Basic Concepts in Parallelization

Ruud van der Pas

Senior Staff Engineer
Oracle Solaris Studio
Oracle
Menlo Park, CA, USA

IWOMP 2010
CCS, University of Tsukuba
Tsukuba, Japan
June 14-16, 2010

Outline

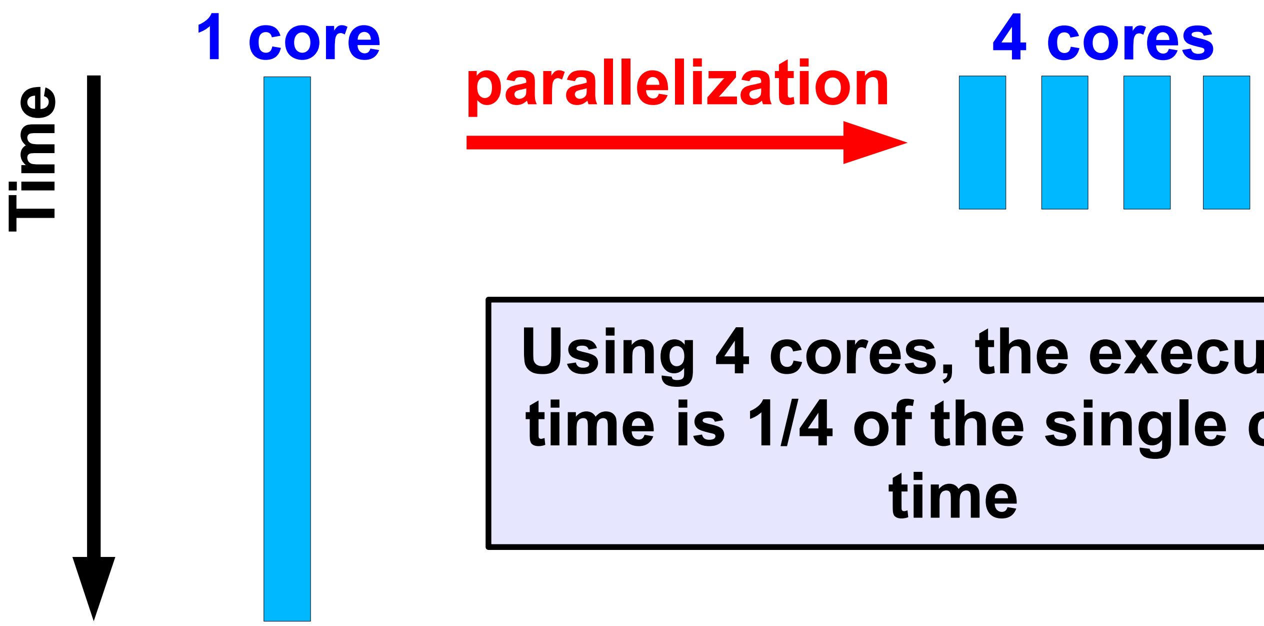
- *Introduction*
- *Parallel Architectures*
- *Parallel Programming Models*
- *Data Races*
- *Summary*

Introduction

Why Parallelization ?

Parallelization is another optimization technique
The goal is to reduce the execution time

To this end, multiple processors, or cores, are used



Using 4 cores, the execution time is 1/4 of the single core time

What Is Parallelization ?

"Something" is parallel if there is a <u>certain level of independence</u> in the order of operations

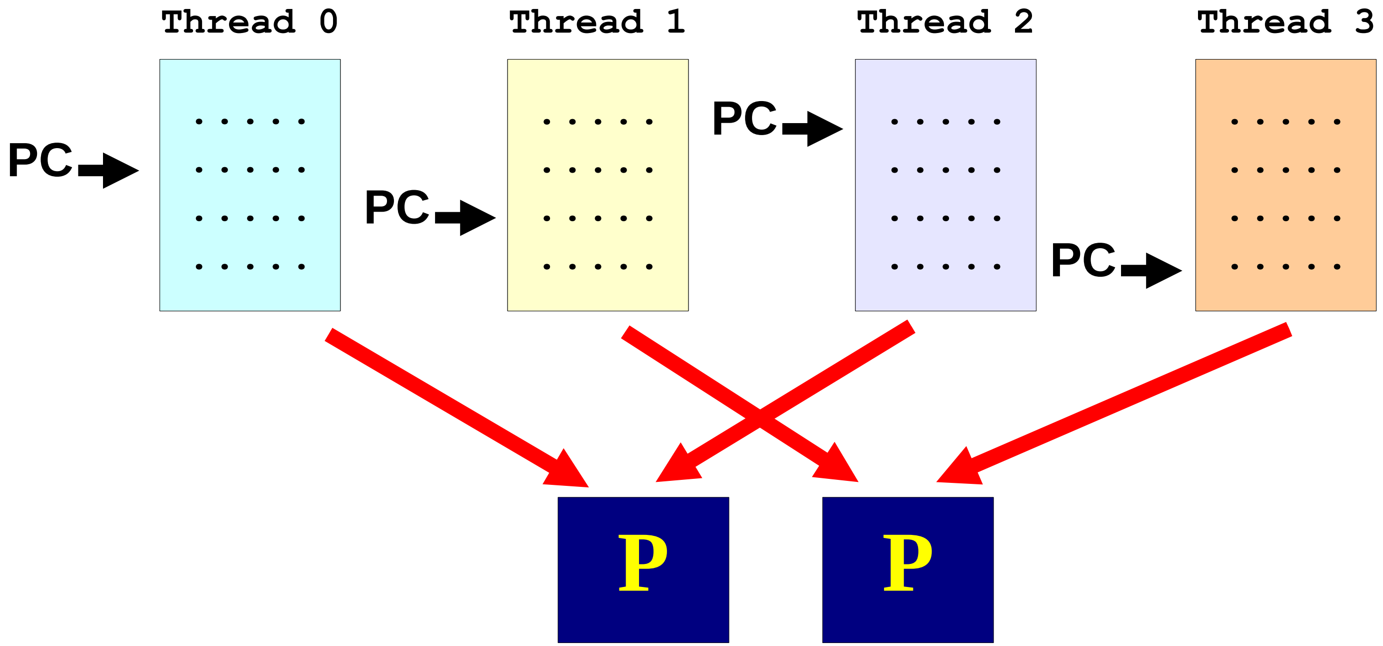
In other words, it doesn't matter in what order those operations are performed

- ***A sequence of machine instructions***
- ***A collection of program statements***
- ***An algorithm***
- ***The problem you're trying to solve***

granularity

What is a Thread ?

- *Loosely said, a thread consists of a series of instructions with it's own program counter ("PC") and state*
- *A parallel program executes threads in parallel*
- *These threads are then scheduled onto processors*

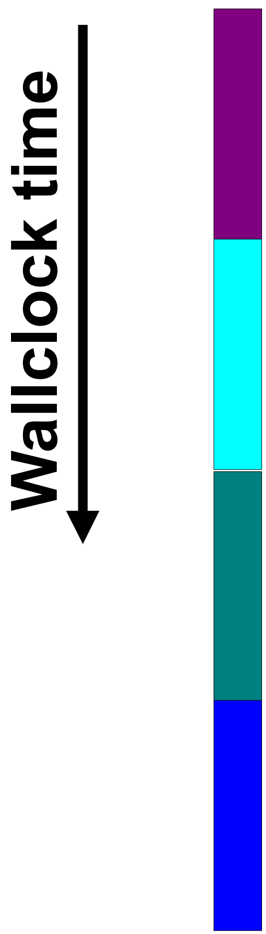


Parallel overhead

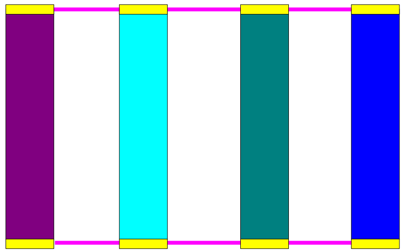
- ***The total CPU time often exceeds the serial CPU time:***
  - ***The newly introduced parallel portions in your program need to be executed***
  - ***Threads need time sending data to each other and synchronizing ("communication")***
    - ✔ ***Often the key contributor, spoiling all the fun***
- ***Typically, things also get worse when increasing the number of threads***
- ***Eff cient parallelization is about minimizing the communication overhead***

Communication

Serial Execution

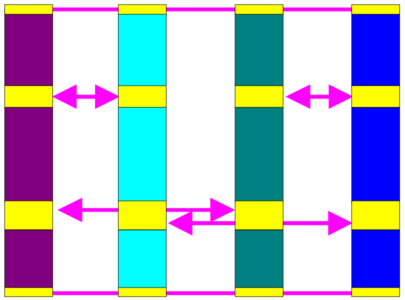


Parallel - Without communication



- Embarrassingly parallel: 4x faster
- Wallclock time is ¼ of serial wallclock time

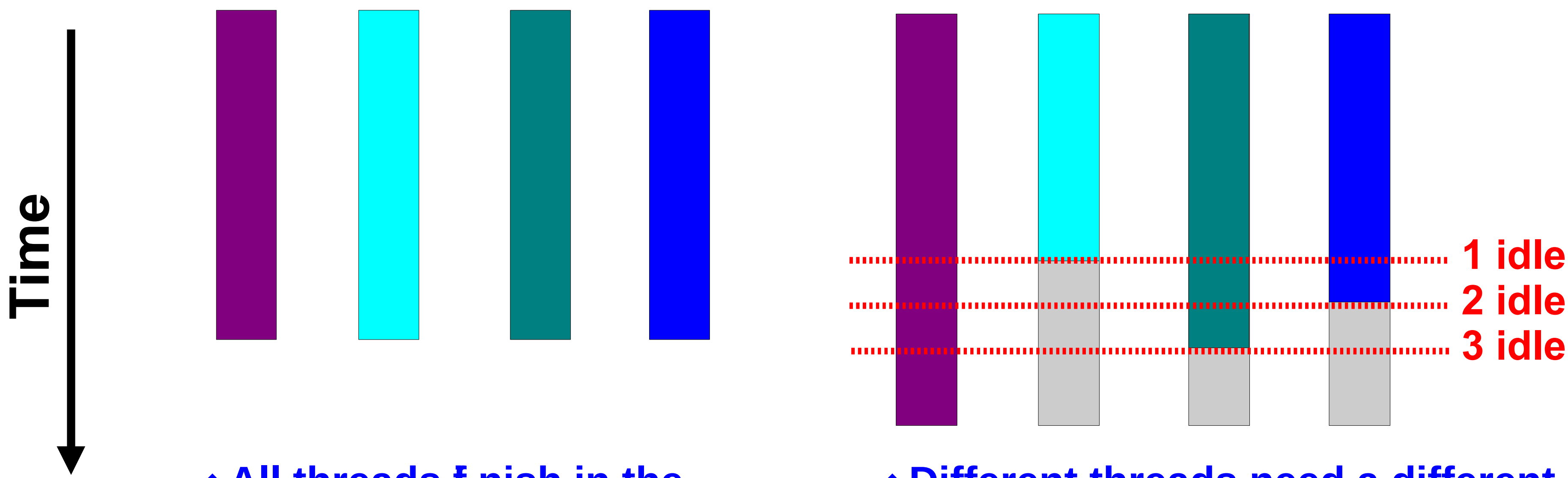
Parallel - With communication



- Additional communication
- Less than 4x faster
- Consumes additional resources
- Wallclock time is <u>more than</u> ¼ of serial wallclock time
- Total CPU time increases

