ORACLE®

An Introduction Into Parallel Computing with OpenMP

Ruud van der Pas
Senior Staff Engineer - Architecture and Performance
SPARC Microelectronics, Oracle, Santa Clara, CA, USA

Outline

- Introduction Parallel Computing
- Parallel Programming Models
- Using OpenMP
- Performance Considerations in Parallel Computing

Introduction Parallel Computing

A comprehensive white paper on parallel programming

An Oracle White Paper
April 2010

Parallel Programming with
Oracle® Developer Tools

http://www.oracle.com/technetwork/systems/parallel-programming-oracle-develop-149971.pdf

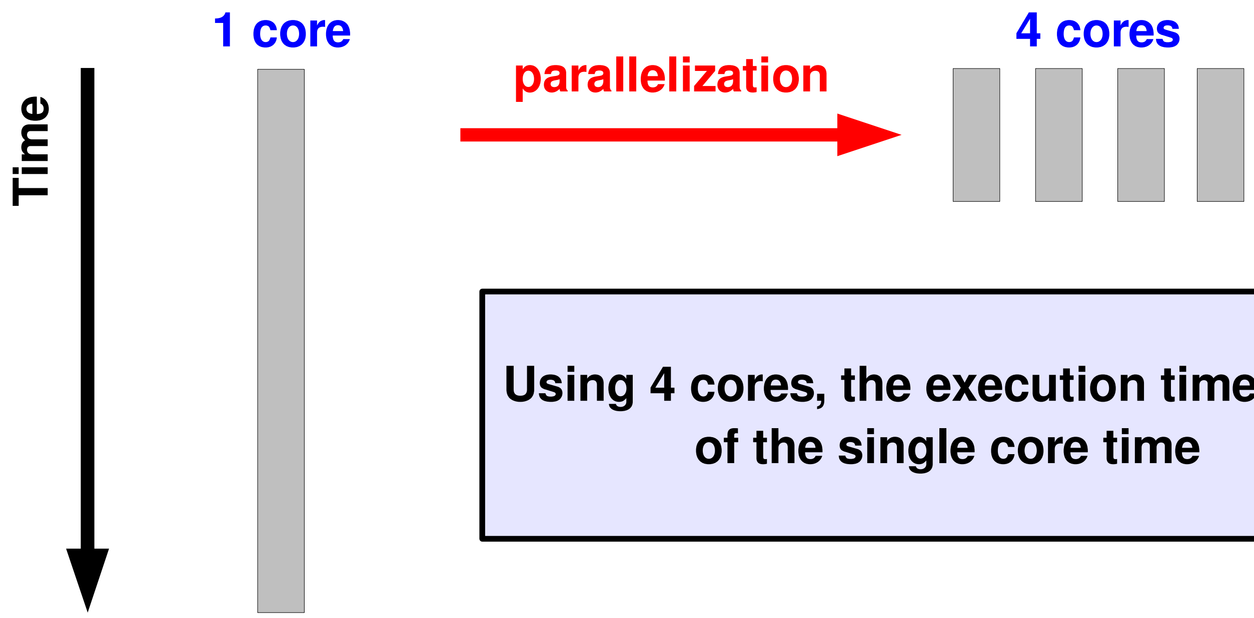
See also:
http://blogs.sun.com/ruud

ORACLE

Why Parallelization ?

Parallelization is a performance 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 ?

