Parallel Programming Intro

Ruud van der Pas

Senior Staff Engineer
Technical Developer Tools
Sun Microsystems, Menlo Park, CA, USA

"Parallel Programming in Computational Engineering and Science"
RWTH Aachen University, Aachen, Germany
March 23-27, 2009

Outline

- *Intro Performance Tuning*
- *Multicore Processor Architectures*
- *Parallel Architectures*
- *Parallel Programming Basics*
- *Parallel Programming Models*

1

Intro Performance Tuning

Why Bother Tuning An Application ?

Compare two computers: System A and System B

The CPU	of System B is	"n"	times faster
The Memory	of System B is	"k"	times faster

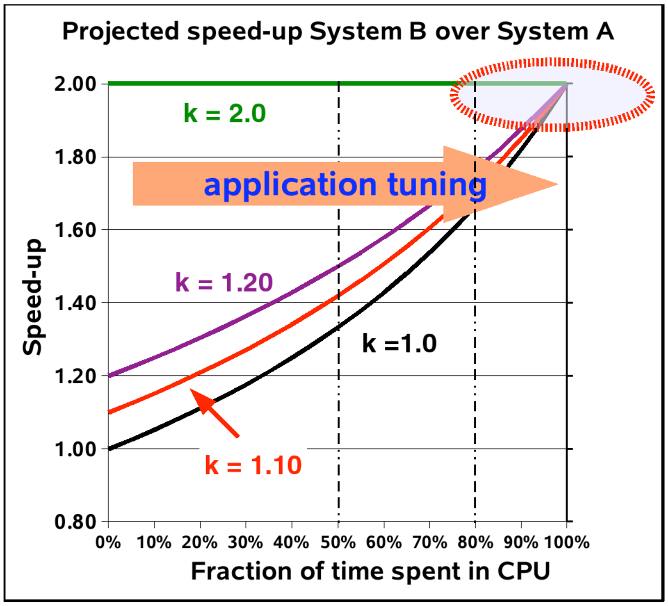
Application Execution Time "T(A)" on System A:

$$T(A) = T(cpu) + T(memory) := f*T(A) + (1-f)*T(A) \quad \text{with } f \in [0,1]$$

The execution time "T(B)" on System B is then given by:

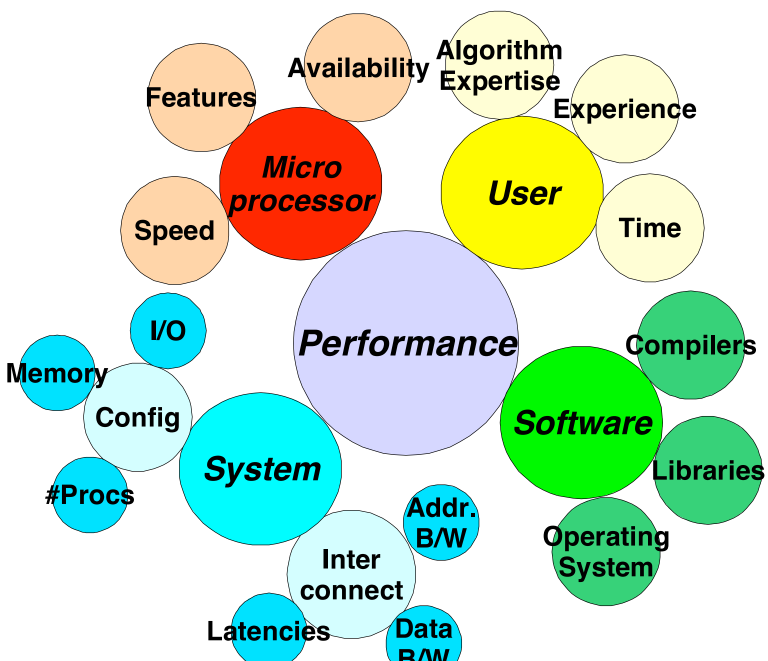
$$T(B) = f*T(A)/n + (1-f)*T(A)/k$$

Oh My!



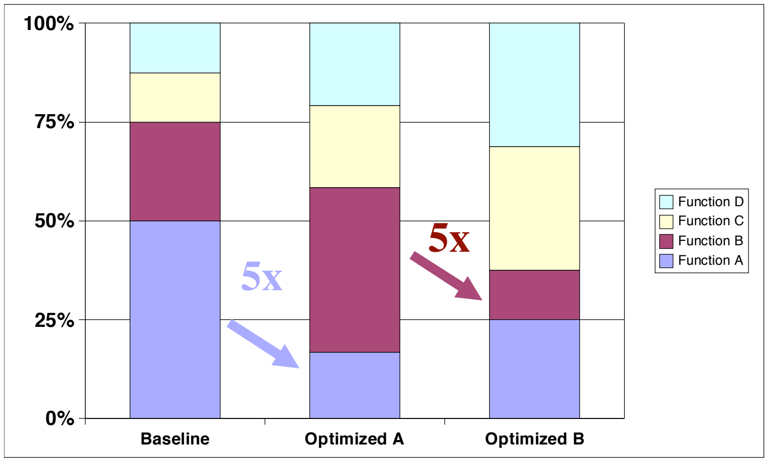
- *We plot the speed-up if the CPU of System B is 2x faster than the CPU of System A*
- *In this chart, "k" denotes the speed-up of the memory system of System B*
- *Different values for "k" are plotted*
- *Only if we spend 80% or more of the time in the CPU we see a significant speed-up!*

Performance Factors



The Effect Of Tuning

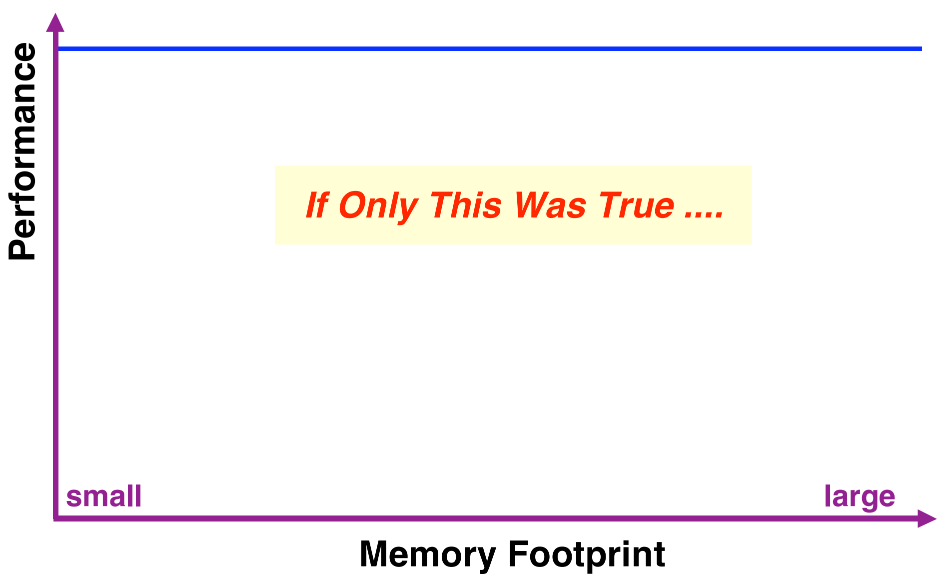
CPU Time Distribution



100 secs   60 secs   40 secs

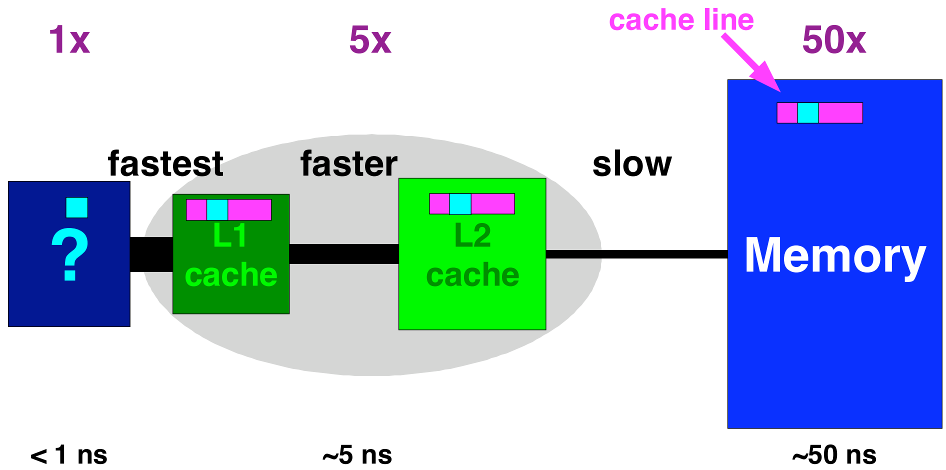
Even though the two routines A and B are 5 times faster each, the total speed-up is "only" 2.5x