Load balancing

Perfect Load Balancing / *Load Imbalance*



Perfect Load Balancing

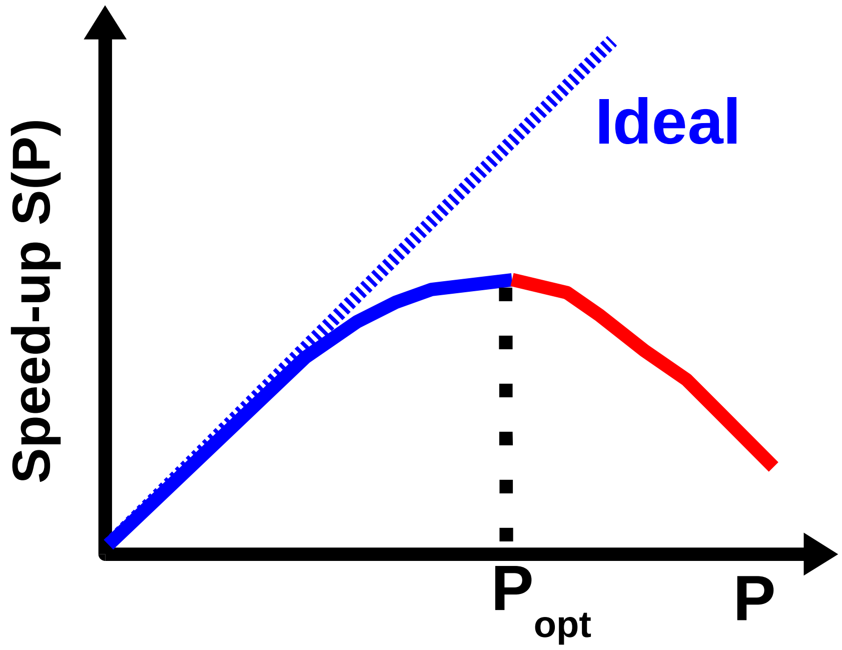
- All threads finish in the same amount of time
- No threads is idle

Load Imbalance

- Different threads need a different amount of time to finish their task
- Total wall clock time increases
- Program does not scale well

Thread is idle

About scalability



In some cases, S(P) exceeds P

This is called "superlinear" behaviour

Don't count on this to happen though

- *Define the speed-up S(P) as* $S(P) := T(1)/T(P)$
- *The efficiency E(P) is defined as* $E(P) := S(P)/P$
- *In the ideal case,* $S(P)=P$ *and* $E(P)=P/P=1=100\%$
- *Unless the application is embarrassingly parallel, S(P) eventually starts to deviate from the ideal curve*
- *Past this point* P_{opt}*, the application sees less and less benefit from adding processors*
- *Note that both metrics give no information on the actual run-time*
- *As such, they can be dangerous to use*

Amdahl's Law

Assume our program has a parallel fraction ***"f"***

This implies the execution time $T(1) := f*T(1) + (1-f)*T(1)$

On P processors: $T(P) = (f/P)*T(1) + (1-f)*T(1)$

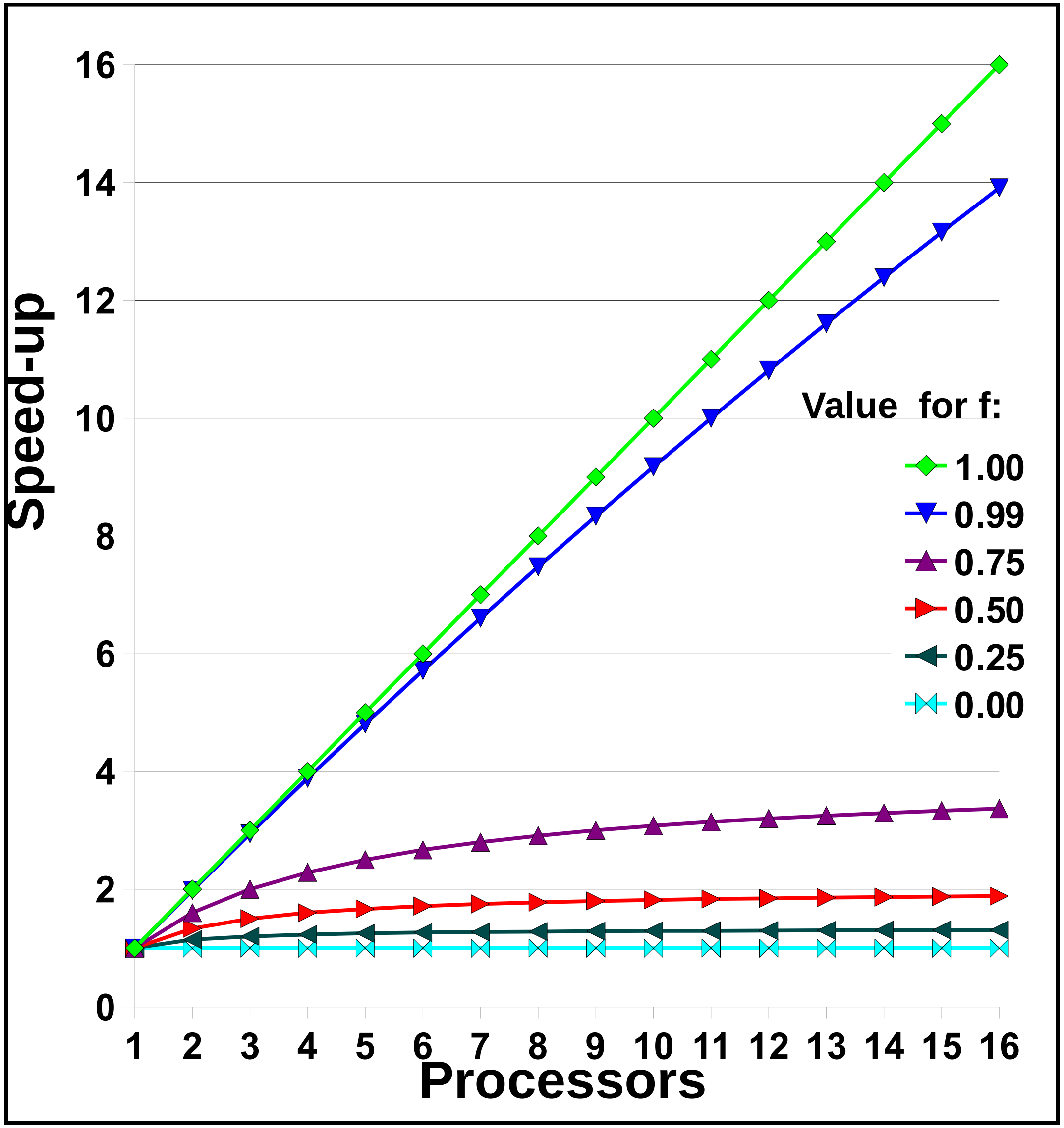
Amdahl's law:

$$S(P) = T(1)/T(P) = 1 / (f/P + 1-f)$$

Comments:

- *This "law' describes the effect the non-parallelizable part of a program has on scalability*
- *Note that the additional overhead caused by parallelization and speed-up because of cache effects are not taken into account*

Amdahl's Law



- ***It is easy to scale on a small number of processors***
- ***Scalable performance however requires a high degree of parallelization i.e. f is very close to 1***
- ***This implies that you need to parallelize that part of the code where the majority of the time is spent***

Amdahl's Law in practice

We can estimate the parallel fraction "f"

Recall: $T(P) = (f/P)*T(1) + (1-f)*T(1)$

It is trivial to solve this equation for "f":

$$f = (1 - T(P)/T(1))/(1 - (1/P))$$

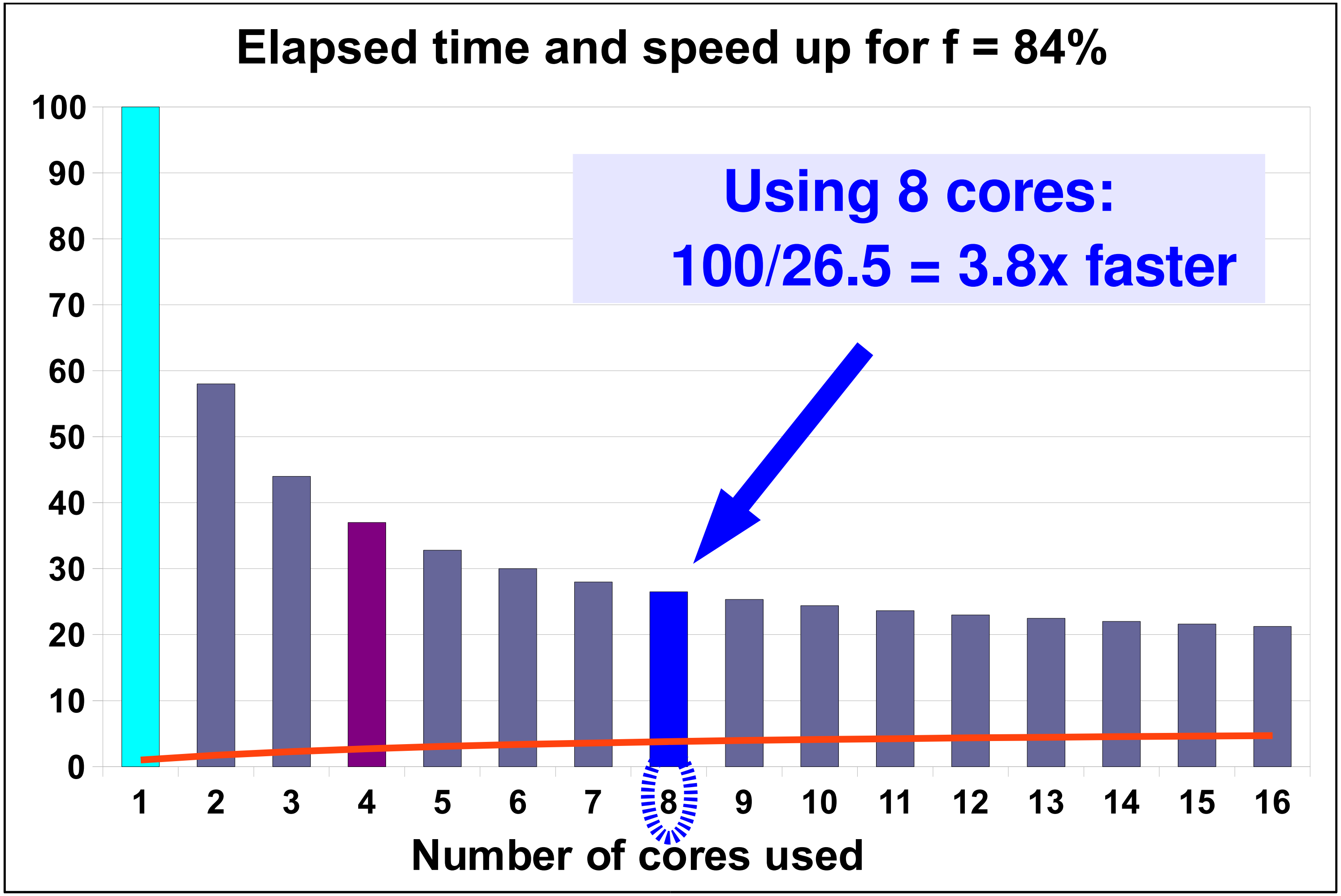
Example:

```
T(1) = 100 and T(4) = 37 => S(4) = T(1)/T(4) = 2.70
f = (1-37/100)/(1-(1/4)) = 0.63/0.75 = 0.84 = 84%
```

