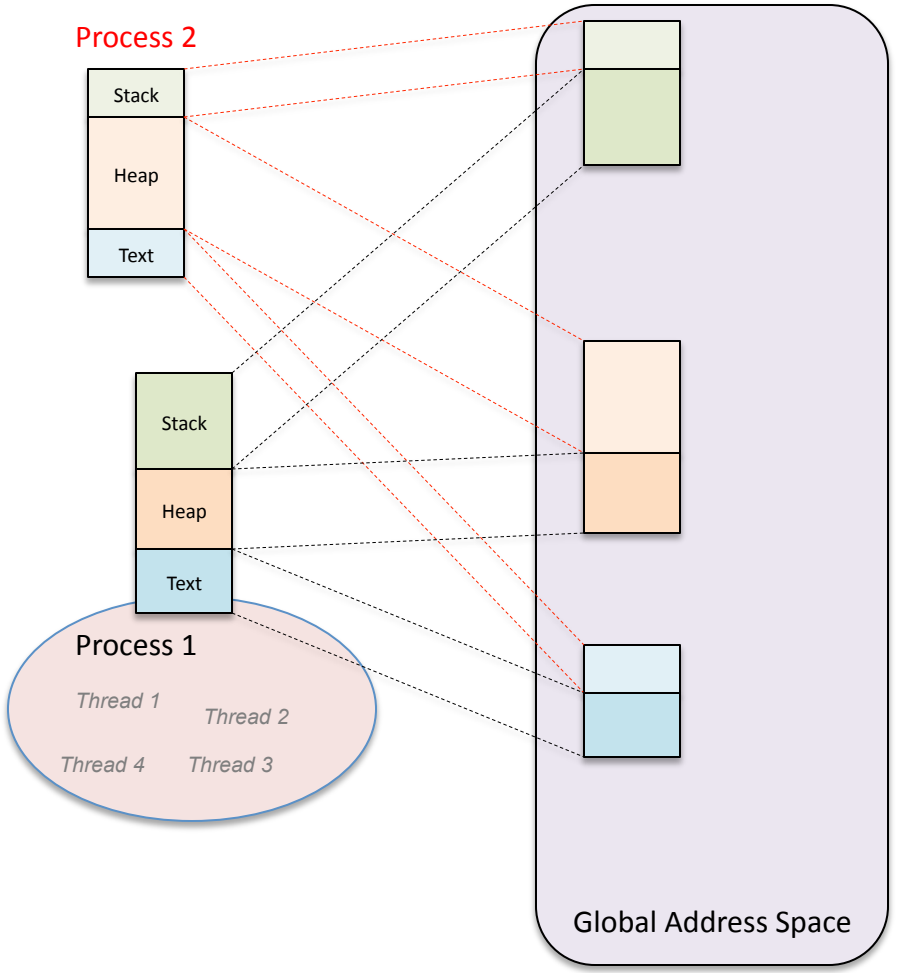
- AMD Opteron Magny-Cours 2-socket x 12-core processors running at 2.1 GHz
  - This is the Cray XE6 at CSCS
  - Cray's "aprun" will enforce thread cpu affinity by default
  - The hardware of the Magny-Cours means that this is effectively 4-sockets x 6-cores

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 !!

Operating System Separation of Processes

- Operating systems provide a separate address space for each process
- One process cannot see the memory of another process
- Need to use kernel level routines to enable message passing
  - This will typically involve multiple copies of data being taken
  - For on-node communication this means unnecessary waste of memory bandwidth
- Multiple threads launched from the same thread share the same address space



Communication Mechanism for Message Passing

- In order for *on-node* communication to take place between two communicating processes, the message may need to be **buffered**
  - There might be multiple memory copies needed in order to transfer data
- Since on-node communication is effectively just a memory copy between MPI processes, any extra buffering will consume memory bandwidth and slow down the communication
- Some libraries exist to minimise the transfers by taking advantage of special kernel features
  - E.g. XPMEM developed for SGI and now used by Cray
  - KNEM to be exploited by OpenMPI 1.5
- OpenMP threads are able to avoid these problems by directly reading memory on the same node