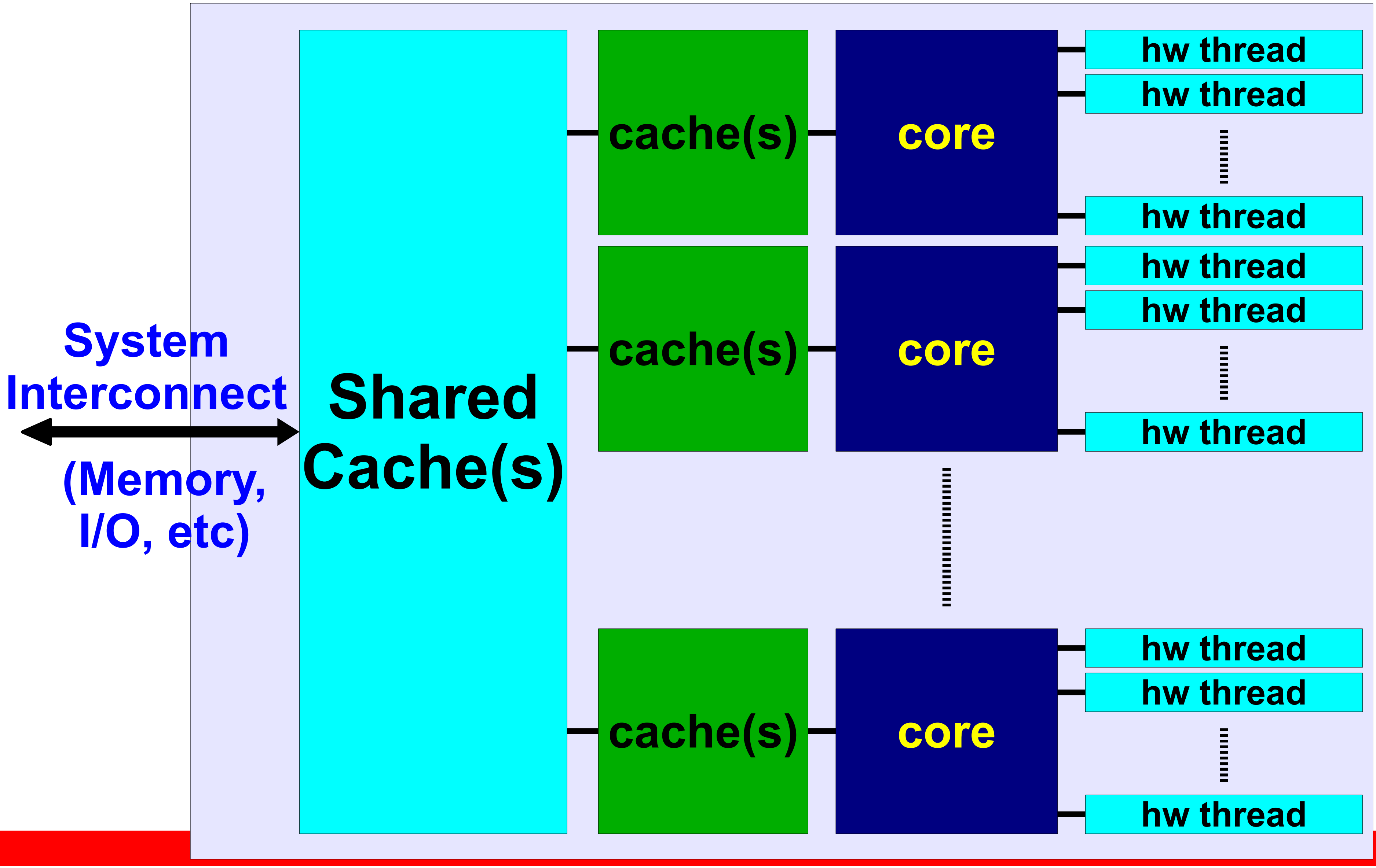
"Something" is parallel if there is a <u>certain level of independence</u> in the order of operations

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

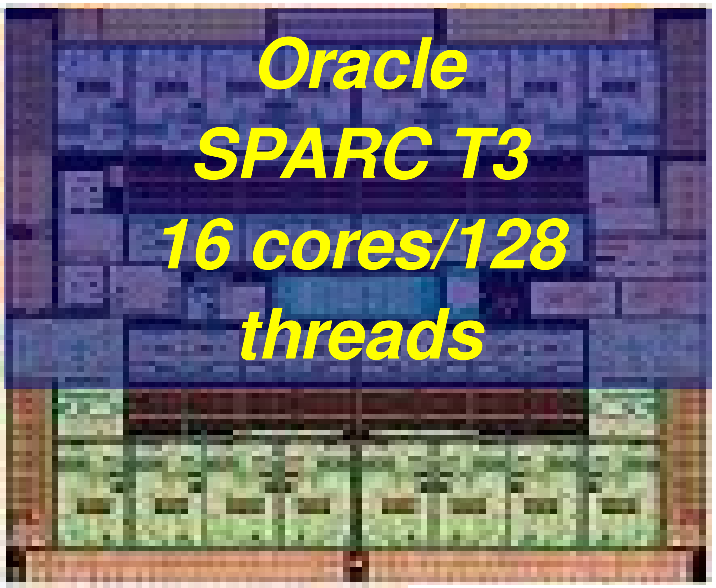
- ***A sequence of machine instructions***
- ***A collection of program statements***
- ***An algorithm***
- ***The problem you're trying to solve***