Estimated performance on 8 processors is then:

```
T(8) = (0.84/8)*100 + (1-0.84)*100 = 26.5
S(8) = T/T(8) = 3.78
```

Threads Are Getting Cheap .....



= Elapsed time

= Speed up

Numerical results

Consider:

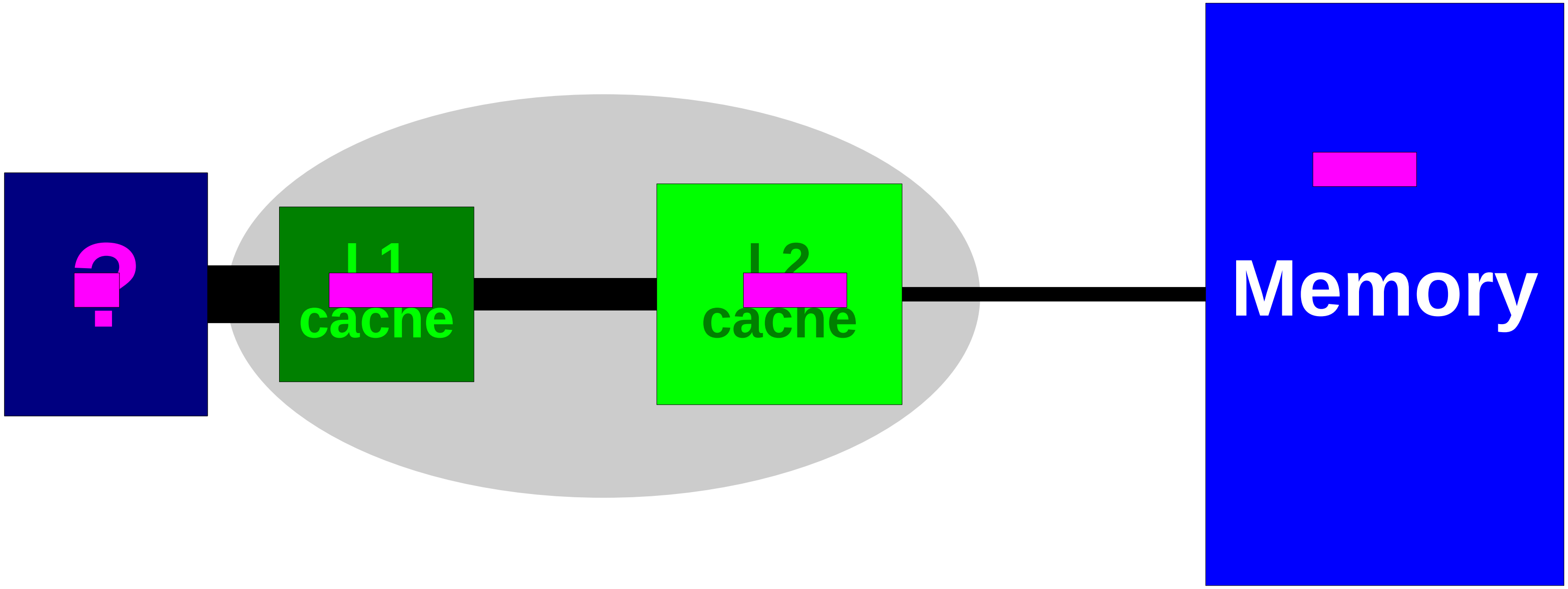
A = B + C + D + E

Serial Processing	*Parallel Processing*	
	Thread 0	**Thread 1**
`A = B + C`	`T1 = B + C`	`T2 = D + E`
`A = A + D`	`T1 = T1 + T2`	
`A = A + E`		

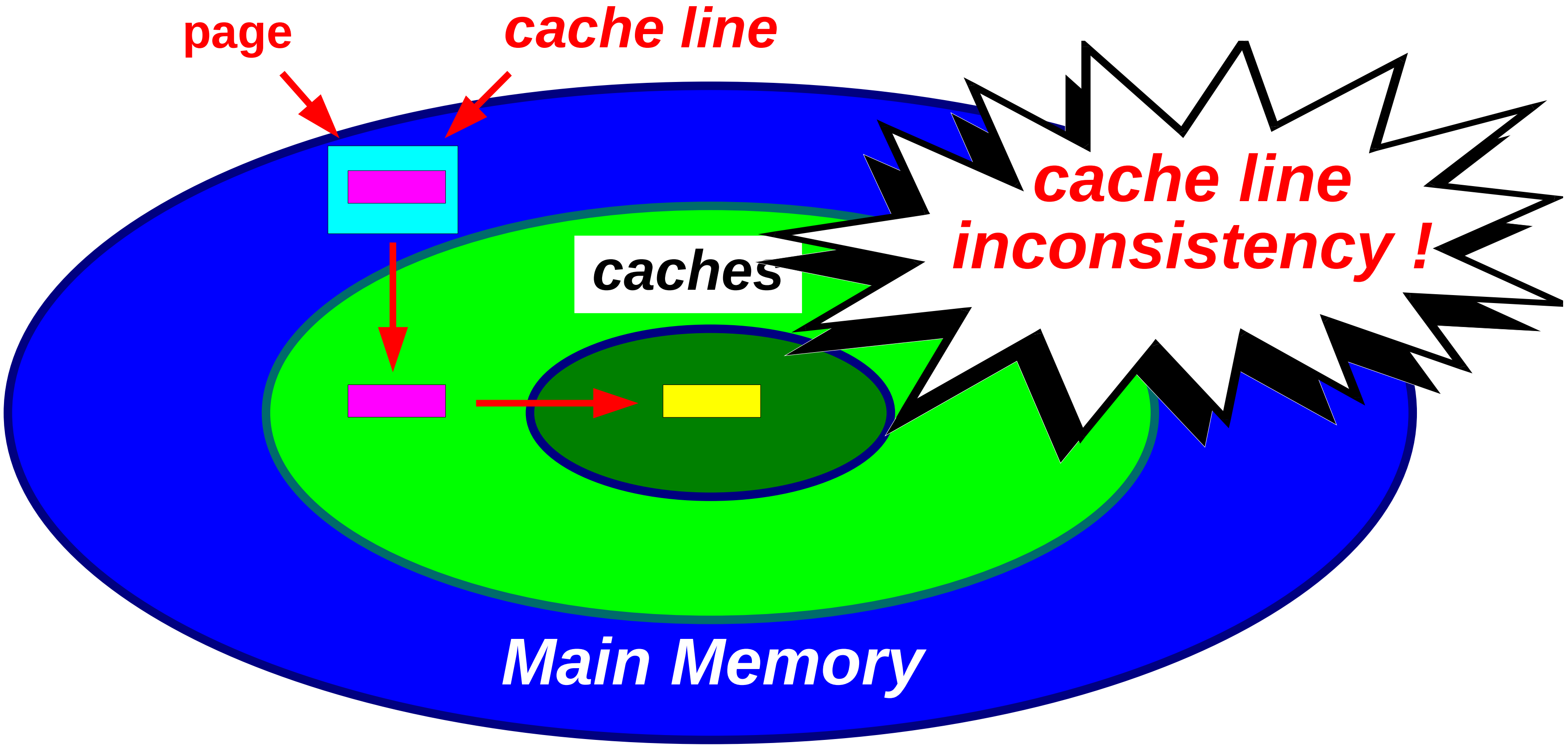
- ☞ ***The roundoff behaviour is different and so the numerical results may be different too***
- ☞ ***This is natural for parallel programs, but it may be hard to differentiate it from an ordinary bug ....***