Copy example MPI processes vs. OpenMP Threads

- Speed of simple MPI example vs. simple OpenMP example
- We use two kernels that do the same thing … copy a piece of data from one process/thread to another on different sockets of the same system
  - The OpenMP implementation is about 2-4 times faster on an AMD Istanbul and Magny-cours based Cray system

```
If(my_rank<half_world)Then
   neighbour=my_rank+half_world
Else
   neighbour=my_rank-half_world
End If
…
Do i=1,full_arr_size
   recvarray(i,my_rank)=sendarray(i,neighbour)
End Do
```

```
If(my_rank<half_world)Then
   neighbour=my_rank+half_world
Else
   neighbour=my_rank-half_world
End If
…
Call MPI_Sendrecv(sendarray,full_arr_size,MPI_DOUBLE_PRECISION,neighbour,msg_tag,&
                  &recvarray,full_arr_size,MPI_DOUBLE_PRECISION,neighbour,msg_tag,&
                  &MPI_COMM_WORLD,status,ierror)
```

Process Memory Model

- The memory of an individual process consists of sections of space for data, text, heap and stack
  - Each of these is a separate mapping that consumes valuable memory
- Separate MPI processes will require buffer space allocated for MPI communications
  - These may be configurable through environment variables, but some space will be needed for each type
- In addition there are some MPI memory requirements which grow with the number of MPI ranks
  - An implementation that exhibits such behaviour is ultimately not scalable
- Other libraries that might be used by your application might require extra buffer space to be allocated for them

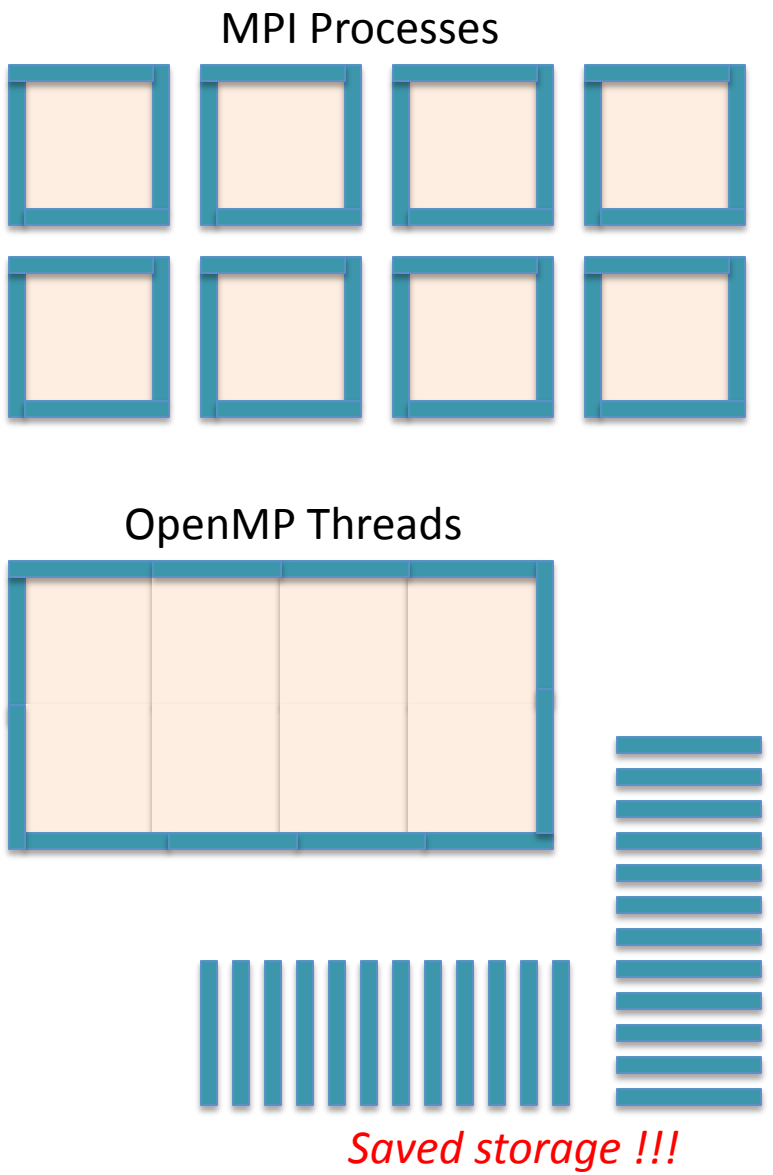
Process Memory Requirements MPI vs. OpenMP