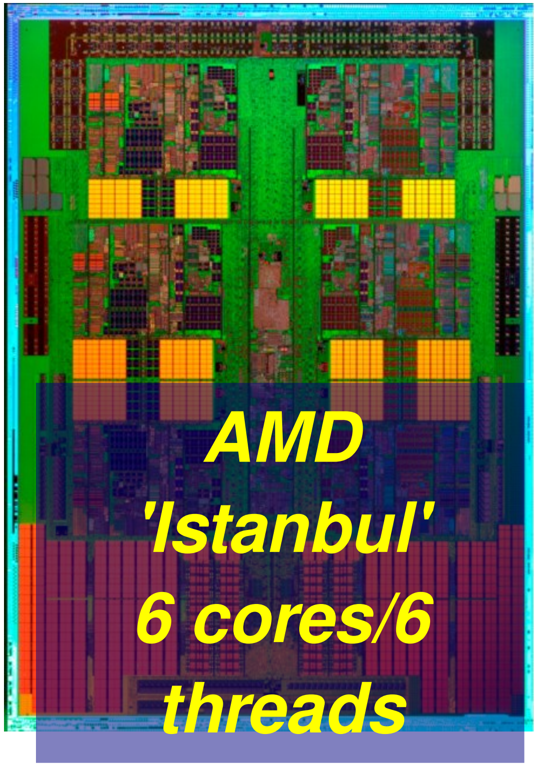
granularity

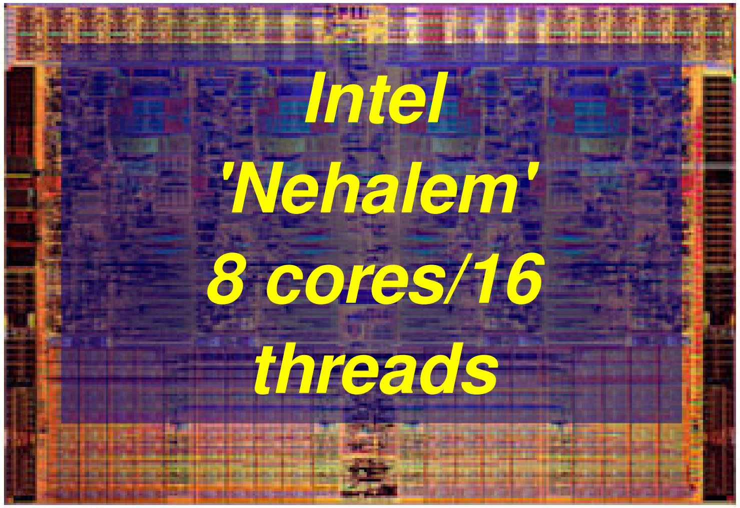
A Generic Multicore Architecture



Some Real World Examples

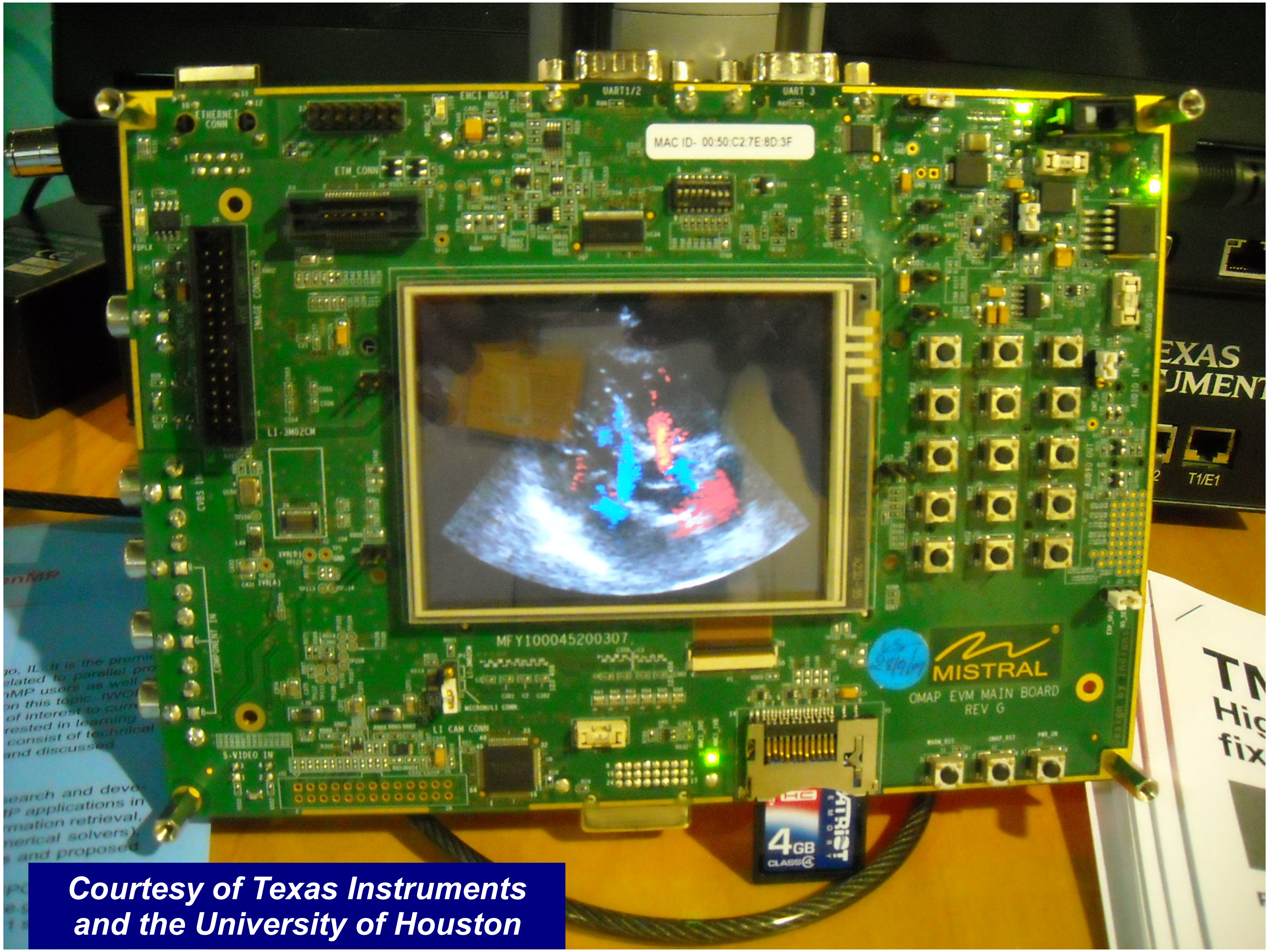






ORACLE

OpenMP Demo At SC'10 in New Orleans



Courtesy of Texas Instruments and the University of Houston

This Is Even More Real World !

LG

December 16, 2010 02.28pm EST | 4 Comments

LG Announces Optimus 2X, World's First Dual-Core Phone

By Peter Pachal

0 Digg Tweet 11 Share 134 Submit Email Print

LG announced today the first smartphone with a dual-core chip, the Optimus 2X. Dual-core phones, which pack more processing power than the current single-core models, are expected to be a major trend for cell phones in 2011.

The Optimus 2X's 1GHz processor, Nvidia's Tegra 2 chip, enables features like HDMI mirroring, which lets you duplicate whatever's on the phone's 4-inch WVGA display, including games, at full HD resolution on an external screen. It's also said to provide smoother Web browsing and allow multitasking of applications with virtually no screen lag.

"Dual-core technology is the next leap forward in mobile technology so this is no small achievement to be the first to offer a smartphone utilizing this technology," Jong-seok Park, CEO LG Electronics Mobile, said in a statement. "With unique features such as HDMI mirroring and exceptional graphics performance, the LG Optimus 2X is proof of LG's commitment to high-end smartphones in 2011."

LG's dual-core announcement is the first in what will probably be the first in a long-line of dual-core mobile devices from several manufacturers. Earlier this month Texas Instruments released detailed specs about its

Loading "http://www.pcmag.com/article2/0,2817,2374435,00.asp", completed 144 of 145 items

Use Of A Parallel/Multicore System

Throughput/Workload

Run multiple programs at the same time

For example, having a chat session while watching a video

Parallel Processing

Focus of this talk

Make one single program run faster

For example, speed up a video processing application

Parallel Programming Models

Computer Instructions/1

A computer program, written in C, to compute the average of a sequence of numbers:

```
double average(int n, double data[])
{
   double sum = 0.0;

   for (int i=0; i<n; i++)
       sum = sum + data[i];

   return(sum/n);
}
```