Cache Coherence

Typical cache based system



Cache line modif cations

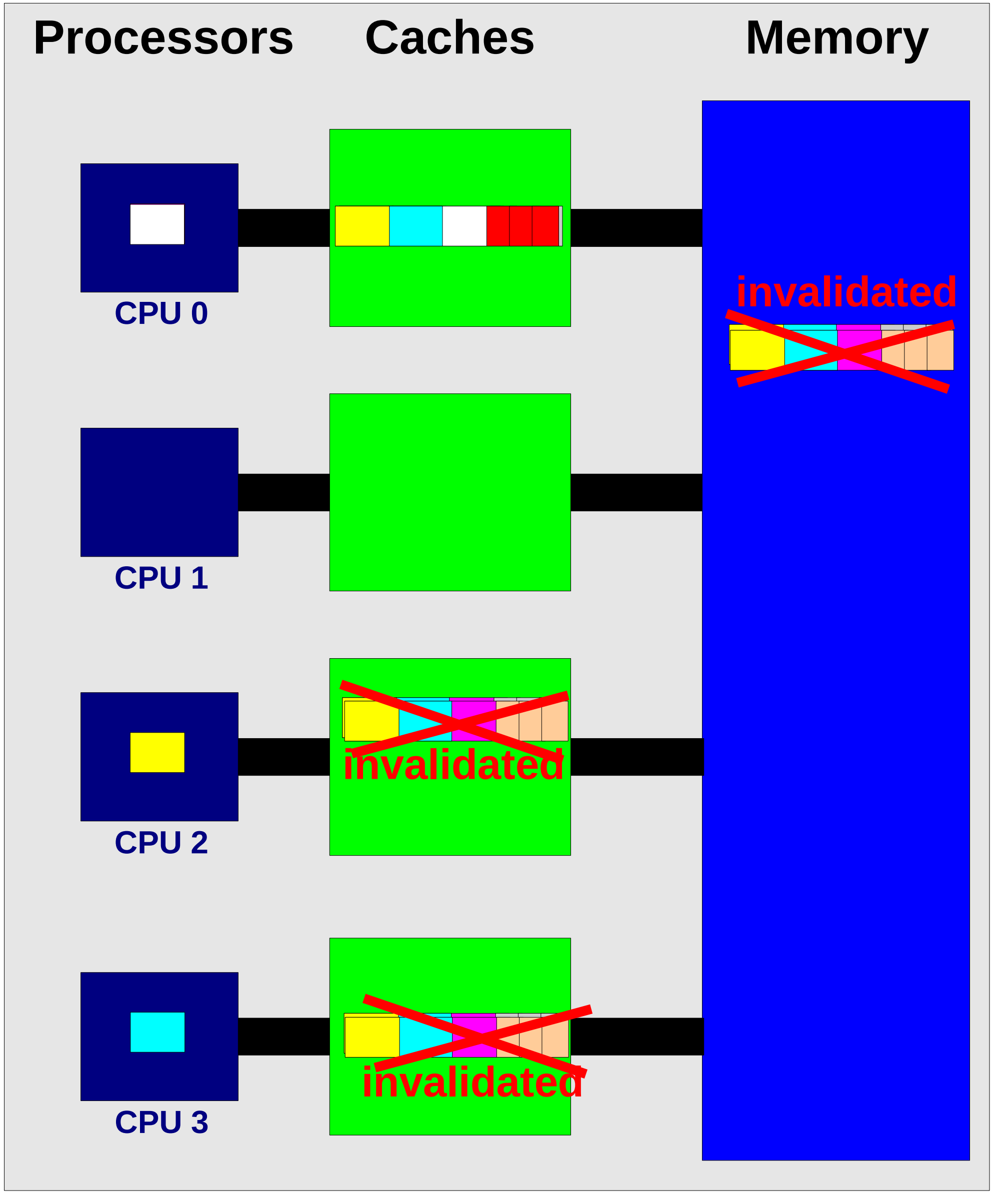


Caches in a parallel computer

- ❑ *A cache line starts in memory*
- ❑ *Over time multiple copies of this line may exist*

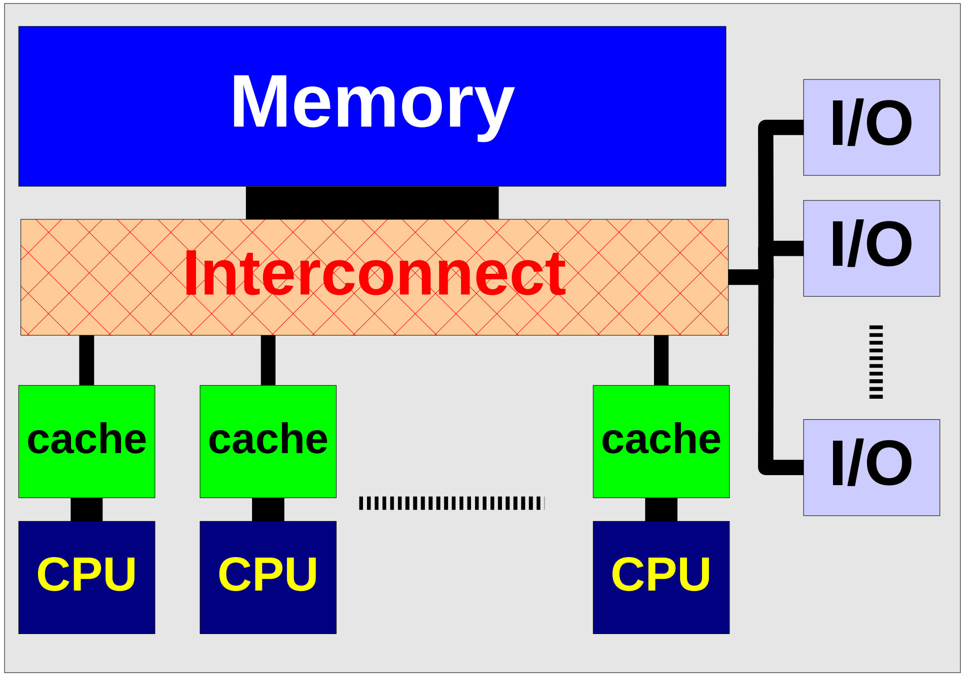
Cache Coherence ("cc"):

- ✔ *Tracks <u>changes</u> in copies*
- ✔ *Makes sure correct cache line is used*
- ✔ *Different implementations possible*
- ✔ *Need hardware support to make it eff cient*



Parallel Architectures

Uniform Memory Access (UMA)



Pro

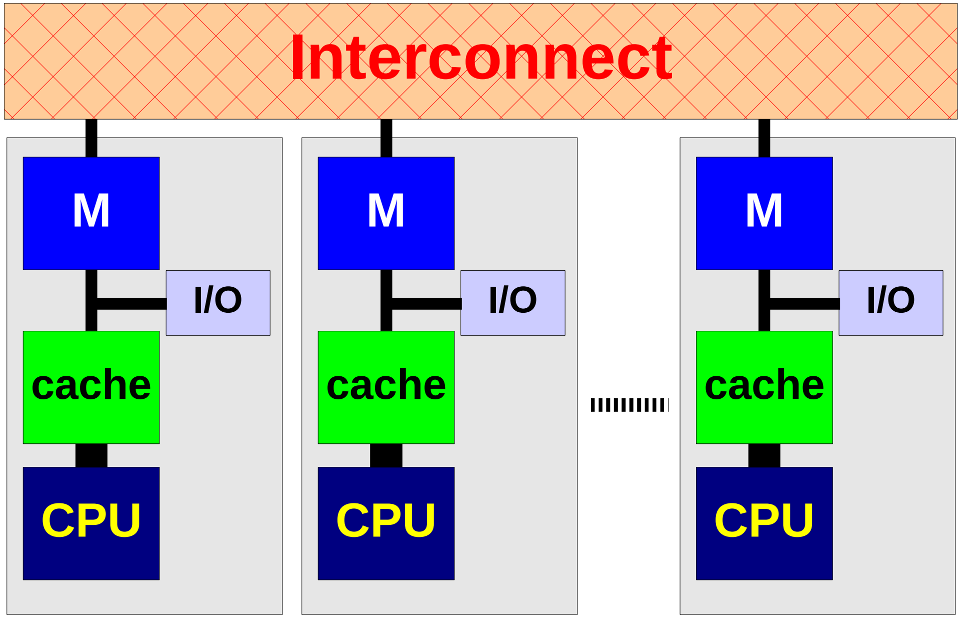
- ✔ *Easy to use and to administer*
- ✔ *Eff cient use of resources*

Con

- ✔ *Said to be expensive*
- ✔ *Said to be non-scalable*

- ❑ *Also called "SMP" (Symmetric Multi Processor)*
- ❑ *Memory Access time is Uniform for all CPUs*
- ❑ *CPU can be multicore*
- ❑ *Interconnect is "cc":*
  - *Bus*
  - *Crossbar*
- ❑ *No fragmentation - Memory and I/O are shared resources*

NUMA



Pro

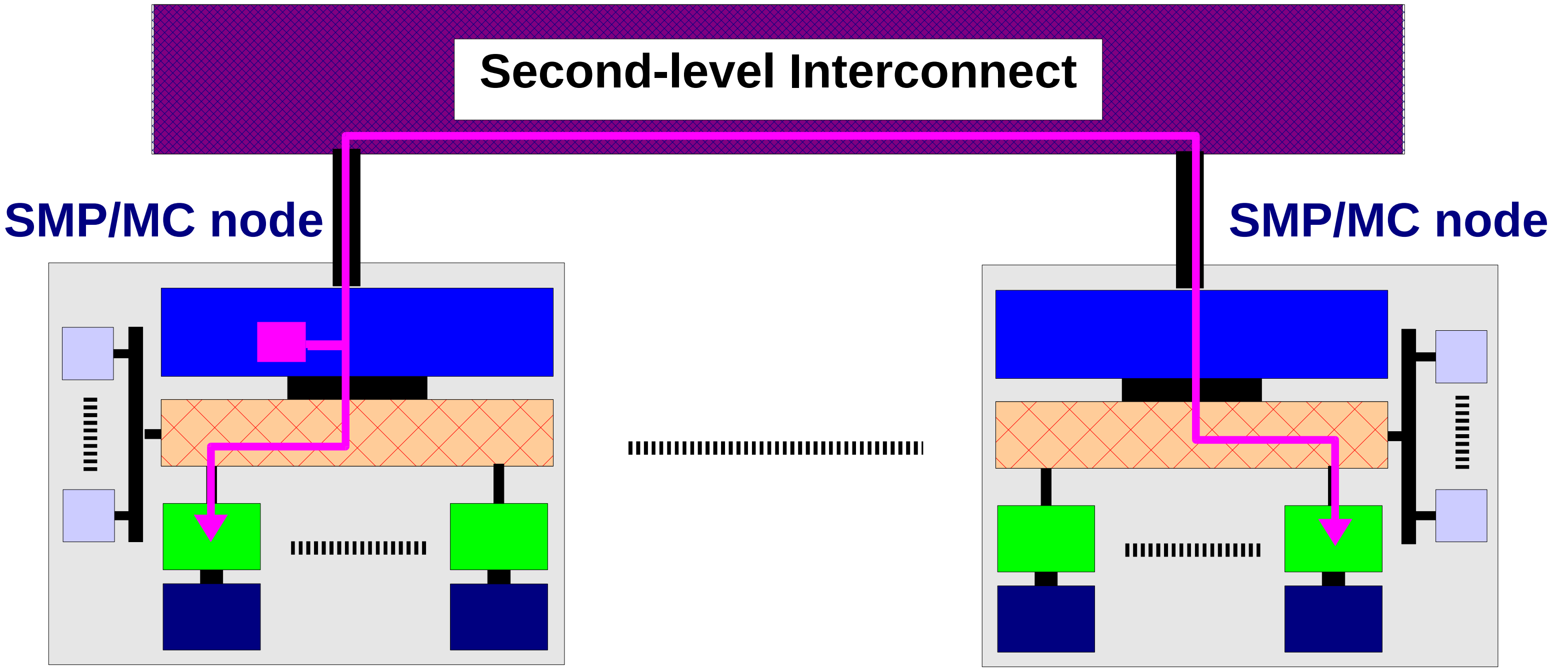
- ✔ *Said to be cheap*
- ✔ *Said to be scalable*