Intuitive Performance Graph



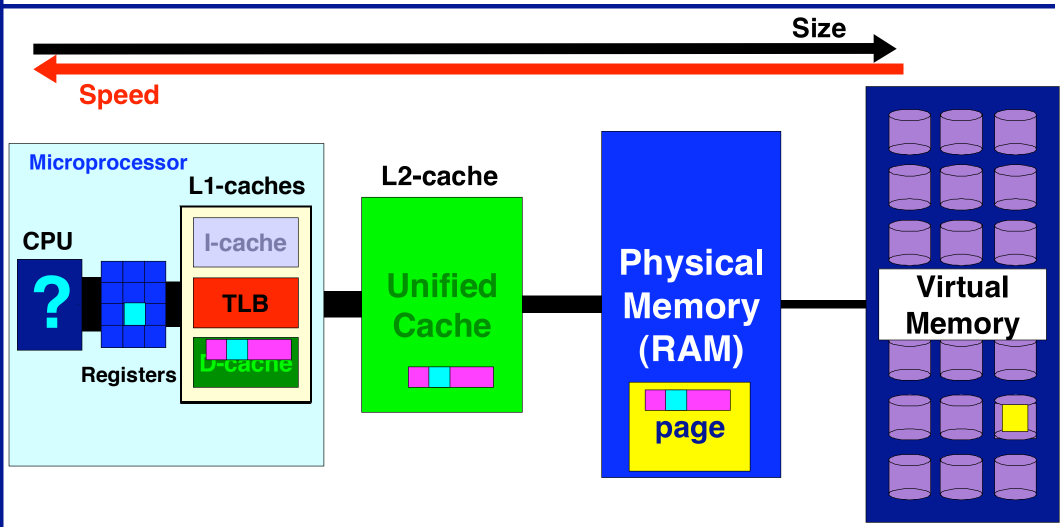
PPCES 2009
RWTH Aachen
March 23-27
2009

A Typical Cache Based System



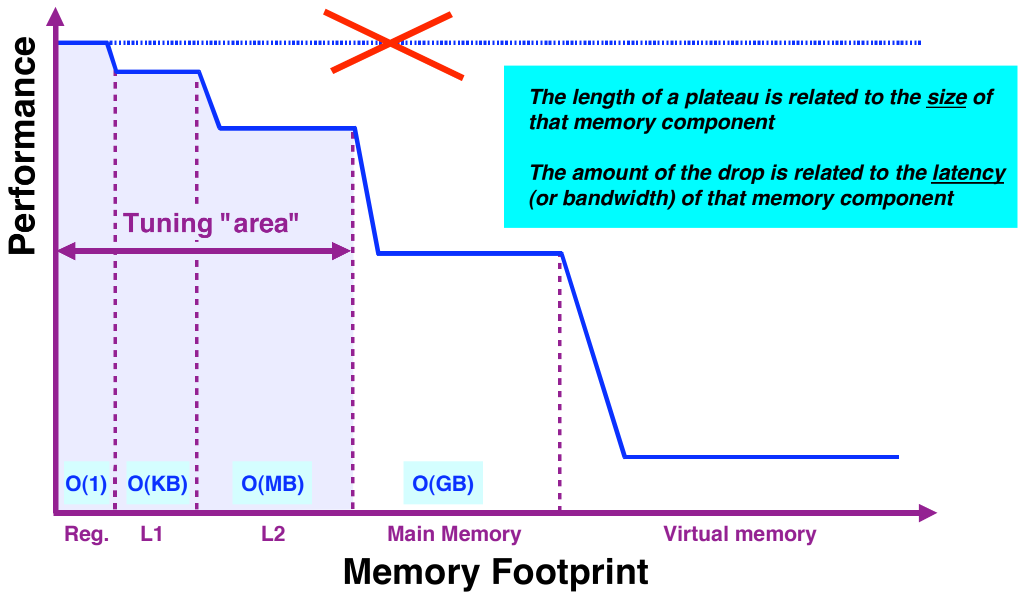
PPCES 2009
RWTH Aachen
March 23-27
2009

The Memory Hierarchy



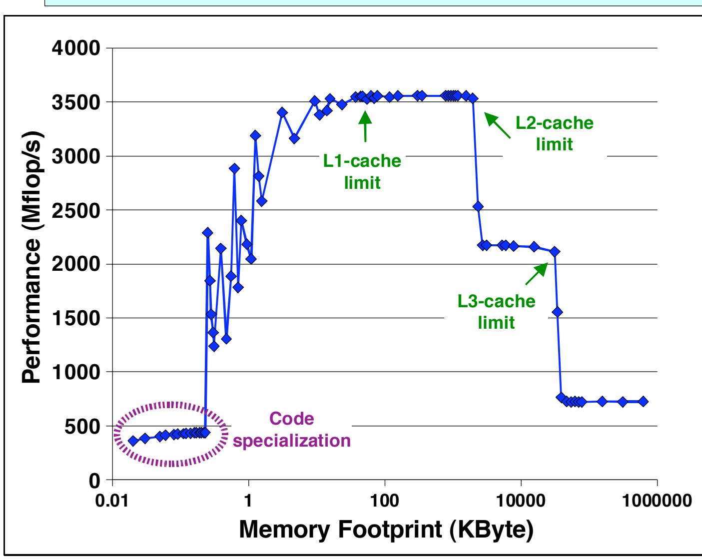
Good Performance:
Get Data Into CPU As Fast As Possible And Keep It There
For As Long As You Can

Performance Is Not Uniform



Example - 13th deg. polynomial

```
for (i=0; i<vlen; i++)
   p[i] = c[0] + q[i]*(c[1] + q[i]*(c[2] + q[i]*(c[3] + ....
```



- *This operation is CPU bound i.e. there are many more floating point operations than memory references*
- *The system realizes 99% of the absolute peak performance !*
- *Note the start-up effect and the performance drop for larger problems*

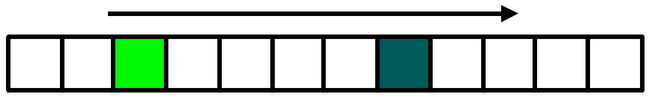
SF E2900 - US IV+ @1800MHz

L1 cache	:	64 KByte
L2 cache	:	2 MByte
L3 cache	:	32 MByte
Peak speed	:	3600 Mflop/s

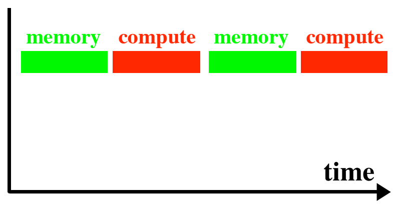
Hiding Memory Latency

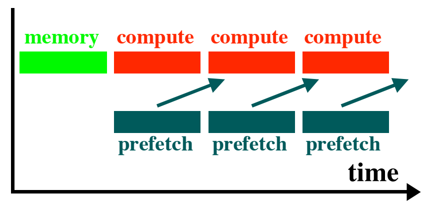
- *The memory access pattern may be predictable:*

  Example: summation of elements



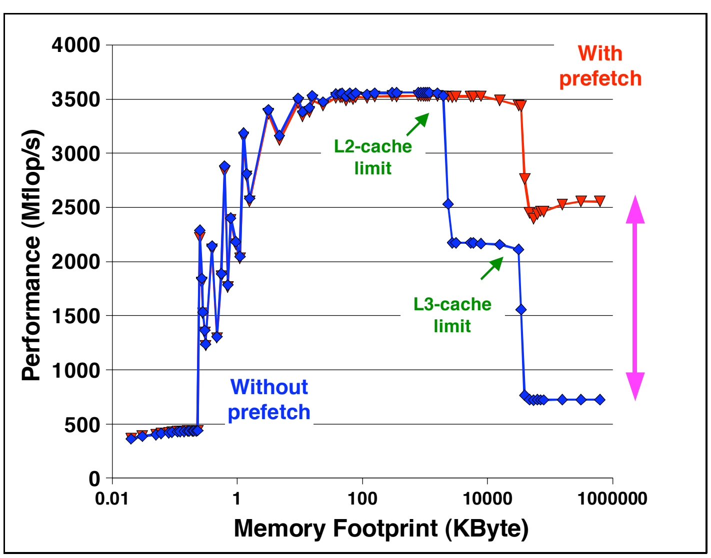
- *With prefetch, one fetches memory before it is needed*
- *This is called a "latency hiding technique"*





Prefetch - 13th deg. polynomial

```
for (i=0; i<vlen; i++)
   p[i] = c[0] + q[i]*(c[1] + q[i]*(c[2] + q[i]*(c[3] + ....
```



- *Re-compiled for automatic prefetch*
- *Performance for L2 resident problem sizes is the same*
- *Hides latency to L3 cache !*
- *For large problem sizes, automatic prefetch is a big win !*

SF E2900 - US IV+ @1800MHz
L1 cache : 64 KByte
L2 cache : 2 MByte
L3 cache : 32 MByte
Peak speed : 3600 Mflop/s

Five Different Ways To Optimize

1. *Operating System features*
   - ✔ *Effort: nothing, just use them*
2. *Faster libraries*
   - ✔ *Effort: relink your application or use environment variables*
3. *The compiler*
   - ✔ *Effort: read*
4. *Source code changes*
   - ✔ *Effort: "unlimited"*
5. *Parallelization*
   - ✔ *Effort: anything between trivial and substantial*

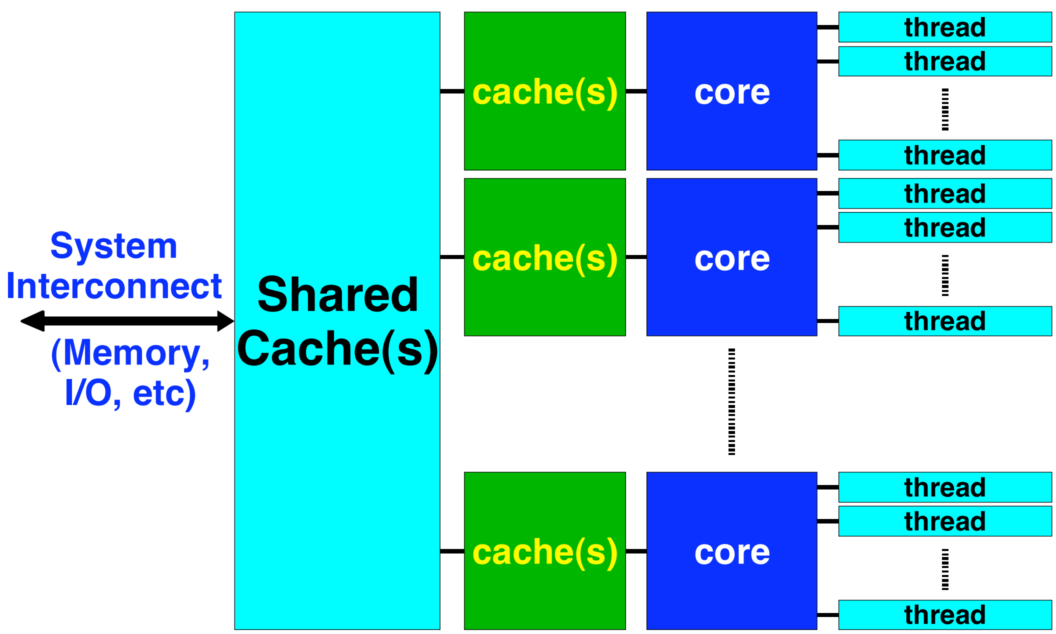
In practice one tends to use a combination of all of these five

Multicore Processor Architectures

What is a Multicore Architecture ?