- Separate processes need separate address and memory space
  - POSIX fork takes a duplication of everything except the process ID
- There are much more lightweight memory requirements for OpenMP threads
  - Only one copy of variables exists between threads when new threads are created
- Only one copy of MPI buffers etc. exists per process, and therefore only one copy exists *shared* between all threads launched from a process
- Using MPI/OpenMP hybrid programming reduces the memory requirement *overhead* from multiple processes

… in addition we may be able to benefit from reduced memory requirements within the application when using MPI/OpenMP hybrid programming …

Halo regions and replicated data in MPI

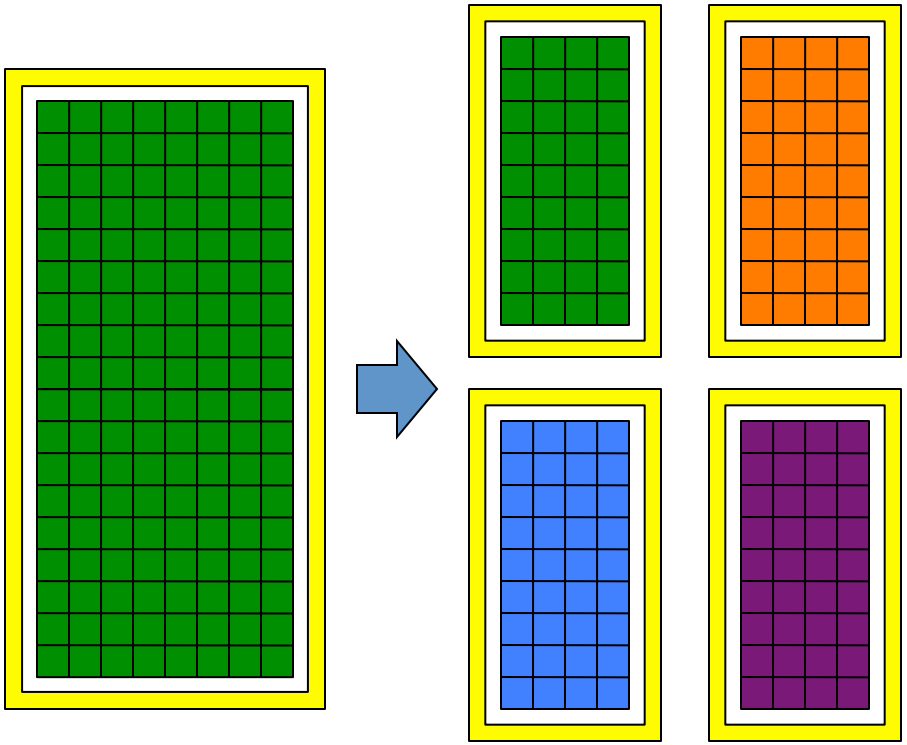
- Halo regions are local copies of remote data that are needed for computations
  - Halo regions need to be copied fequently
- Using OpenMP parallelism reduces the size of halos region copies that need to be stored
- Other data structures than these might also lead to a benefit of MPI/OpenMP applications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

Example of domain decomposition on a 2D grid



P processors, each with …
MxN Grid points
2M+2N Halo points

4P processors, each with …
(M/2)x(N/2) Grid points
M+N Halo points

Idealised 2D grid layout:

Increasing the number of processors by 4 leads to each processor having

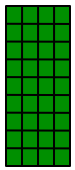
- *one quarter* the number of grid points to compute
- *one half* the number of halo points to communicate

Serial parts of the code do not change.

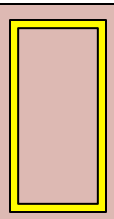
The same amount of total data needs to be output at each time step.

Using 4 OpenMP threads rather than 4 MPI processes keeps the halo region constant

Idealised scalability for a 2D Grid-based problem

Computation:  Scales O(P) for P processors

Minor scaling problem – issues of halo memory bandwidth, vector lengths, efficiency of software pipeline etc.