Con

- ✔ *Diff cult to use and administer*
- ✔ *In-eff cient use of resources*

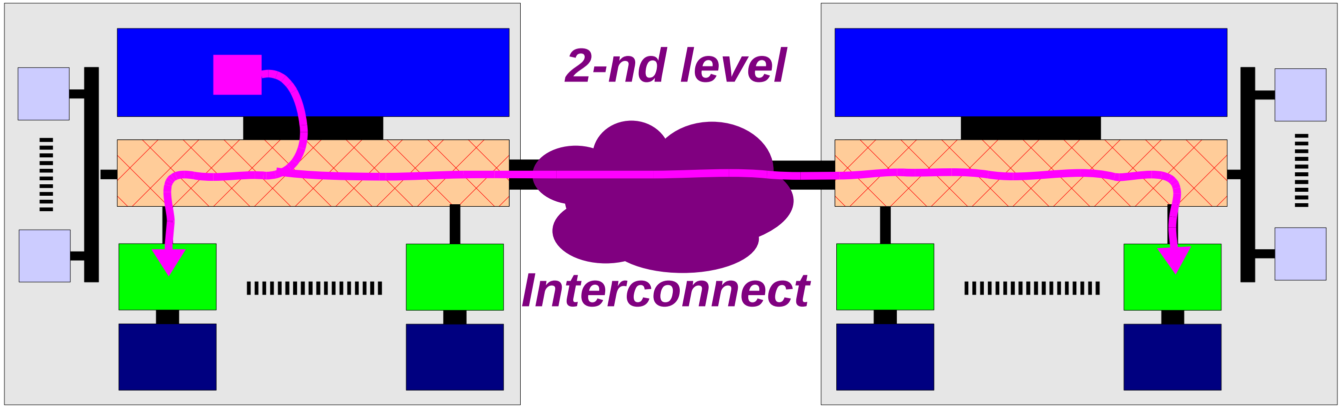
- ❑ ***Also called "Distributed Memory" or NORMA (No Remote Memory Access)***
- ❑ ***Memory Access time is Non-Uniform***
- ❑ ***Hence the name "NUMA"***
- ❑ ***Interconnect is not "cc":***
  - ***Ethernet, Inf niband, etc, ...***
- ❑ ***Runs 'N' copies of the OS***
- ❑ ***Memory and I/O are distributed resources***

The Hybrid Architecture



- **_Second-level interconnect is not cache coherent_**
  - **_Ethernet, Infiniband, etc, ..._**
- **_Hybrid Architecture with all Pros and Cons:_**
  - **_UMA within one SMP/Multicore node_**
  - **_NUMA across nodes_**

cc-NUMA



- Two-level interconnect:
  - UMA/SMP within one system
  - NUMA between the systems
- Both interconnects support cache coherence i.e. the system is fully cache coherent
- Has all the advantages ('look and feel') of an SMP
- Downside is the Non-Uniform Memory Access time

Parallel Programming Models

How To Program A Parallel Computer?

- ❑ *There are numerous parallel programming models*
- ❑ *The ones most well-known are:*
  - *Distributed Memory*
    - ✔ *Sockets (standardized, low level)*
    - ✔ *PVM - Parallel Virtual Machine (obsolete)*
    - ✔ *MPI - Message Passing Interface (de-facto std)*
  - *Shared Memory*
    - ✔ *Posix Threads (standardized, low level)*
    - ✔ *OpenMP (de-facto standard)*
    - ✔ *Automatic Parallelization (compiler does it for you)*

Any Cluster and/or single SMP

SMP only

Parallel Programming Models
Distributed Memory - MPI

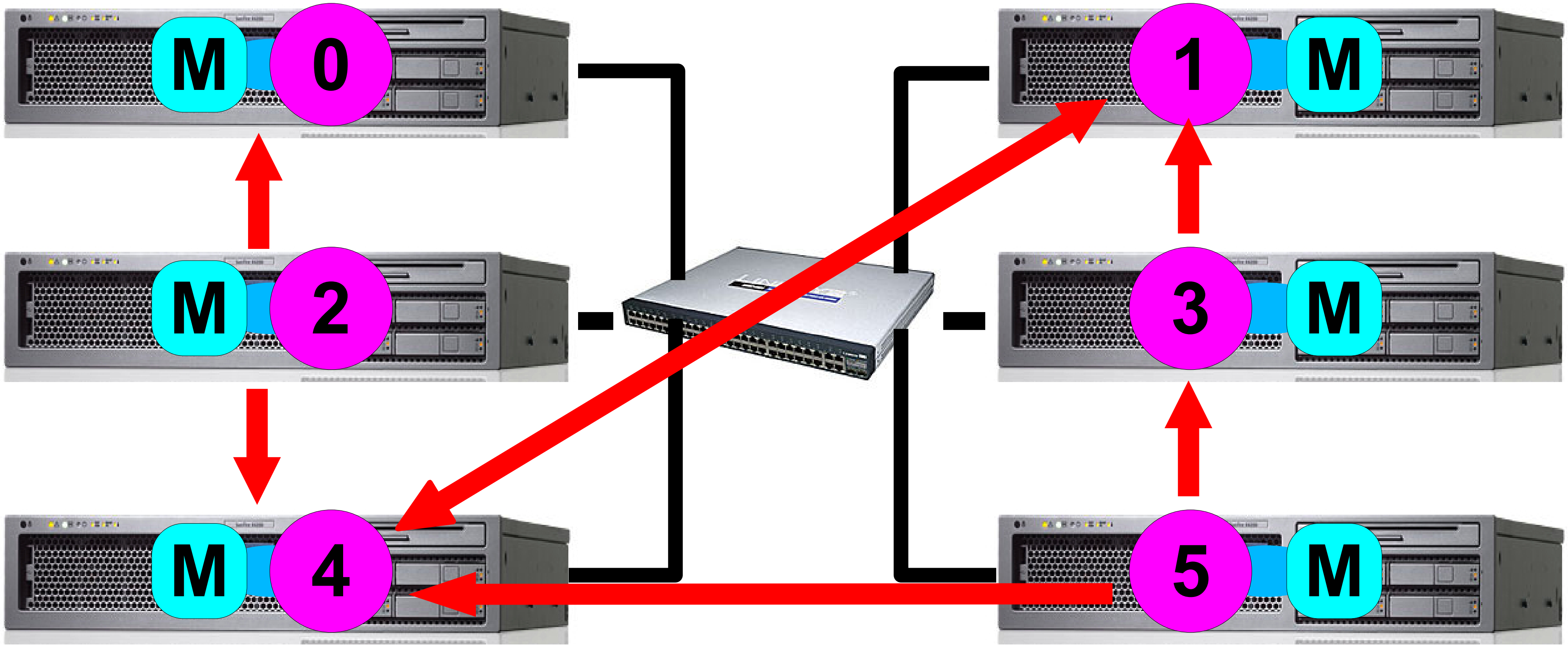
What is MPI?

- ❑ *MPI stands for the "Message Passing Interface"*
- ❑ *MPI is a very extensive de-facto parallel programming API for distributed memory systems (i.e. a cluster)*
  - *An MPI program can however also be executed on a shared memory system*
- ❑ *First specif cation: 1994*
- ❑ *The current MPI-2 specif cation was released in 1997*
  - *Major enhancements*
    - ✔ *Remote memory management*
    - ✔ *Parallel I/O*
    - ✔ *Dynamic process management*

More about MPI

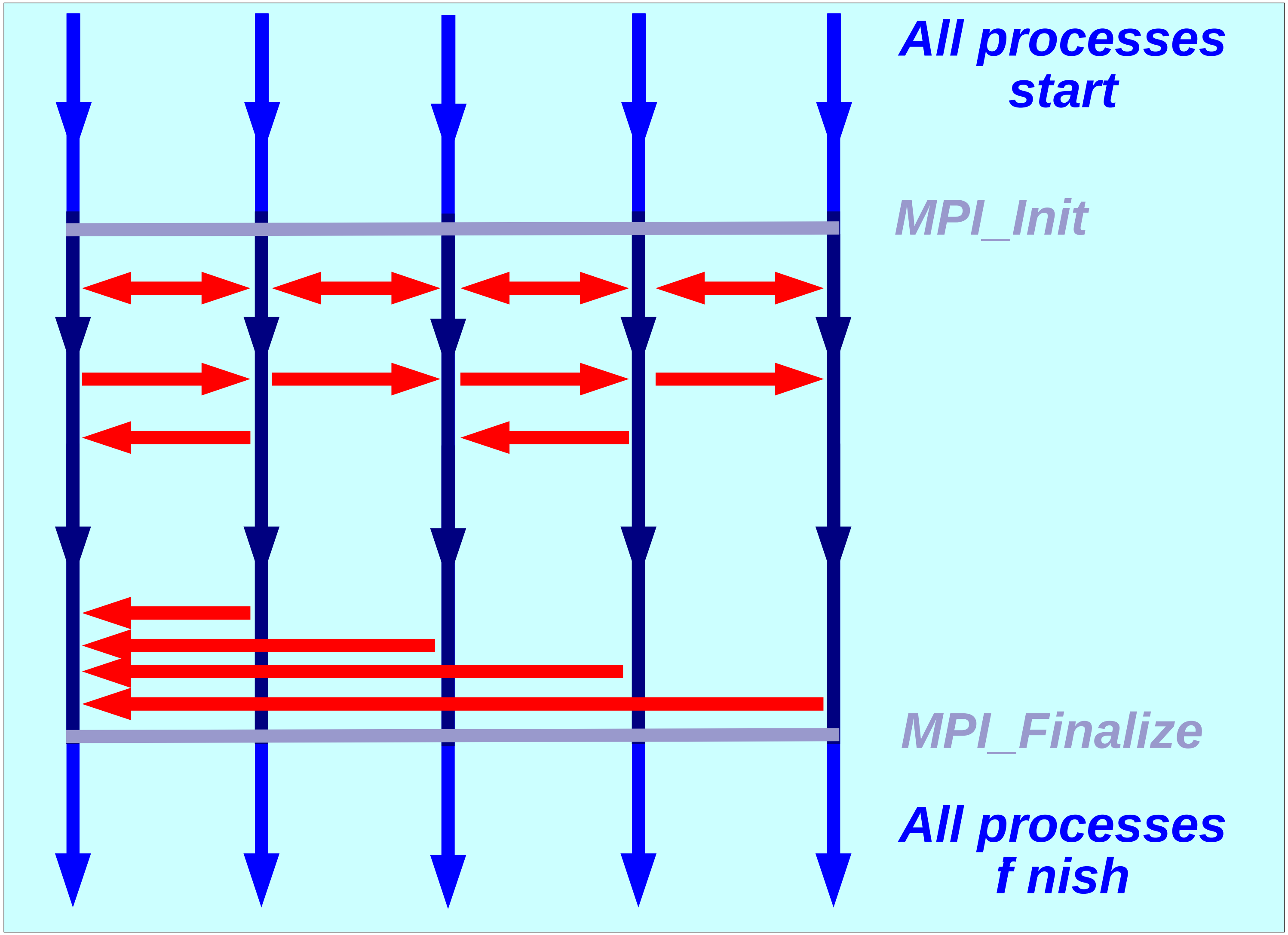
- MPI has its own data types (e.g. MPI_INT)
  - User defined data types are supported as well
- MPI supports C, C++ and Fortran
  - Include file <mpi.h> in C/C++ and "mpif.h" in Fortran
- An MPI environment typically consists of at least:
  - A library implementing the API
  - A compiler and linker that support the library
  - A run time environment to launch an MPI program
- Various implementations available
  - HPC Clustertools, MPICH, MVAPICH, LAM, Voltaire MPI, Scali MPI, HP-MPI, ...