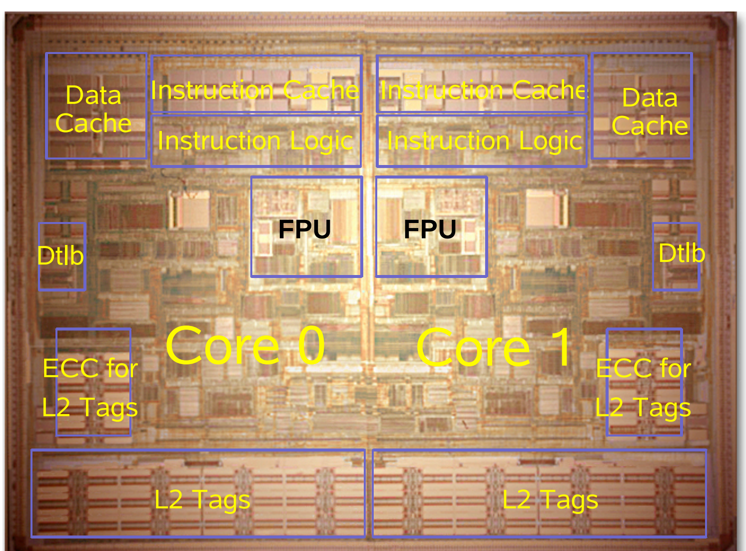
- ❑ *In a Multicore processor, there is more than "one" core*
  - *A "core" is not well defined - A (very) simplified view is to see it as a CPU*
- ❑ *Different implementations possible and available*
  - *Could be two levels of parallelism for example*
  - *Like Sun's UltraSPARC™ T1 or T2 processor*
    - ✔ ***Multiple cores and multiple threads within one processor***
- ❑ *Often, there is also a cache hierarchy of private and shared caches*
- ❑ *For the developers, it mostly matters there is hardware parallelism at the chip level they can take advantage of*

A Generic Multicore Architecture

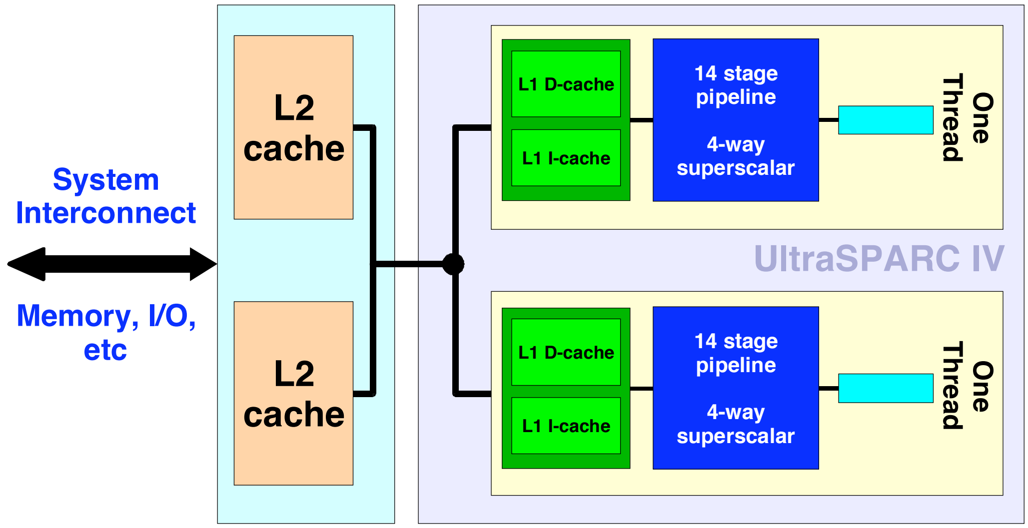


The UltraSPARC IV Processor

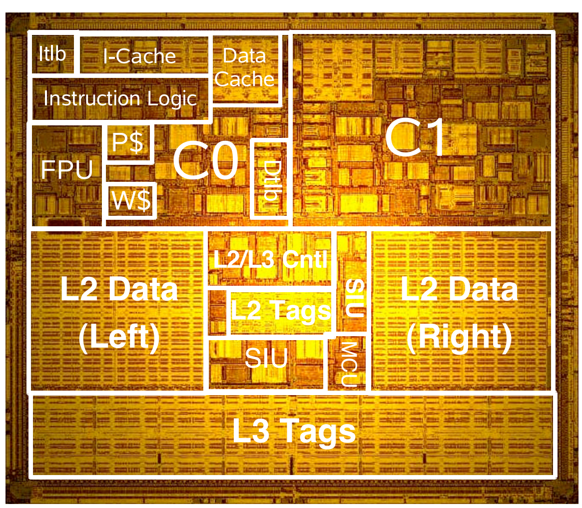
The UltraSPARC IV Processor



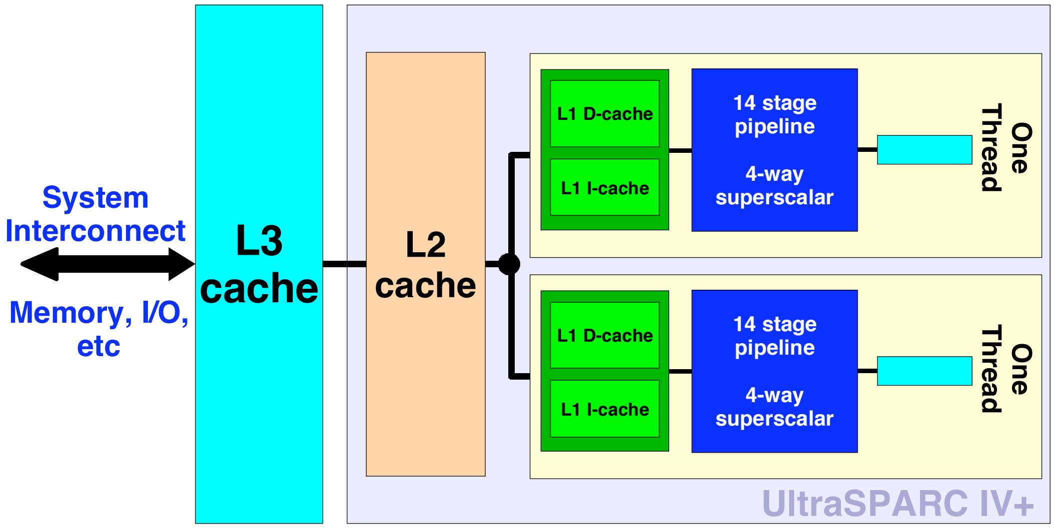
UltraSPARC IV - Block diagram



The UltraSPARC IV+ Processor



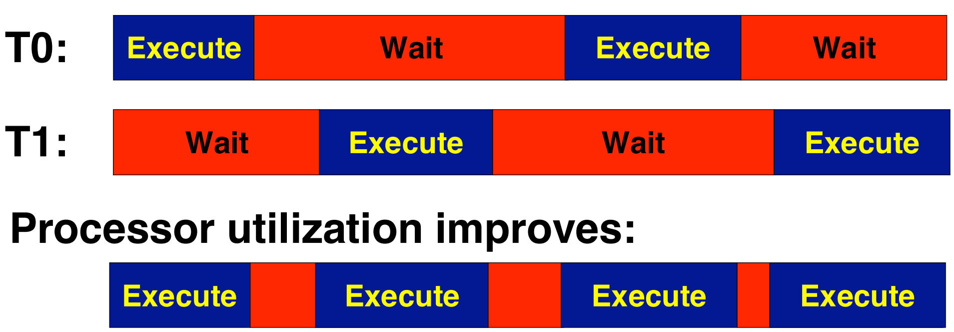
UltraSPARC IV+ - Block diagram



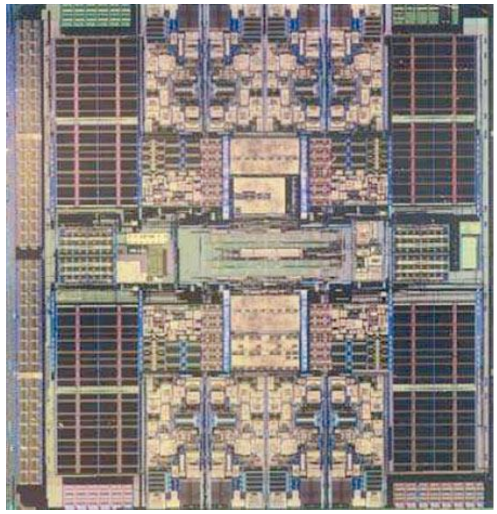
Throughput Computing

Key Concept

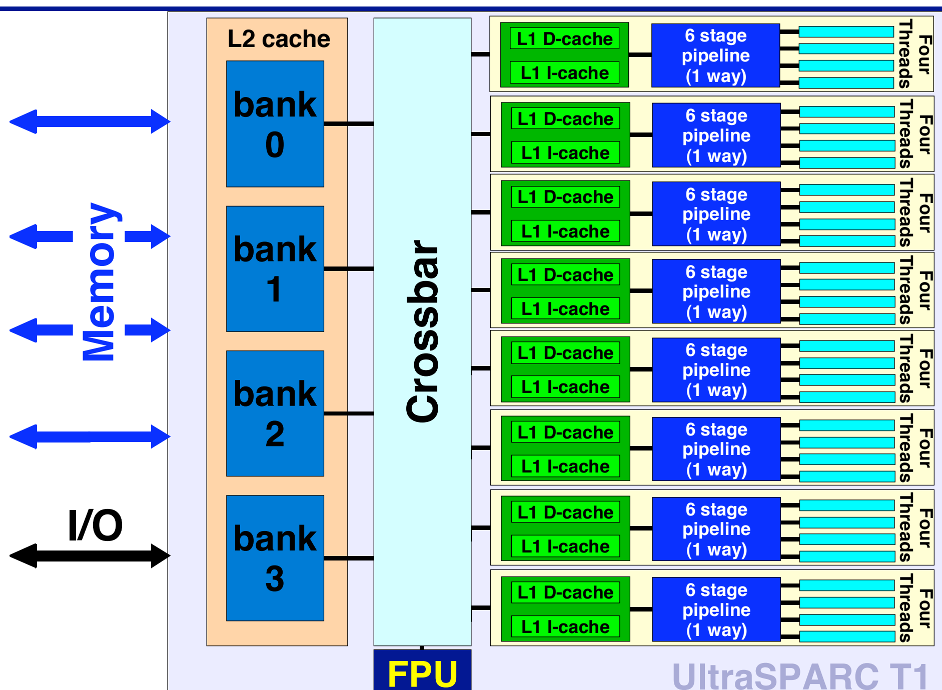
Other threads continue while one or more threads wait for a resource