Communication:  Scales O(√P) for P processors

Major scaling problem – the halo region decreases slowly as you increase the number of processors

I/O and serial parts: No scaling

Limiting factor in scaling *– the same amount of work is carried out, or total data is output at each time step*

Reduced Communication of Halos and Updates

- Example: 8x8x8 cube with 1 element halo becomes a 10x10x10 cube (50% halo)
  - This grows to a 16x16x16 cube and halo of 18x18x18 on 8 cores (30% halo)
  - … and then potentially on to a cube of 32x32x32 and halo of 34x34x34 on 64 cores [4 x Interlagos] (%17 halo)
- An example of a wider halo is in the COSMO-2 numerical weather prediction simulations
  - These are run 8 times per day on 1000 processors
  - An individual process has typically a 20x10 (2D distribution) of grid points [extended by 60 atmospheric levels]
  - With a halo width of 3 elements this give a grid+halo of 26x16 (50% halo)
  - This would be a cube with 40x20 grid points and 46x26 grid+halo on 4 cores (33% halo)
  - … and then a grid of 80x80points and 86x86 grid+halo on 32 cores (13.5% halo)
- Replication of data is not restricted only to structured grids, but extends to most areas of computational science that do not rely on map-reduce methods

Stop-start Mechanisms in MPI

- The standard model of MPI communication is to have a stop-start mechanism
  - Compute, communicate, compute, communicate … etc.
- Communication using point to point *might* be done asynchronously, but collectives (under MPI-2) are definitely blocking
- If a large part of an application is concerned with communication then this will form a bottleneck at runtime

Asynchronous Implementations in MPI Libraries

- To an application programmer, the use of asynchronous point-to-point communications appears to offer a good opportunity for overlapping computation and communication
- The internals of most MPI implementations are not designed to really overlap communication with other activities
  - Common implementations such as MPICH etc. are not threaded
  - Most MPI implementations aren't thread-enabled and so communications only typically take place when an MPI call takes place
  - OpenMPI has introduced threading support in its internals
- Asynchronous transfers typically rely on interconnect hardware being able to do transfers using RDMA
- If communication does not take place immediately when the call is made it may have to wait until the MPI_Wait call (or some other MPI call)

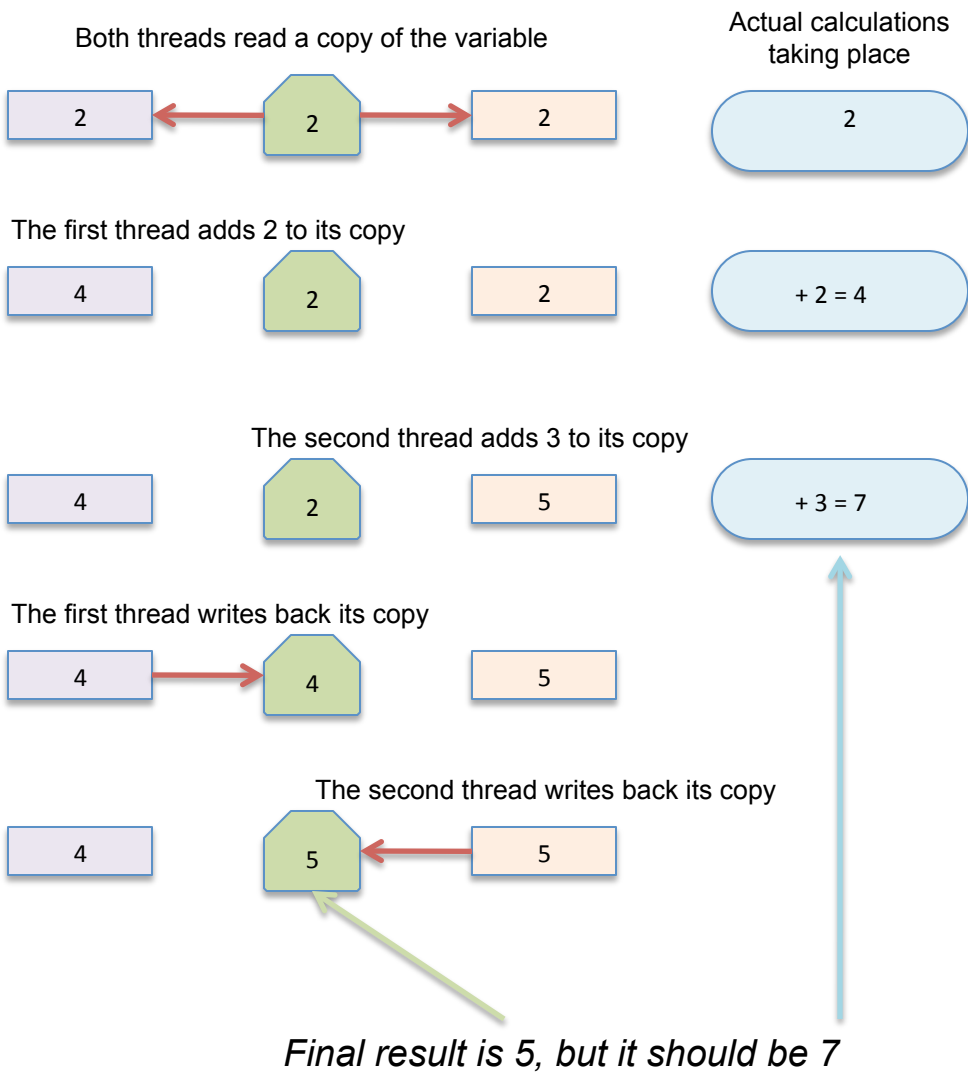
The Dangers of OpenMP !!!