The MPI Programming Model



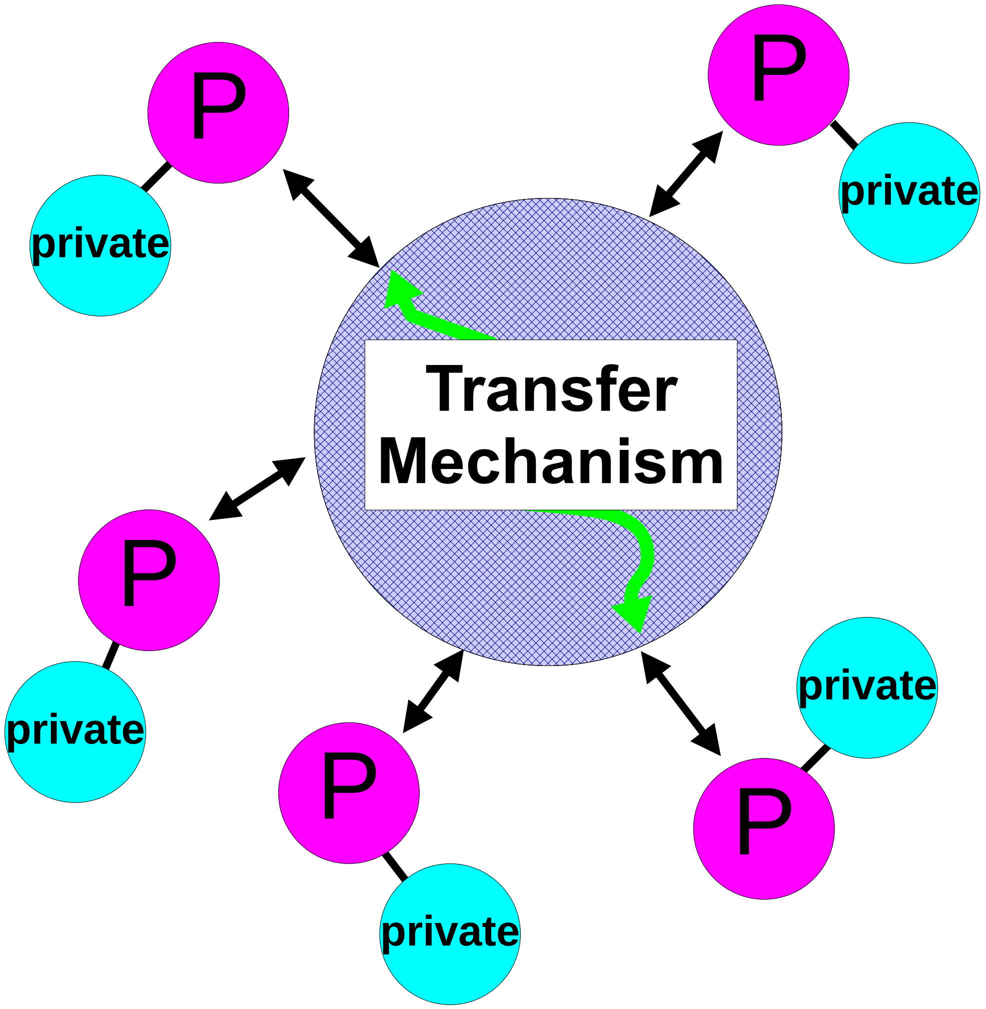
A Cluster Of Systems

The MPI Execution Model



= *communication*

The MPI Memory Model



- ✔ *All threads/processes have access to their own, private, memory only*
- ✔ *Data transfer and most synchronization has to be programmed explicitly*
- ✔ *All data is private*
- ✔ *Data is shared explicitly by exchanging buffers*

Example - Send “N” Integers

```
#include <mpi.h>          include f le
you = 0; him = 1;
MPI_Init(&argc, &argv);     initialize MPI environment
MPI_Comm_rank(MPI_COMM_WORLD, &me);     get process ID
if ( me == 0 ) {          process 0 sends
   error_code = MPI_Send(&data_buffer, N, MPI_INT,
                         him, 1957, MPI_COMM_WORLD);
} else if ( me == 1 ) {     process 1 receives
   error_code = MPI_Recv(&data_buffer, N, MPI_INT,
                         you, 1957, MPI_COMM_WORLD,
                         MPI_STATUS_IGNORE);
}
MPI_Finalize();     leave the MPI environment
```

Run time Behavior

Process 0

```
you = 1
him = 0

me = 0

MPI_Send
```

N integers
destination = you = 1
label = 1957

N integers
sender = him =0
label = 1957

Process 1

```
you = 1
him = 0

me = 1

MPI_Recv
```

Yes ! Connection established

The Pros and Cons of MPI

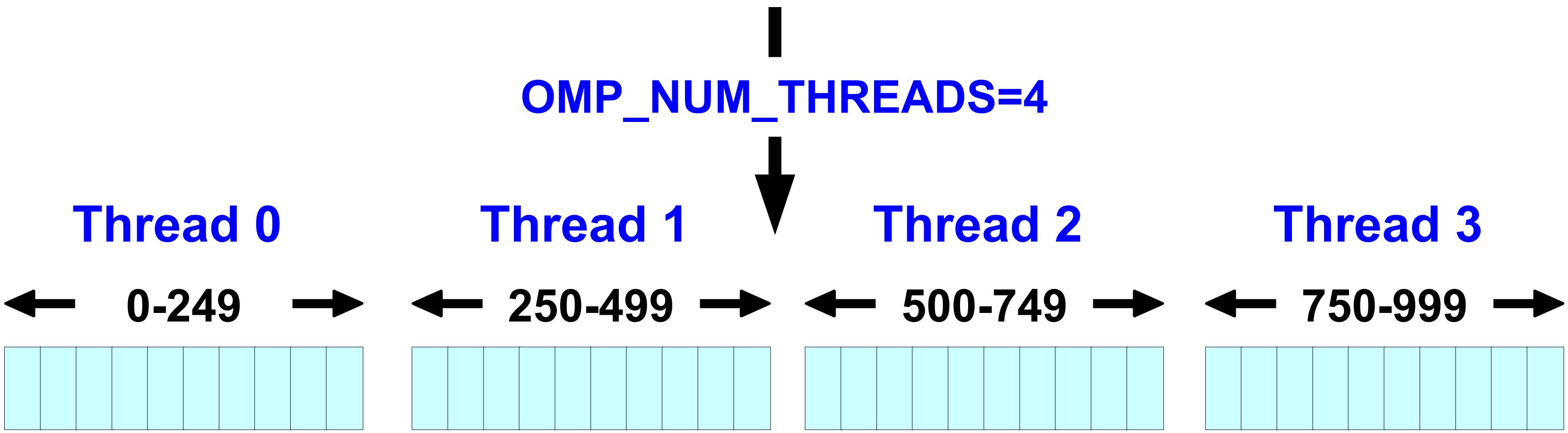
- ***Advantages of MPI:***
  - ***Flexibility - Can use any cluster of any size***
  - ***Straightforward - Just plug in the MPI calls***
  - ***Widely available - Several implementations out there***
  - ***Widely used - Very popular programming model***
- ***Disadvantages of MPI:***
  - ***Redesign of application - Could be a lot of work***
  - ***Easy to make mistakes - Many details to handle***
  - ***Hard to debug - Need to dig into underlying system***
  - ***More resources - Typically, more memory is needed***
  - ***Special care - Input/Output***

Parallel Programming Models Shared Memory - Automatic Parallelization

Automatic Parallelization (-xautopar)

- ***Compiler performs the parallelization (loop based)***
- ***Different iterations of the loop executed in parallel***
- ***<u>Same</u> binary used for <u>any</u> number of threads***

```
for (i=0; i<1000; i++)
  a[i] = b[i] + c[i];
```