This needs to be translated into machine instructions

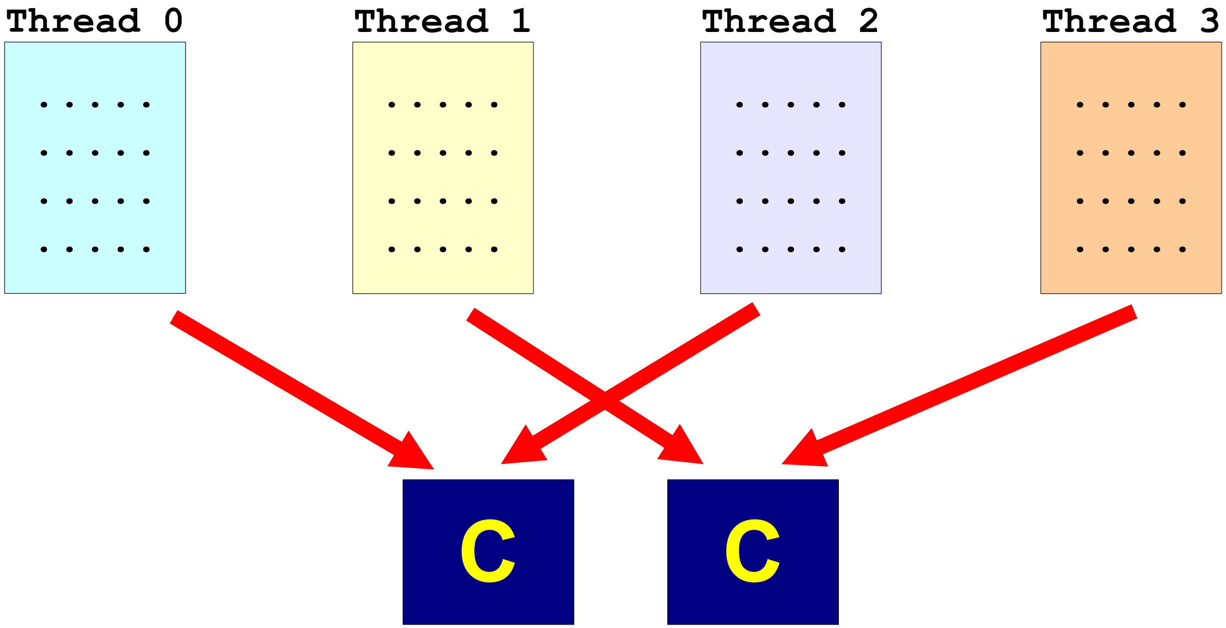
Computer Instructions/2

Machine instructions for function "average"

```
                 ..............
400b98:  jns        .+4 [ 0x400b9c ]
400b9a:  xorl       %edx,%edx
400b9c:  cmpl       $8,%edx
400b9f:  jl         .+0x42 [ 0x400be1 ]
400ba1:  leal       -8(%rdi),%edx
400ba4:  prefetcht0  0x100(%rsi)
400bab:  addsd      (%rsi),%xmm0
400baf:  addsd      8(%rsi),%xmm0
400bb4:  addsd      0x10(%rsi),%xmm0
400bb9:  addsd      0x18(%rsi),%xmm0
400bbe:  addsd      0x20(%rsi),%xmm0
400bc3:  addsd      0x28(%rsi),%xmm0
400bc8:  addsd      0x30(%rsi),%xmm0
400bcd:  addsd      0x38(%rsi),%xmm0
400bd2:  addq       $0x40,%rsi
400bd6:  addl       $8,%ecx
400bd9:  cmpl       %edx,%ecx
400bdb:  jle        .-0x37 [ 0x400ba4 ]
400bdd:  cmpl       %eax,%ecx
                 ..............
```

What is a Thread ?

- *Loosely said, a thread consists of a series of instructions*
- *A parallel program executes threads simultaneously*
- *These threads are scheduled onto the core(s)*



A Simple Serial Computation

The average of 4 numbers:

```
A = (B + C + D + E)/4
```

Computer

Step 1: `A = B + C`

Step 2: `A = A + D`

Step 3: `A = A + E`

Step 4: `A = A/4`

A Parallel Version/1

The average of 4 numbers:

```
A = (B + C + D + E)/4
```

	Computer 1	Computer 2
Step 1:	`T1 = B + C`	`T2 = D + E`
Step 2:	`A  = (T1 + T2)/4`	

A Parallel Version/2

More general, assuming we have 100 data points and 4 threads:

```
T1 = data[0] +...+ data[24]      A = (T1+T2+T3+T4)/100
T2 = data[25]+...+ data[49]
T3 = data[50]+...+ data[74]
T4 = data[75]+...+ data[99]
```

Time

How Can We Parallelize This?

1. *Inform each thread what part of the data to work on*
2. *Make sure the thread has access to the data it needs*
3. *Each thread computes the sum of its part of the data*
4. *This partial sum is accumulated into the total sum*
5. *One thread computes the f nal result by dividing the sum by the number of data points*

Question

How Do We Program This?

How To Program A Parallel Computer?

- ***Numerous <u>portable</u> parallel programming models***
- ***The ones most well-known are:***
  - ***A Single System ("Shared Memory")***
    - ✔ ***POSIX Threads (standardized, low level)***
    - ✔ ***Automatic Parallelization (compiler does it for you)***
    - ✔ ***OpenMP (de-facto standard)***