The UltraSPARC T1 Processor

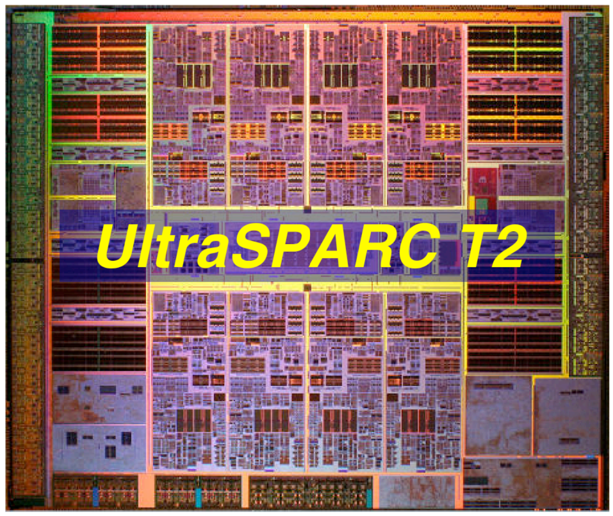


UltraSPARC T1 - Block diagram



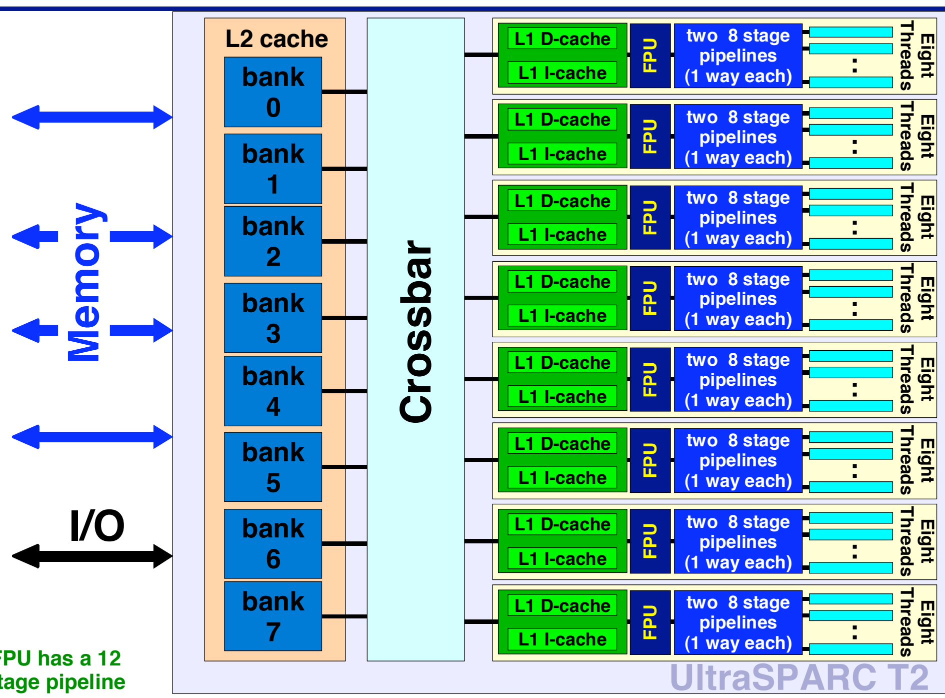
Sun microsystems

8 cores, 64 threads
60+ GB/s Mem. BW



UltraSPARC T2 - Block diagram

Sun microsystems

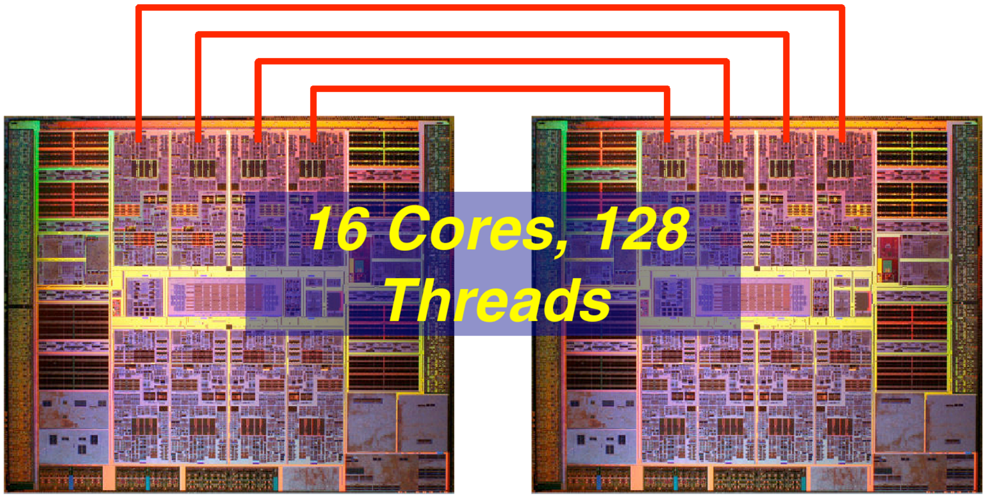


FPU has a 12 stage pipeline

PPCES 2009
RWTH Aachen
March 23-27
2009
29

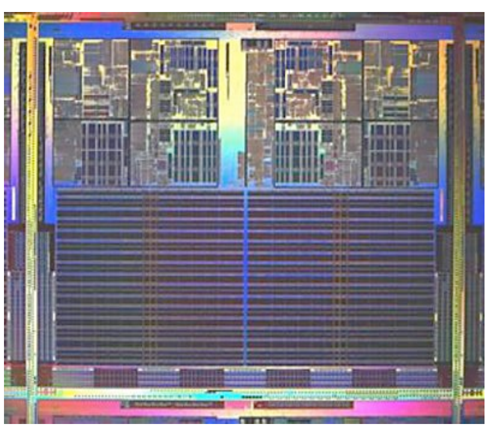
UltraSPARC T2 Plus

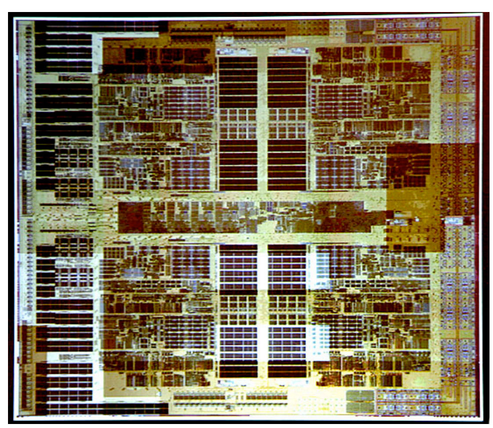
Four Cache Coherence Links (50+ GB/s)