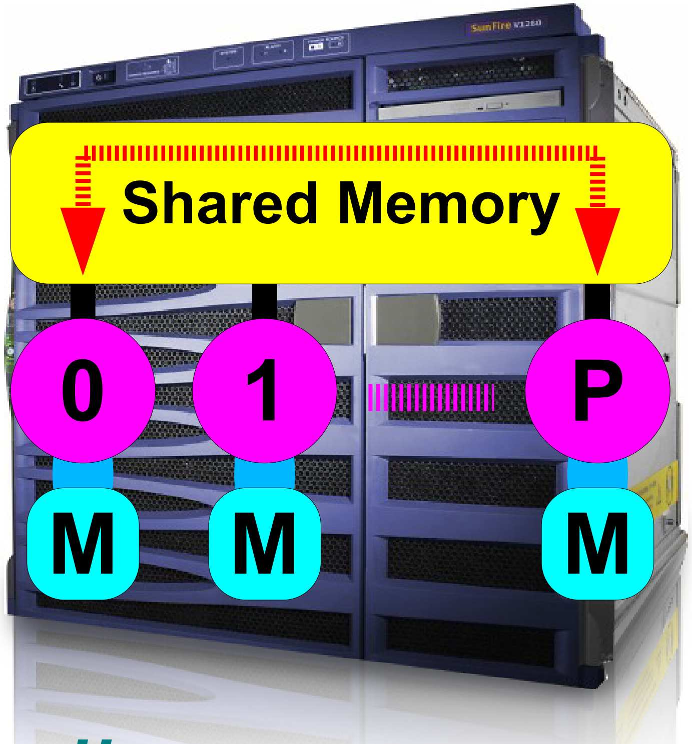


International Workshop IWOMP on OpenMP

Parallel Programming Models
Shared Memory - OpenMP

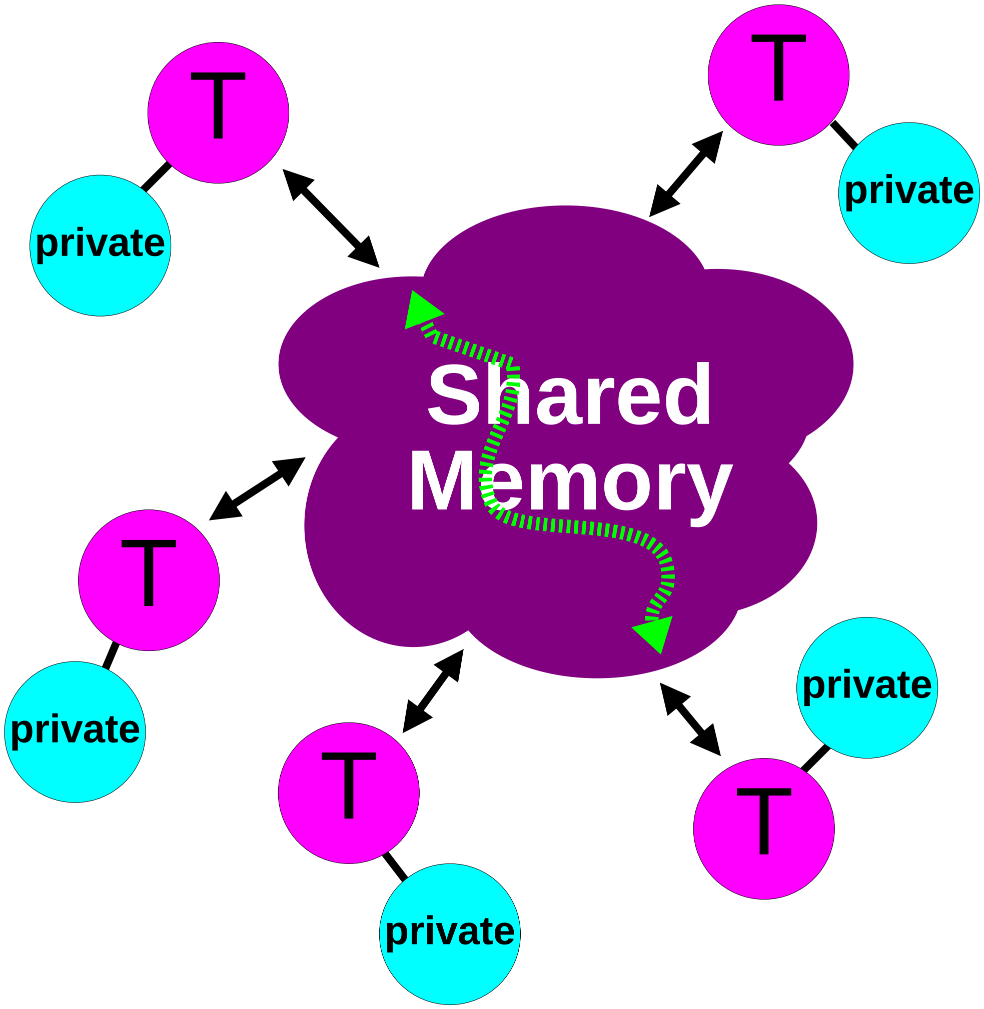
International Workshop IWOMP on OpenMP

OpenMP™



http://www.openmp.org

Shared Memory Model



Programming Model

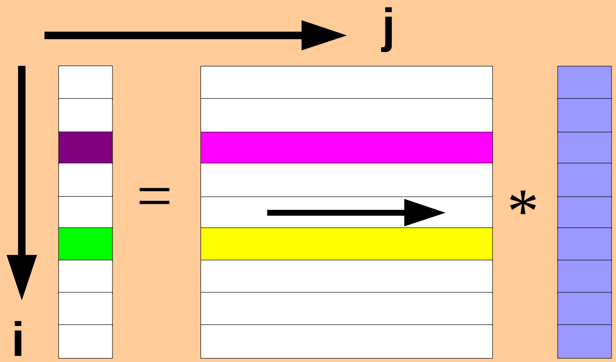
- *All threads have access to the same, globally shared, memory*
- *Data can be shared or private*
- *Shared data is accessible by all threads*
- *Private data can only be accessed by the thread that owns it*
- *Data transfer is transparent to the programmer*
- *Synchronization takes place, but it is mostly implicit*

About data

- *In a shared memory parallel program variables have a "label" attached to them:*
  - ☞ *Labelled "Private" ⇨ Visible to one thread only*
    - ✔ *Change made in local data, is not seen by others*
    - ✔ *Example - Local variables in a function that is executed in parallel*
  - ☞ *Labelled "Shared" ⇨ Visible to all threads*
    - ✔ *Change made in global data, is seen by all others*
    - ✔ *Example - Global data*

Example - Matrix times vector

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
   sum =   b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum =   b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
   sum =   b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum =   b[i=6][j]*c[j]
   a[6] = sum
```

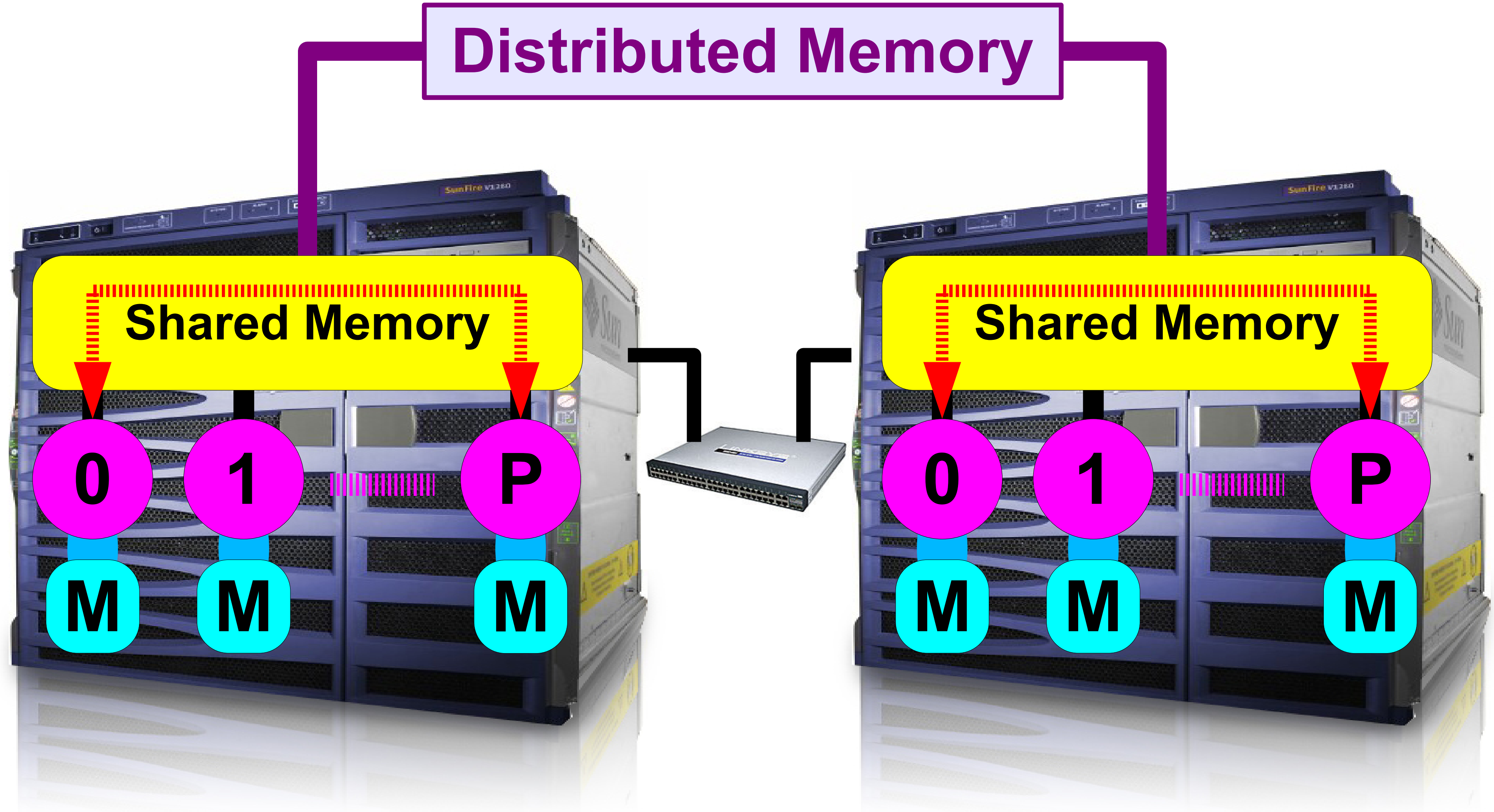
... etc ...

A Black and White comparison

MPI	OpenMP
De-facto standard	De-facto standard
Endorsed by all key players	Endorsed by all key players
Runs on any number of (cheap) systems	Limited to one (SMP) system
“Grid Ready”	Not (yet?) “Grid Ready”
High and steep learning curve	Easier to get started (but, ...)
You're on your own	Assistance from compiler
All or nothing model	Mix and match model
No data scoping (shared, private, ..)	Requires data scoping
More widely used (but ....)	Increasingly popular (CMT !)
Sequential version is not preserved	Preserves sequential code
Requires a library only	Need a compiler
Requires a run-time environment	No special environment
Easier to understand performance	Performance issues implicit

The Hybrid Parallel Programming Model

The Hybrid Programming Model



Data Races

About Parallelism

Parallelism

↓

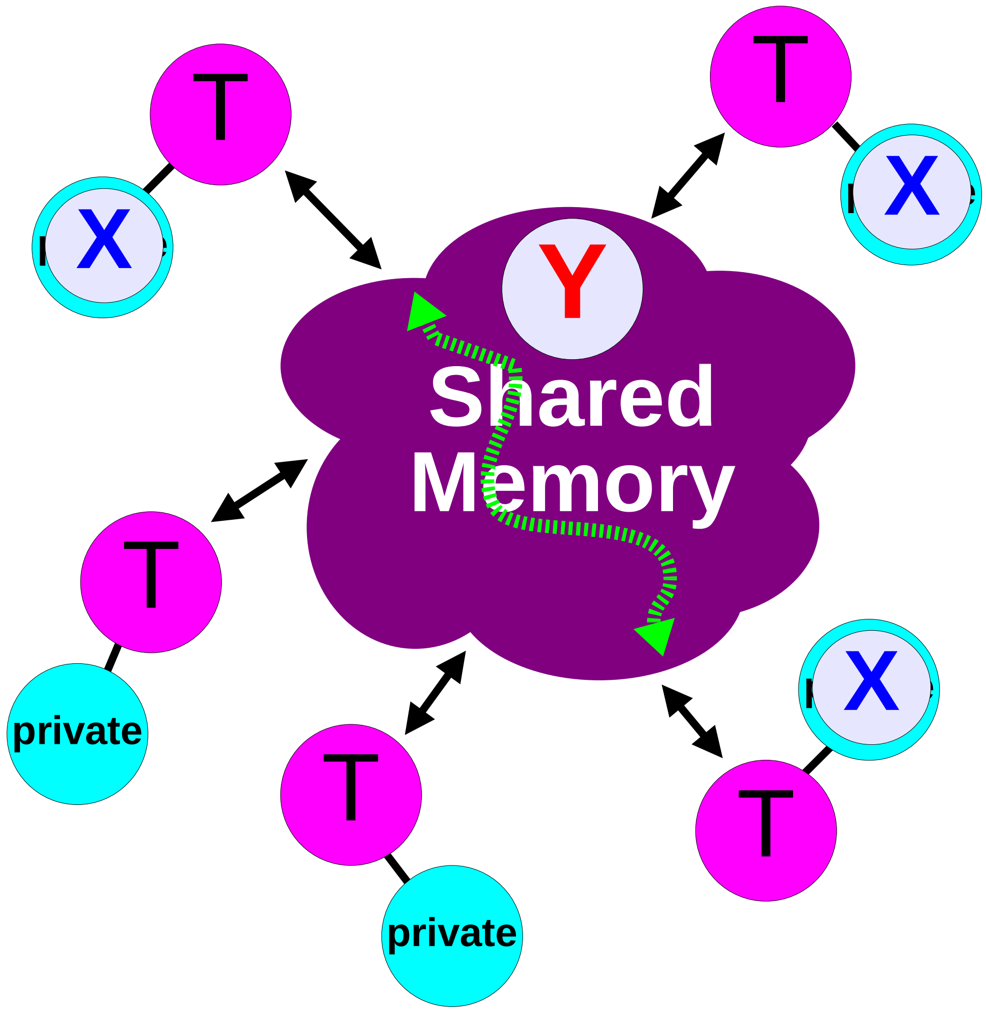
Independence

↓

No Fixed Ordering

"Something" that does not obey this rule, is not parallel (at that level ...)

Shared Memory Programming



Threads communicate via shared memory

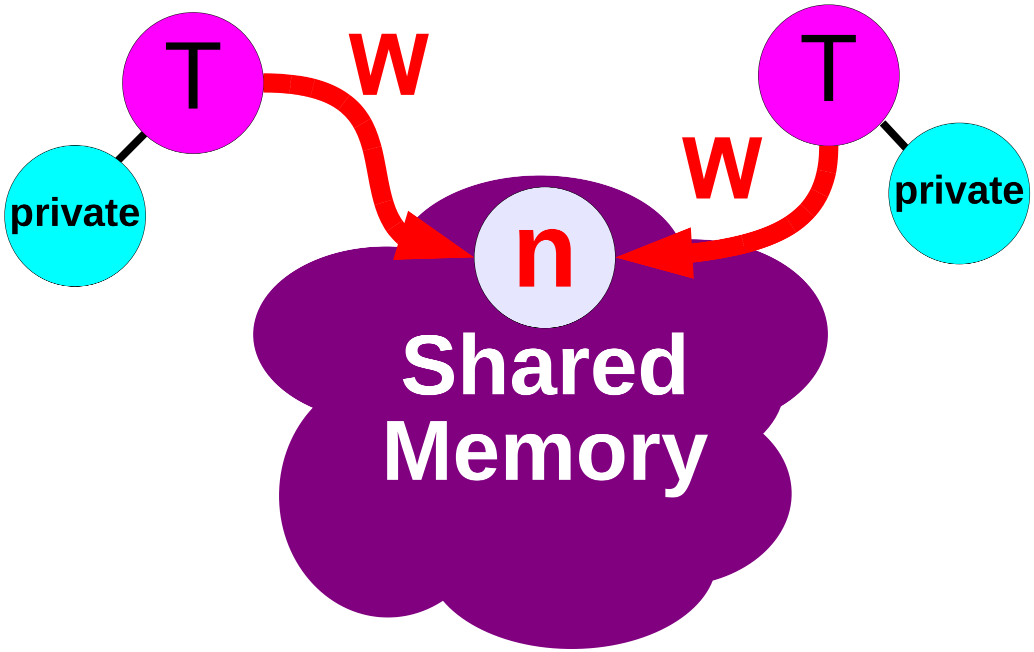
What is a Data Race?

- ***Two different threads in a multi-threaded shared memory program***
- ***Access the <u>same</u> (=shared) memory location***
  - ***Asynchronously <u>and</u>***
  - ***Without holding any common exclusive locks <u>and</u>***
  - ***At least one of the accesses is a write/store***

Example of a data race