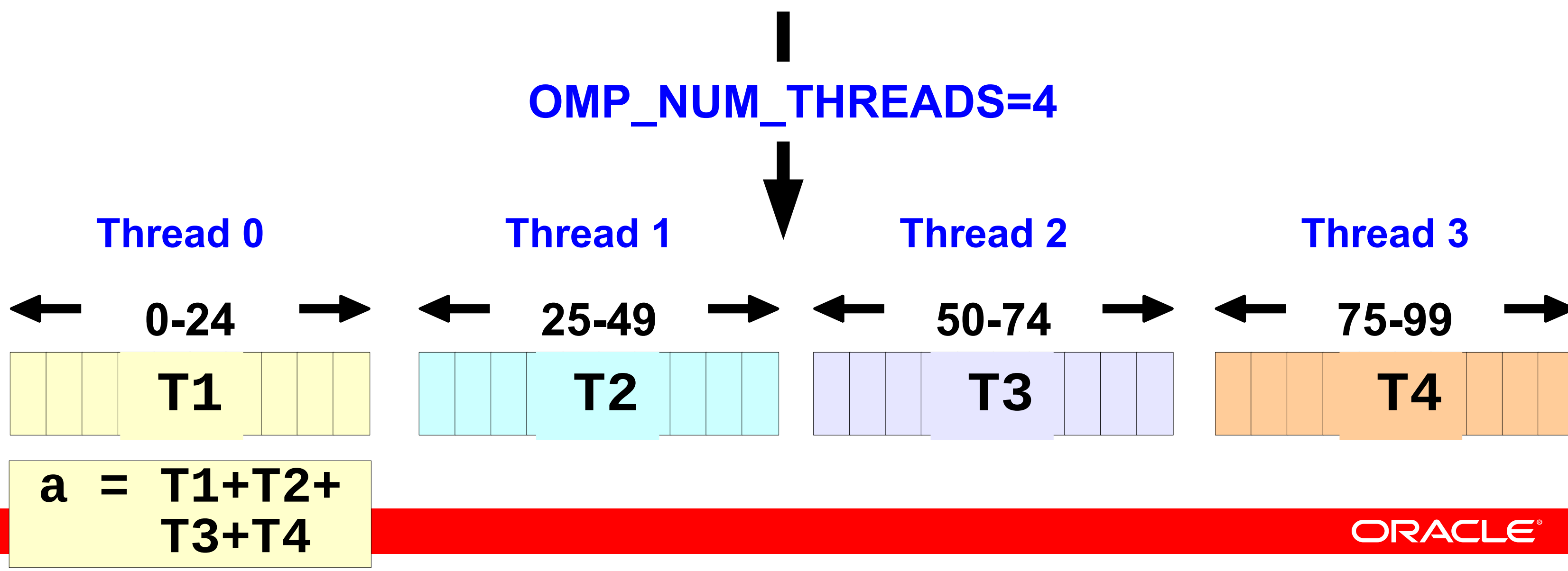
  - ***A Cluster of Systems ("Distributed Memory")***
    - ✔ ***Sockets (standardized, low level)***
    - ✔ ***PVM - Parallel Virtual Machine (obsolete)***
    - ✔ ***MPI - Message Passing Interface (de-facto std)***

Automatic Parallelization

Automatic Parallelization

- ***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<100; i++)
   a = a + data[i];
```



The Example with AutoPar