PPCES 2009
RWTH Aachen
March 23-27
2009
30

AMD Opteron

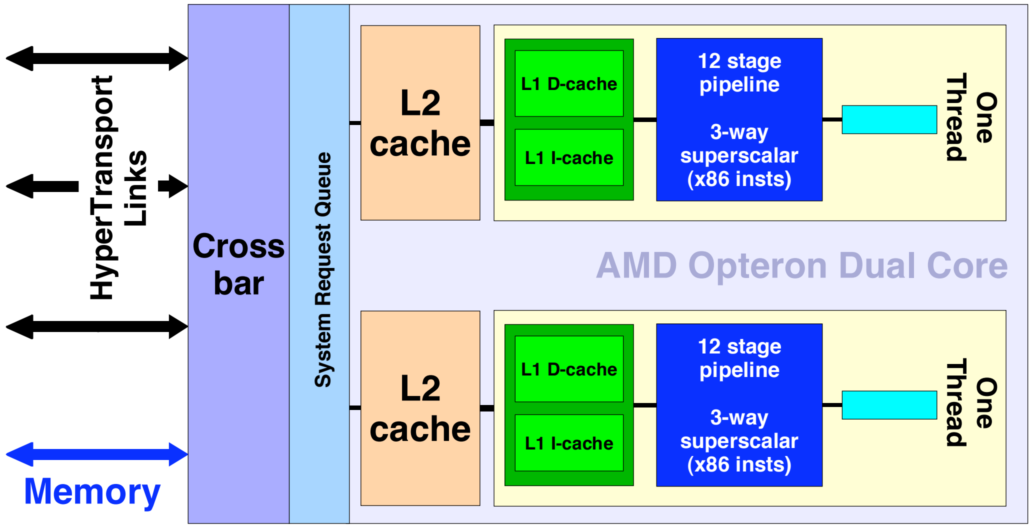




PPCES 2009
RWTH Aachen
March 23-27
2009

31

AMD Opteron - Dual core

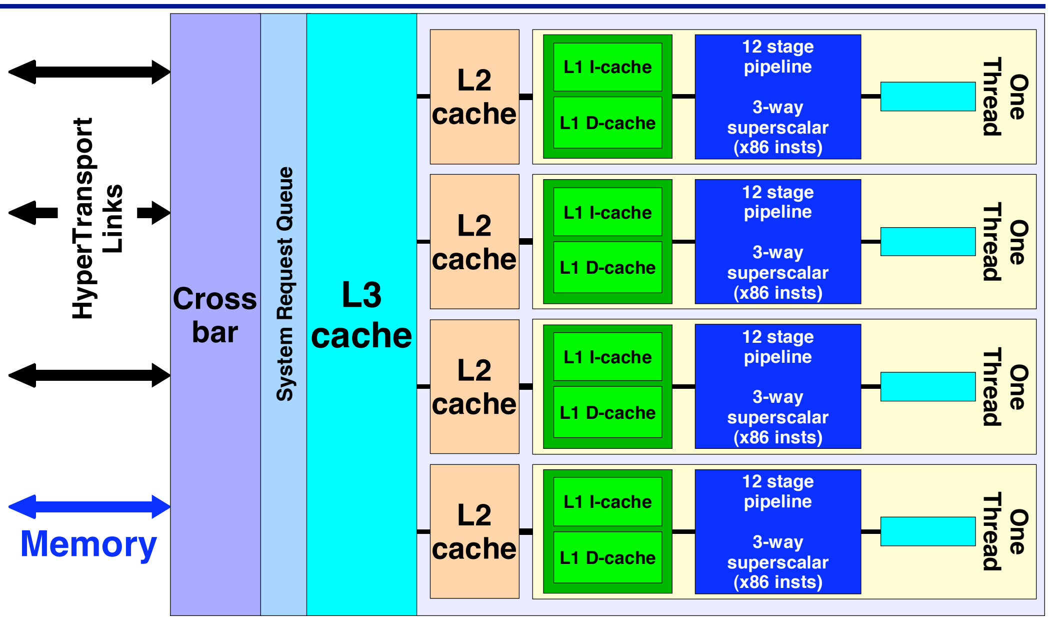


RvdP/V1 Parallel Programming Intro

PPCES 2009
RWTH Aachen
March 23-27
2009

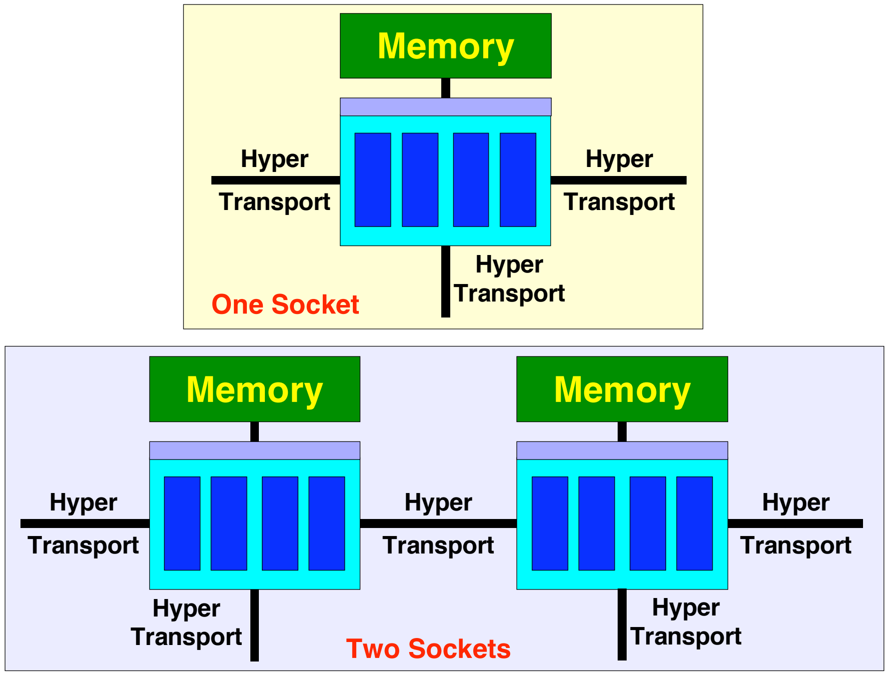
32

AMD Opteron - Quad core

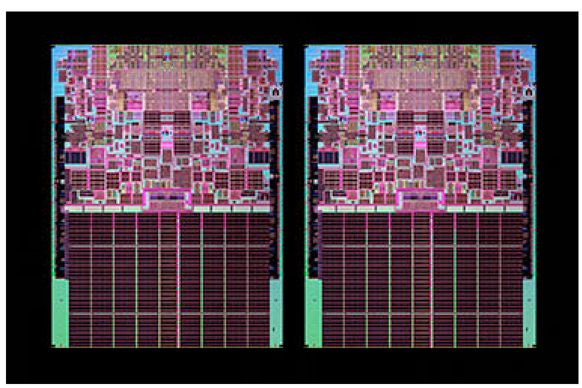


RvdP/V1 Parallel Programming Intro

The Hypertransport Interconnect

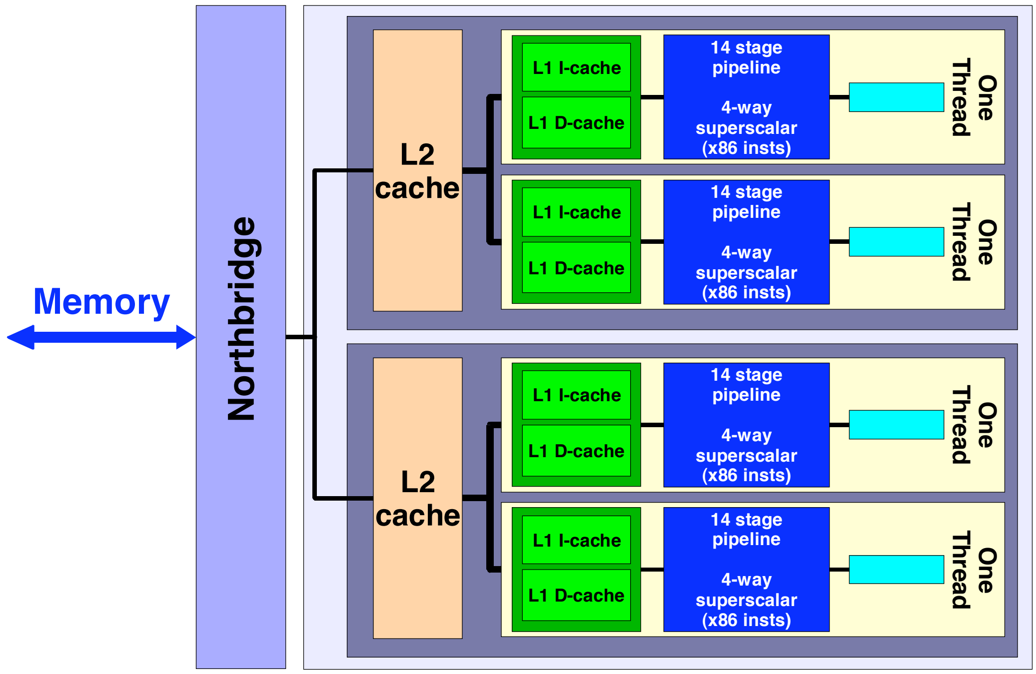


Intel Xeon 5300 ('Clovertown')



1

Intel Xeon 5300



Parallel Programming Intro

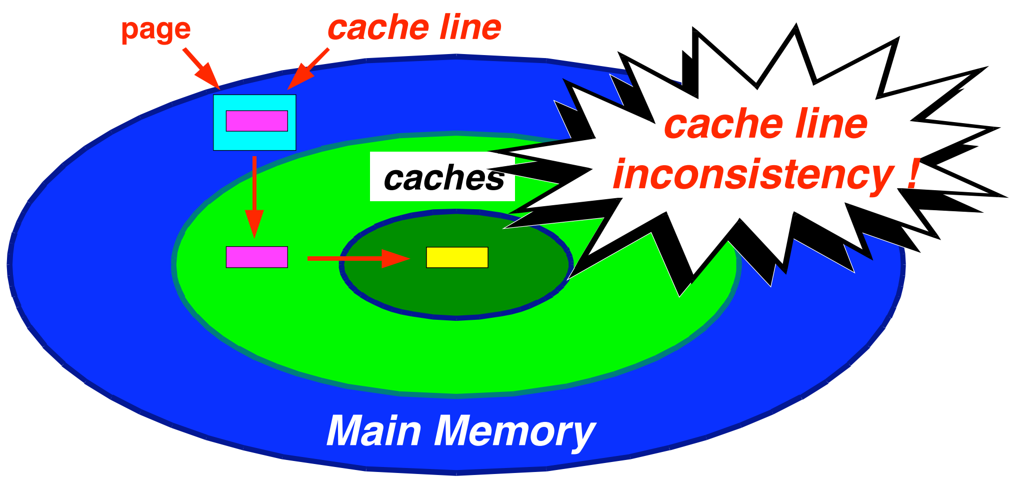
Summary

- *Multicore has arrived and is here to stay*
- *Substantial differences between architectures*
  - *Number of cores*
  - *Number of threads per core*
  - *Cache organization*
    - ✔ *Caches private to one core*
      - *Typical at the L1 level (instruction, data, TLB)*
    - ✔ *Shared caches*
      - *Could be more than one*
      - *How many cores share one cache*
- *To the developer this means that about every processor is, or soon will be, a (small) parallel computer*

Parallel Architectures

PPCES 2009
RWTH Aachen
March 23-27
2009

Cache line modifications

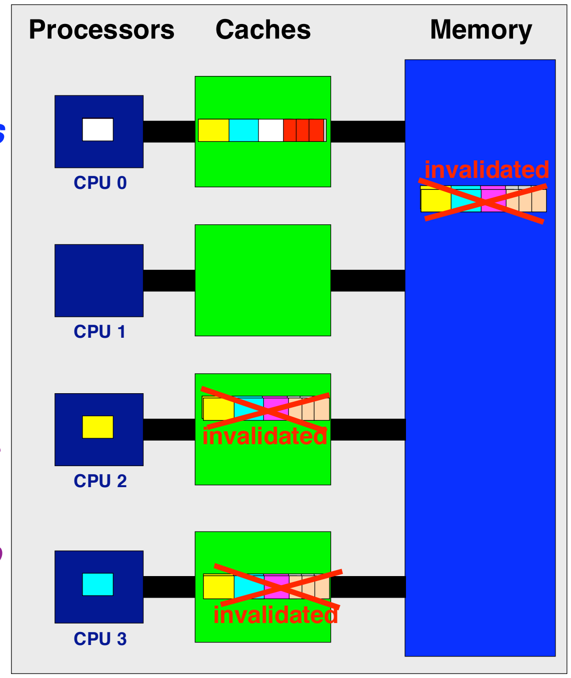


Caches in a parallel computer

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

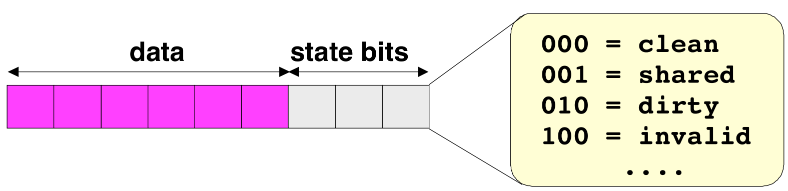
Cache Coherence ("cc"):

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



Cache Coherence ("cc")

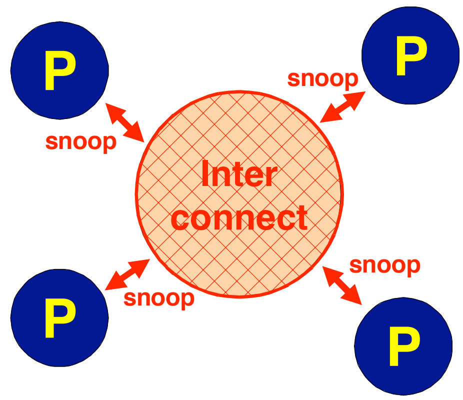
- *<u>Needed in a write-back cache organization</u>*
- *Keeps track of the status of cache lines*
- *This is called the "state" information*



- *The system uses signals ("coherence traffic") to update the status bits of cache lines*
- *Cache Coherence is a very convenient feature to have*
- *It makes it possible to build <u>efficient</u> shared memory parallel systems*