```
#pragma omp parallel shared(n)
```

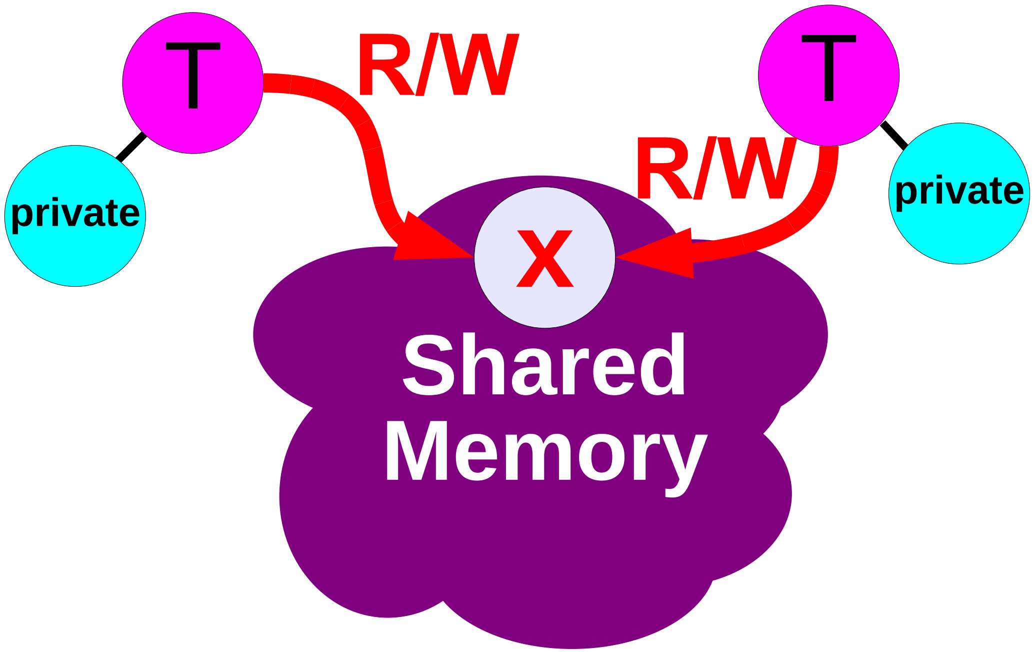
```
{n = omp_get_thread_num();}
```



Another example

```
#pragma omp parallel shared(x)
```

```
{x = x + 1;}
```



About Data Races

- ***Loosely described, a data race means that the <u>update</u> of a shared variable is not well protected***
- ***A data race tends to show up in a nasty way:***
  - ***Numerical results are (somewhat) different from run to run***
  - ***Especially with Floating-Point data diff cult to distinguish from a numerical side-effect***
  - ***Changing the number of threads can cause the problem to seemingly (dis)appear***
    - ✔ ***May also depend on the load on the system***
  - ***May only show up using many threads***

A parallel loop

```
for (i=0; i<8; i++)
    a[i] = a[i] + b[i];
```

Every iteration in this loop is independent of the other iterations

Thread 1	Thread 2
`a[0]=a[0]+b[0]`	`a[4]=a[4]+b[4]`
`a[1]=a[1]+b[1]`	`a[5]=a[5]+b[5]`
`a[2]=a[2]+b[2]`	`a[6]=a[6]+b[6]`
`a[3]=a[3]+b[3]`	`a[7]=a[7]+b[7]`

Time

Not a parallel loop

```
for (i=0; i<8; i++)
    a[i] = a[i+1] + b[i];
```

The result is not deterministic when run in parallel !

Thread 1	Thread 2
a[0]=a[1]+b[0]	a[4]=a[5]+b[4]
a[1]=a[2]+b[1]	a[5]=a[6]+b[5]
a[2]=a[3]+b[2]	a[6]=a[7]+b[6]
a[3]=a[4]+b[3]	a[7]=a[8]+b[7]

Time

About the experiment

- ***We manually parallelized the previous loop***
  - ***The compiler detects the data dependence and does not parallelize the loop***
- ***Vectors* a *and* b *are of type integer***
- ***We use the checksum of* a *as a measure for correctness:***
  - ***checksum += a[i]   for i = 0, 1, 2, ..., n-2***
- ***The correct, sequential, checksum result is computed as a reference***
- ***We ran the program using 1, 2, 4, 32 and 48 threads***
  - ***Each of these experiments was repeated 4 times***

Numerical results

Data Race In Action !

```
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953
threads:   1 checksum   1953 correct value   1953

threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953
threads:   2 checksum   1953 correct value   1953

threads:   4 checksum   1905 correct value   1953
threads:   4 checksum   1905 correct value   1953
threads:   4 checksum   1953 correct value   1953
threads:   4 checksum   1937 correct value   1953

threads:  32 checksum   1525 correct value   1953
threads:  32 checksum   1473 correct value   1953
threads:  32 checksum   1489 correct value   1953
threads:  32 checksum   1513 correct value   1953

threads:  48 checksum    936 correct value   1953
threads:  48 checksum   1007 correct value   1953
threads:  48 checksum    887 correct value   1953
threads:  48 checksum    822 correct value   1953
```

Summary

Parallelism Is Everywhere

Multiple levels of parallelism:

Granularity	*Technology*	*Programming Model*
Instruction Level	***Superscalar***	***Compiler***
Chip Level	***Multicore***	***Compiler, OpenMP, MPI***
System Level	***SMP/cc-NUMA***	***Compiler, OpenMP, MPI***
Grid Level	***Cluster***	***MPI***

Threads Are Getting Cheap