```
1   double average(int n, double data[])
2   {
3      double sum = 0.0;
4
5      for (int i=0; i<n; i++)
6          sum = sum + data[i];
7
8      return(sum/n);
9   }
```

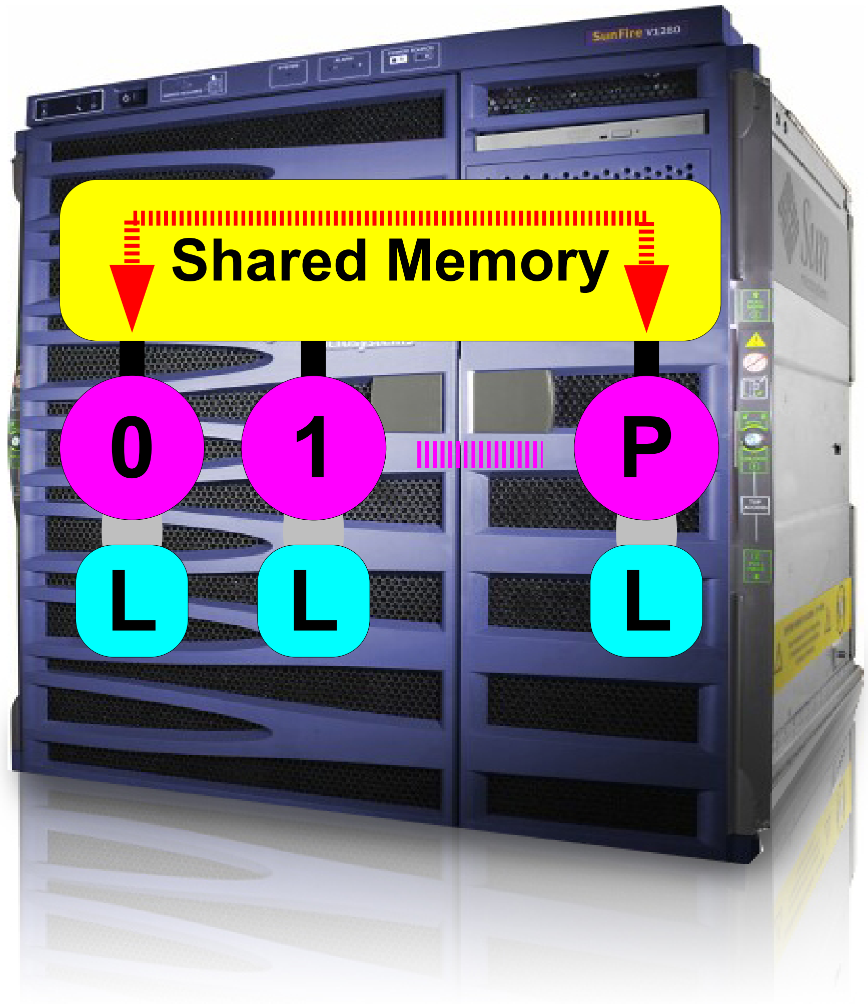
```
$ cc -c -fast -g main.c
$ cc -c -fast -g -xopenmp get_num_threads.c
$ cc -c -o average_apar.o -fast -g
     -xautopar -xreduction -xloopinfo average.c
"average.c", line 5: PARALLELIZED, reduction,
 and serial version generated
$ cc -o main_apar.exe main.o get_num_threads.o
average_apar.o -fast -g -xautopar
$
```

The Result

```
$ export OMP_NUM_THREADS=4      ← set number of threads
$ ./main_apar.exe
Please give the number of data points: 51
Number of threads used: 4       ← check # of threads
n = 51 a = 26.000000            ← numerical result
```