Snoopy based Cache Coherence

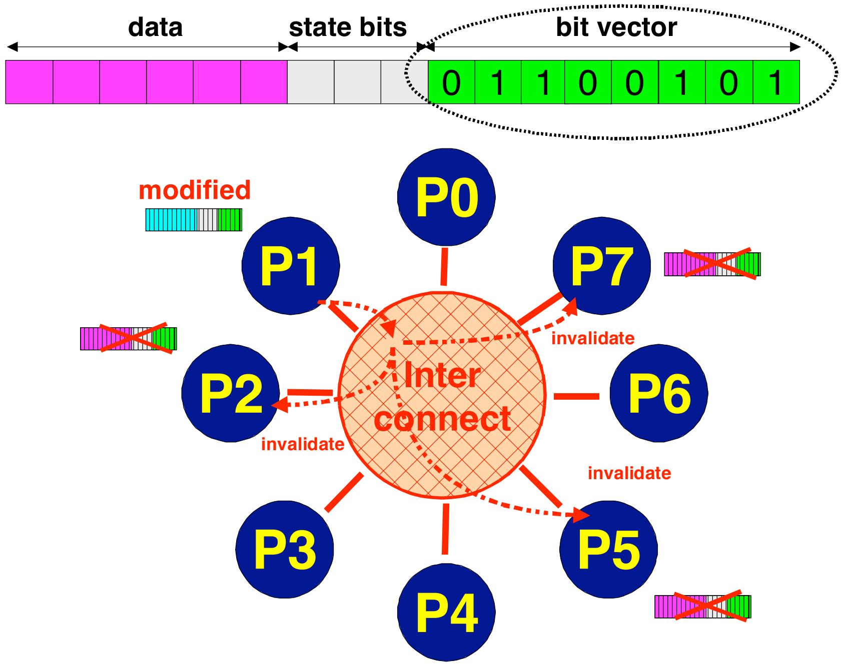
- Also called "Broadcast" cache coherence
  - All addresses sent to all devices
  - Result of the snoop is computed in a few cycles
- Advantages:
  - Low latency in general
  - Fast cache-to-cache transfers
- Disadvantage
  - Data bandwith limited by snoop bandwidth
  - Difficult to scale to a large number of processors



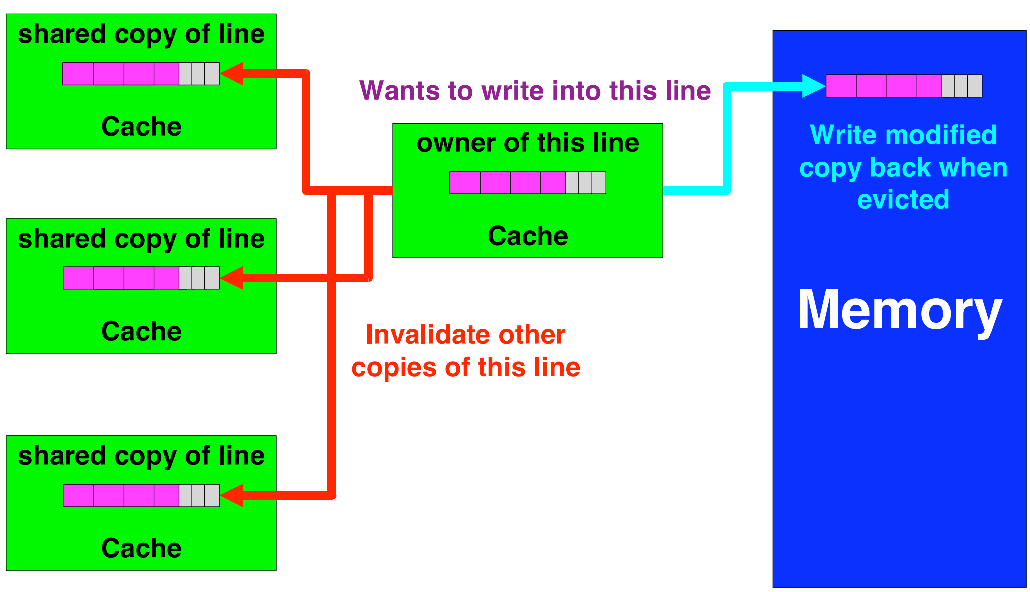
Directory based Cache Coherence

- Also called SSM (Scalable Shared Memory)
- This is a point-to-point protocol
- Through a directory, the system keeps track which processor(s) are involved in a particular line
- Address requests sent to specific caches only
- Advantages:
  - Bandwidth can be much greater
  - Scalable to large processor count
- Disadvantages
  - Latency is usually longer and no longer uniform
  - Slower cache-to-cache transfers
  - Need to store the additional directory entries

Example SSM

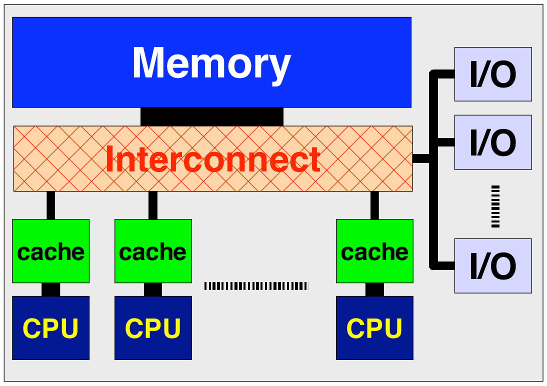


The Owner



Note: other caches interested in a modified line get it from the owner

Uniform Memory Access (UMA)



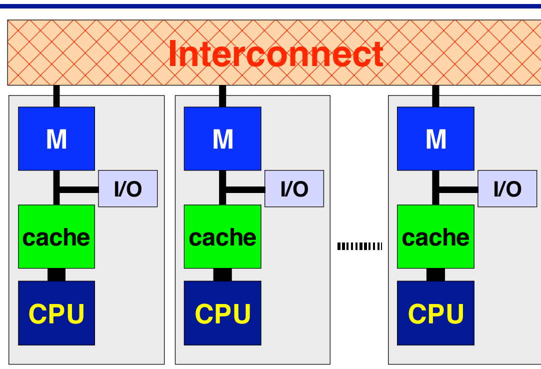
- Also called "SMP" (Symmetric Multi Processor)
- Memory Access time is Uniform for all CPUs
- CPU can be multicore
- Interconnect is "cc":
  - Bus
  - Crossbar
- No fragmentation - Memory and I/O are shared resources

Pro
- ✔ Easy to use and to administer
- ✔ Efficient use of resources

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

PPCES 2009
RWTH Aachen
March 23-27
2009

NUMA



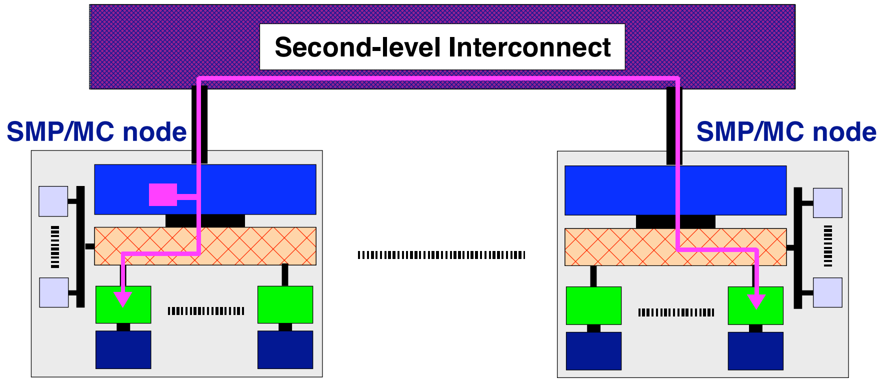
- Also called "Distributed Memory" or NORMA (No Remote Memory Access)
- Memory Access time is Non-Uniform
- Hence the name "NUMA"
- Interconnect is not "cc":
  - Ethernet, Infiniband, etc, ...
- Runs 'N' copies of the OS
- Memory and I/O are distributed resources

Pro
- ✔ Said to be cheap
- ✔ Said to be scalable

Con
- ✔ Difficult to use and administer
- ✔ In-efficient use of resources

PPCES 2009
RWTH Aachen
March 23-27
2009
47

Cluster of SMP/Multicore nodes

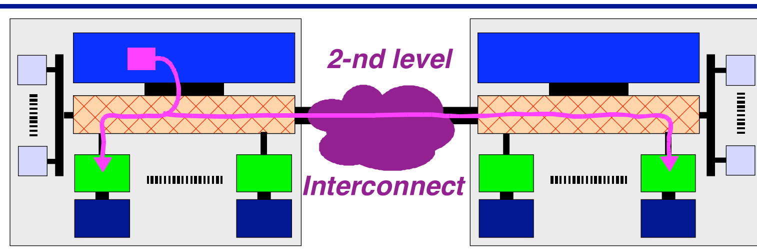


- ***Second-level interconnect is not cache coherent***
  - ***Ethernet, Infiniband, etc, ...***
- ***Hybrid Architecture with all Pros and Cons:***
  - ***UMA within one SMP/Multicore node***
  - ***NUMA across nodes***

RvdP/V1 Parallel Programming Intro

PPCES 2009
RWTH Aachen
March 23-27
2009
48

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***

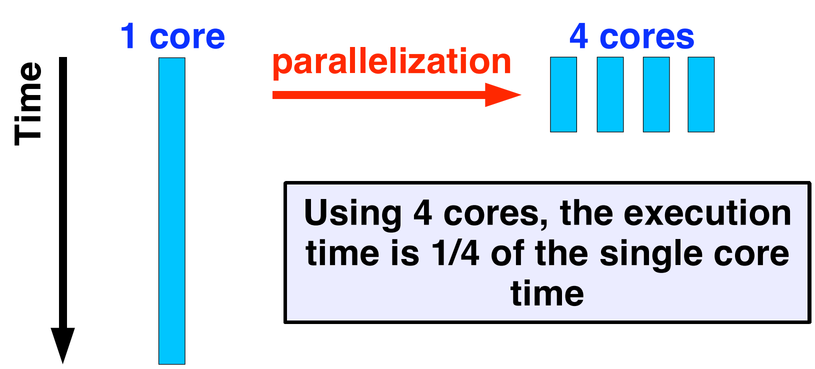
RvdP/V1 Parallel Programming Intro

Parallel Programming Basics

Why Parallelization ?