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 … (more later)

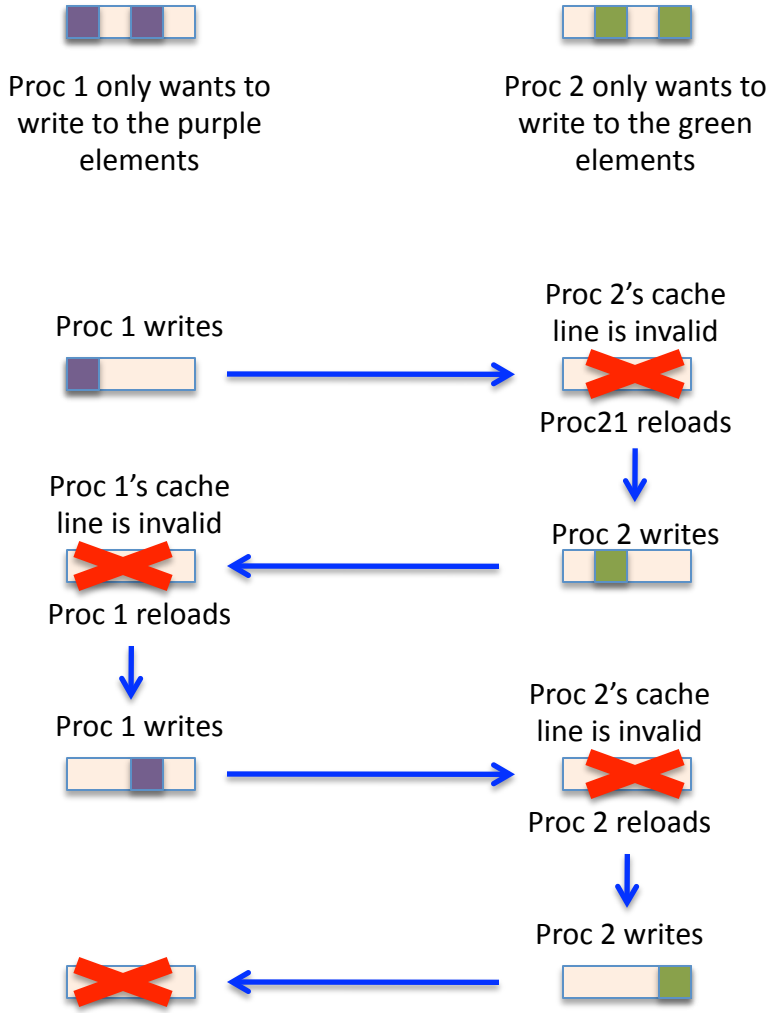
A common problem – race conditions

- 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



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



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

User-level thinking with distributed and shared memory

- One of the most difficult problems encountered with MPI programmers moving to OpenMP is that it *appears* to be easy
- MPI coding forces you to think about every data transfer – OpenMP lets you use data transfers in memory and so you don't need to think as carefully
- Consider some of the "invisible" performance problems that you might encounter and try to avoid them
  - Think about your whole application
  - Consider memory locality
  - Make sure that you initialise your data with the thread that will *mainly* use those memory locations
  - Be careful where you put your directives
    - Placing a directive immediately before a DO/for loop forces it to be the loop that is parallelised
      - This can inhibit compiler optimisations
- If you have a lot of MPI traffic make sure that you use the simplest threading model that will suffice
  - Normally you should be able to get away with MPI_THREAD_FUNNELED
- Try to make parallel regions as large as possible
- Use a NOWAIT clause wherever possible – but beware that this no longer implies a flush!
  - Nowait is at the beginning of a "#pragma omp parallel for" statement, but at the end of the "! $omp end do"