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

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



What Is Parallelization ?

"Something" is parallel if there is a certain level of independence in the order of operations

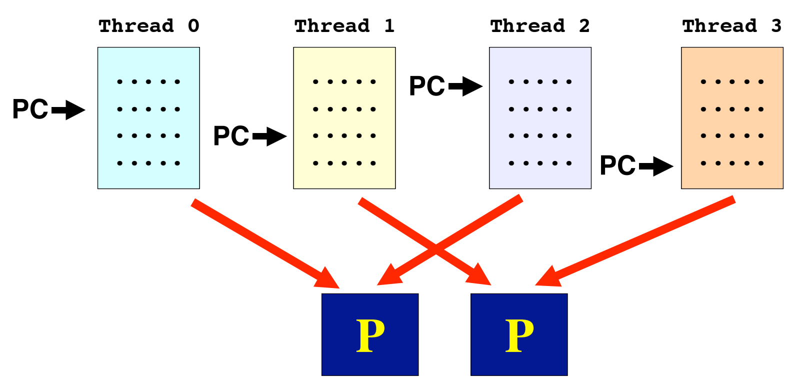
In other words, it doesn't matter in what order those operations are performed

- ***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***



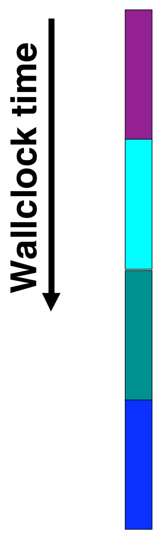
Parallel Overhead

- The goal is to reduce the wall clock time
- 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
- Efficient parallelization is about minimizing the communication overhead

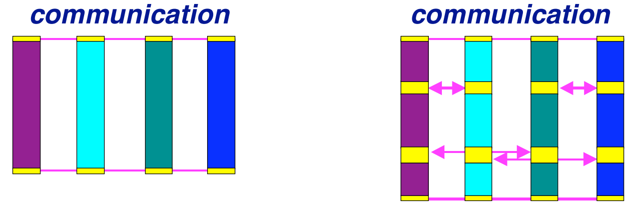
PPCES 2009
RWTH Aachen
March 23-27
2009

Communication

Serial Execution



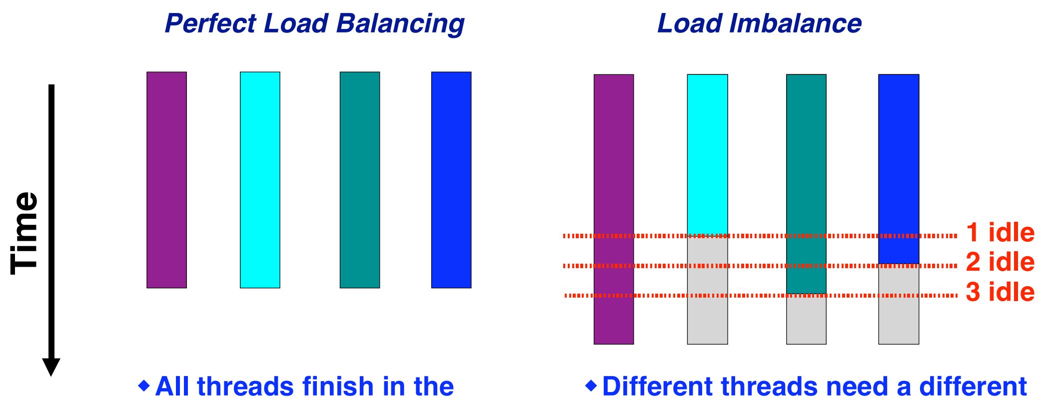
Parallel - Without communication | Parallel - With communication



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

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

Load balancing

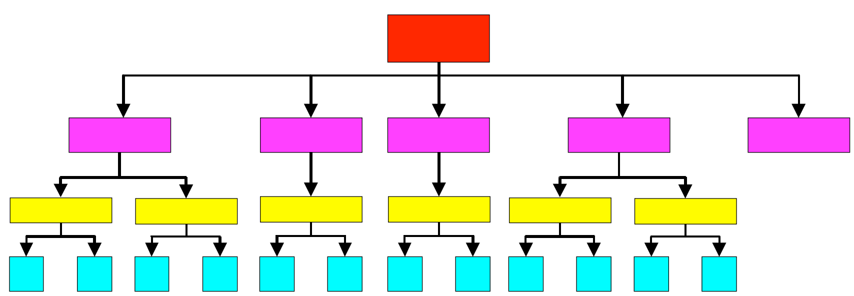


- All threads finish in the same amount of time
- No threads is idle

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

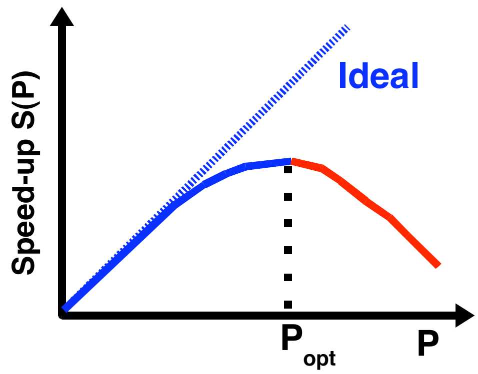
Thread is idle

Different levels of parallelism



- *Parallelization at the highest ( ) level:*
  - *Low communication cost*
  - *Limited to 5 processors only*
  - *Potential load balancing issue*
- *Parallelization at the lowest ( ) level:*
  - *Higher communication cost*
  - *Not limited to a certain number of processors*
  - *Load balancing probably less of an issue*

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)*

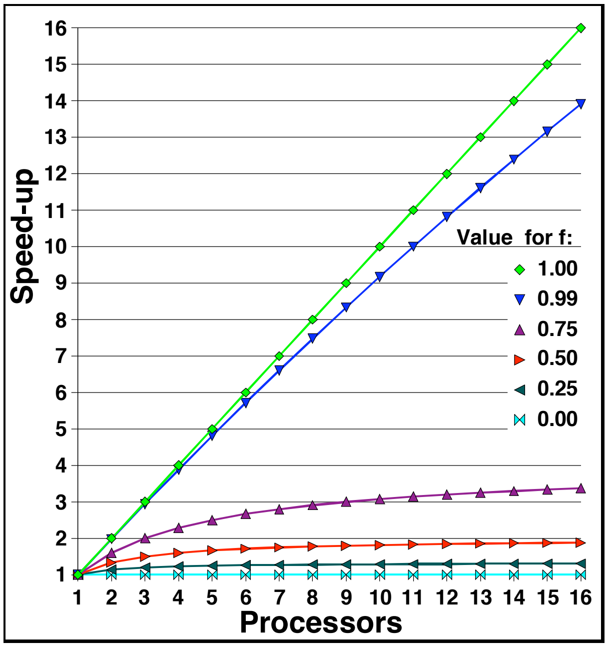
Amdahl's law:

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

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*
- *Use the performance analyzer to find these parts*

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
```


Numerical results

Consider:
A = B + C + D + E

Serial Processing

```
A = B + C

A = A + D

A = A + E
```

Parallel Processing

Thread 0	Thread 1
`T1 = B + C`	`T2 = D + E`
`T1 = T1 + T2`	

- *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 ....*

Parallel Programming Models

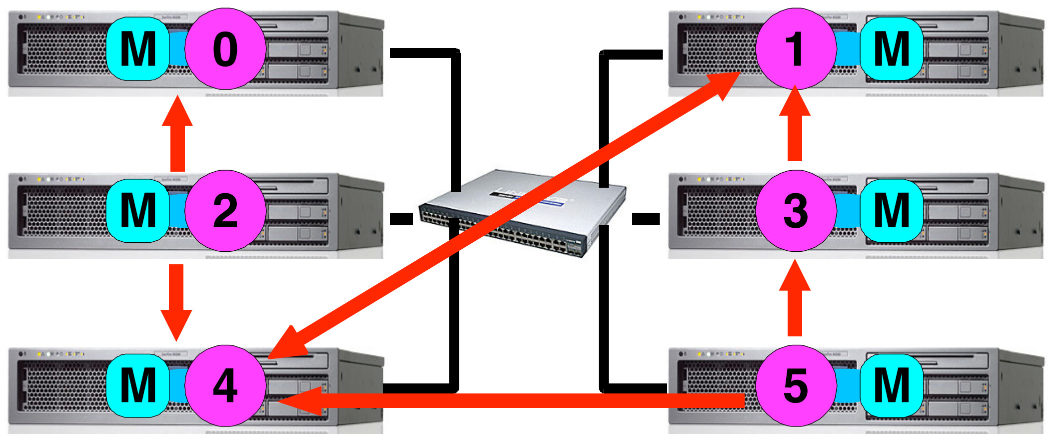
How To Program A Parallel Computer?

- There are numerous parallel programming models
- The ones most well-known are:
  - A Cluster Of Systems ("Distributed Memory")
    - ✔ Sockets (standardized, low level)
    - ✔ PVM - Parallel Virtual Machine (obsoleted)
    - ✔ MPI - Message Passing Interface (de-facto standard)
  - A Single System ("Shared Memory")
    - ✔ Native Threading Model (standardized, low level)
    - ✔ OpenMP (de-facto standard)
    - ✔ Automatic Parallelization (compiler does it for you)

Parallel Programming Models
Distributed Memory - MPI

PPCES 2009
RWTH Aachen
March 23-27
2009
65

The MPI Distributed Memory Model

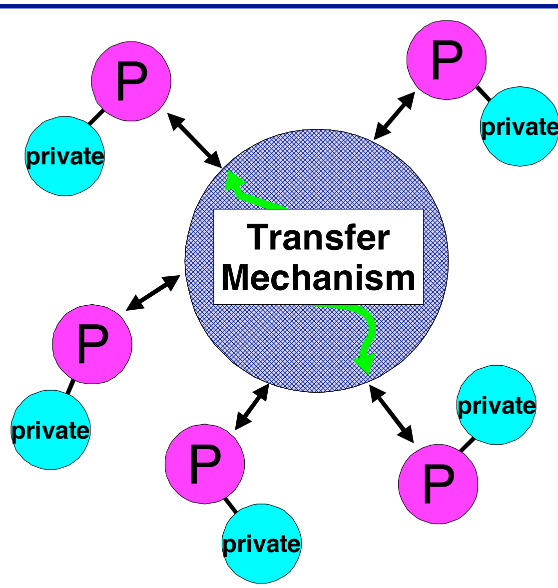


A Cluster Of Systems

RvdP/V1 Parallel Programming Intro

PPCES 2009
RWTH Aachen
March 23-27
2009
66

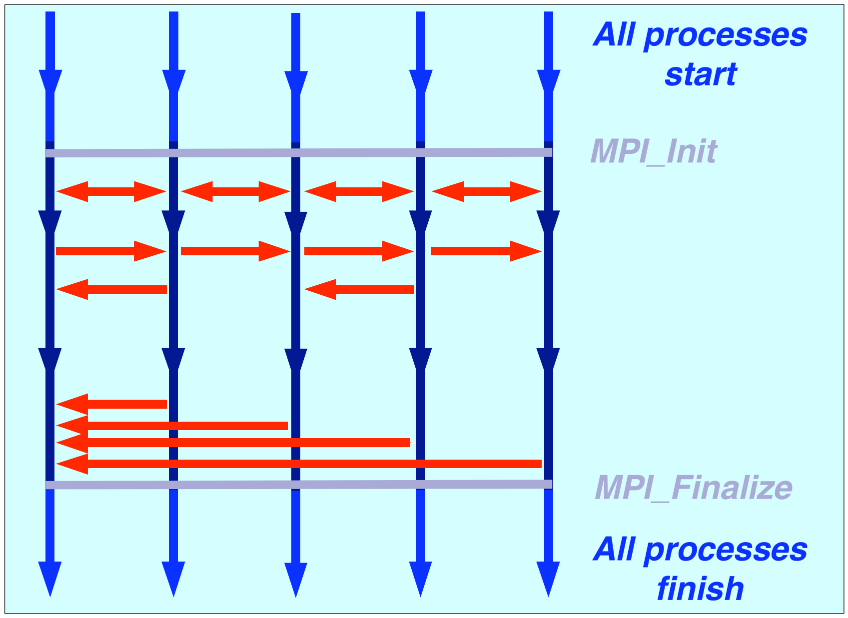
The MPI Memory Model



- ✔ *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*

RvdP/V1 Parallel Programming Intro

The MPI Execution Model/2



= communication

Example - "Hello World"

```
#include <stdio
#include <stdli

#include <mpi.h

int main (int a
{

   int me;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);

   printf("Hello Parallel World, I am MPI process %d\n",me);

   MPI_Finalize();

}
```

```
amd$ mpicc hello-world.c
amd$ mpirun -np 4 ./a.out
Hello Parallel World, I am MPI process 2
Hello Parallel World, I am MPI process 1
Hello Parallel World, I am MPI process 0
Hello Parallel World, I am MPI process 3
amd$
```


Example - Send “N” Integers

```
#include "mpi.h"                  include file
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);
}                                 leave the MPI environment
MPI_Finalize();
```


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

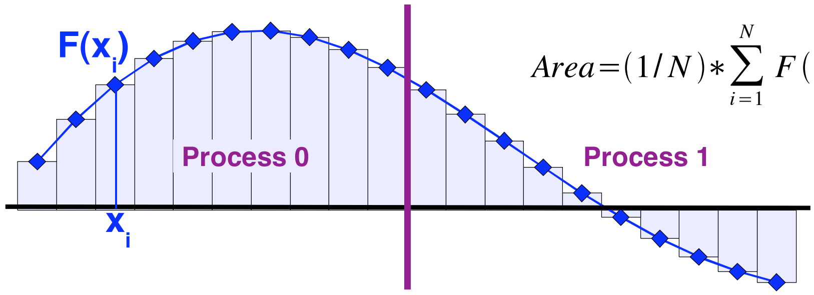
The Pros and Cons of MPI

- 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

A Different Way Of Thinking

- Because of the distributed memory model, a different way of approaching the problem is required
- Have to think about:
  - Dividing the problem into pieces
  - How to distribute the data over the nodes
  - Communication pattern between processes

Example - Numerical Integration

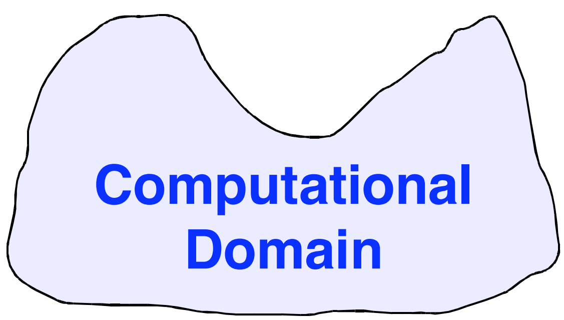


$$Area = (1/N) * \sum_{i=1}^{N} F(x_i)$$

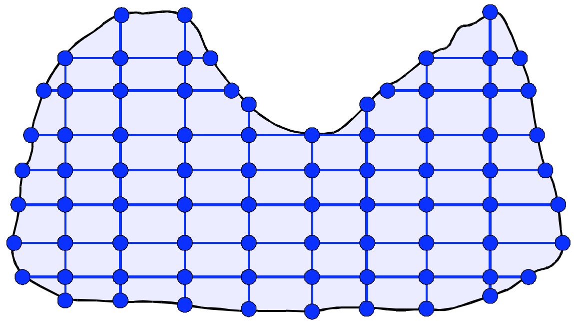
Parallel algorithm using MPI:

1. ***Master process sends number of points to each MPI process***
2. ***Each MPI process then:***
   - ***Defines what set of points to work on***
   - ***Sums up the function values in those points***
   - ***Sends partial sum to main process***
3. ***Master process collects partial sums***
4. ***Master process computes global sum***

Example - Domain Decomposition/1

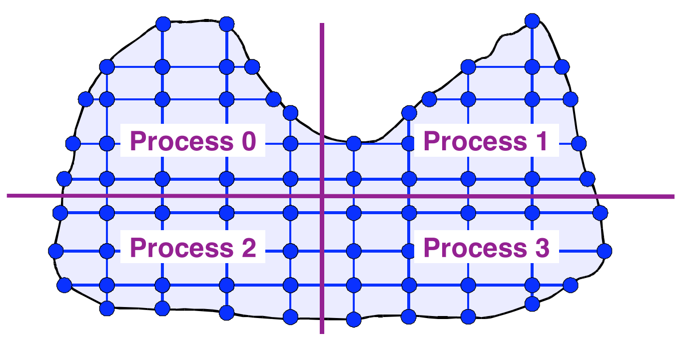


Example - Domain Decomposition/2



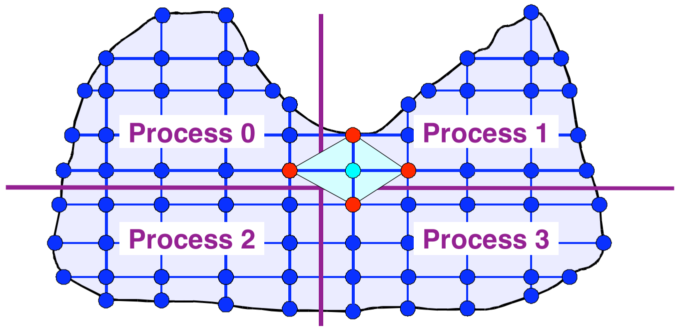
Discretize the domain
Solve problem on the grid points

Example - Domain Decomposition/4



Split domain in disjoint parts
Assign a domain to an MPI process

Example - Domain Decomposition/5



Problems

Some of the data is in the memory of other processes

Communication is needed

Load balancing is another potential issue

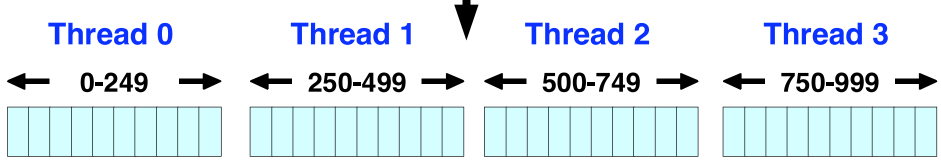
Parallel Programming Models
Shared Memory
Automatic Parallelization

Automatic Parallelization (-xautopar)

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

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

OMP_NUM_THREADS=4



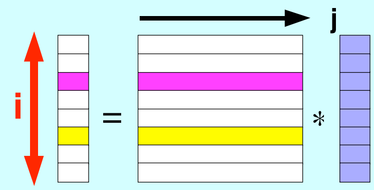
Automatic Parallelization

- Supported on the C, C++ and Fortran compilers
- The parallelization is loop oriented:
  - A loop (nest) is first optimized for serial performance
  - Next, the (nested) loop is analyzed for data dependences and parallelized if safe to do so
- User can check the parallelization messages with the `-xloopinfo` option and/or the `er_src` command
  - The latter gives more elaborate messages
    - ✔ Example: `% er_src -scc parallel funcA.o`

Automatic Parallelization Example

```
 1  void mxv(int m,int n,double *a,double *b[],double *c)
 2  {
 3     for (int i=0; i<m; i++)
 4     {
 5       double sum = 0.0;
 6       for (int j=0; j<n; j++)
 7         sum += b[i][j]*c[j];
 8       a[i] = sum;
 9     }
10  }
```

parallel loop




```
% cc -c -fast -xrestrict -xautopar -xloopinfo mxv.c
"mxv.c", line 3: PARALLELIZED, and serial
         version generated
"mxv.c", line 6: not parallelized, unsafe
         dependence (sum)
```


The Fundamental Problem

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

M = 0 : Parallel
M = 1 : Not Parallel

What to do if you were a compiler ?

PPCES 2009
RWTH Aachen
March 23-27
2009

83

Parallel Programming Models
Shared Memory - OpenMP

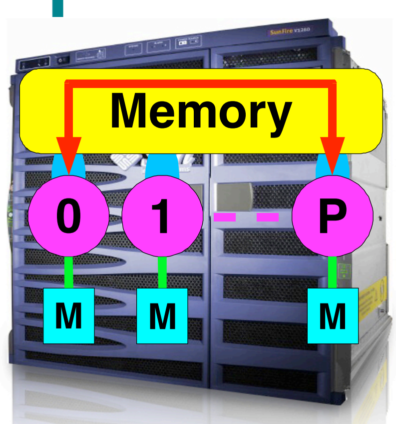
RvdP/V1

Parallel Programming Intro

PPCES 2009
RWTH Aachen
March 23-27
2009

84

OpenMP™



http://www.openmp.org

RvdP/V1

Parallel Programming Intro

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