```
$ export OMP_NUM_THREADS=8      ← set number of threads
$ ./main_apar.exe
Please give the number of data points: 51
Number of threads used: 8       ← check # of threads
n = 51 a = 26.000000            ← numerical result
$
```

OpenMP


Shared Memory
0
1
P
L
L
L

http://www.openmp.org

http://www.compunity.org

http://www.openmp.org

Shameless Plug - "Using OpenMP"

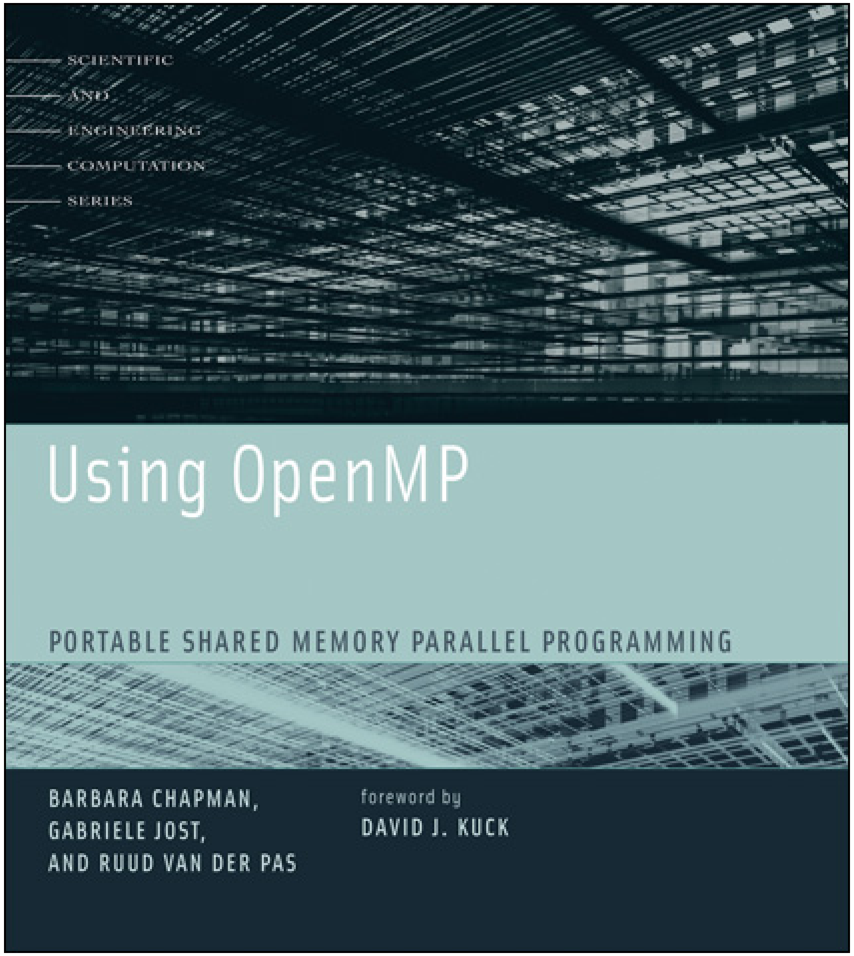
"Using OpenMP"
Portable Shared Memory Parallel Programming

Chapman, Jost, van der Pas

MIT Press, 2008

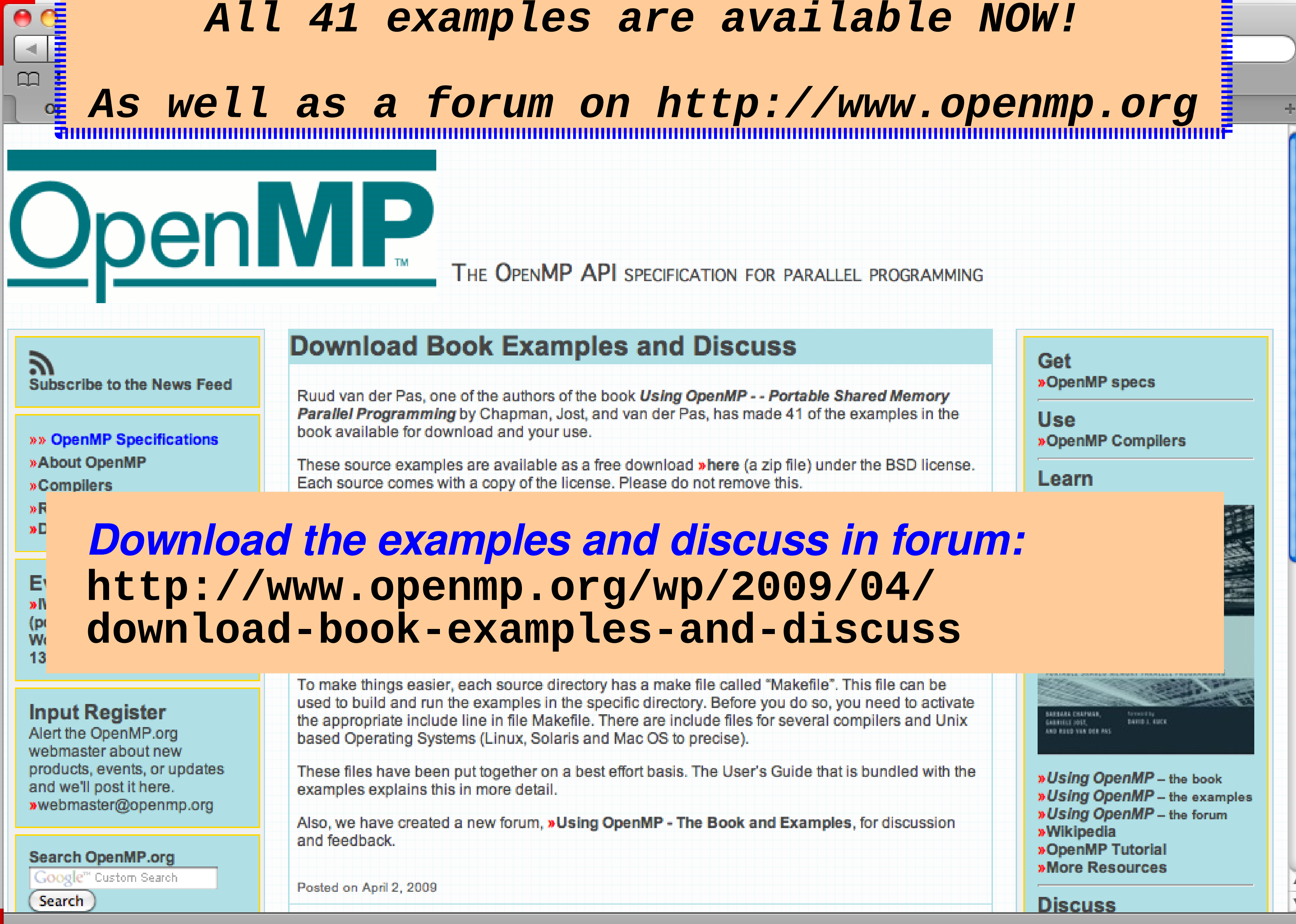
ISBN-10: 0-262-53302-2
ISBN-13: 978-0-262-53302-7

List price: 35 $US



All 41 examples are available NOW!

As well as a forum on http://www.openmp.org



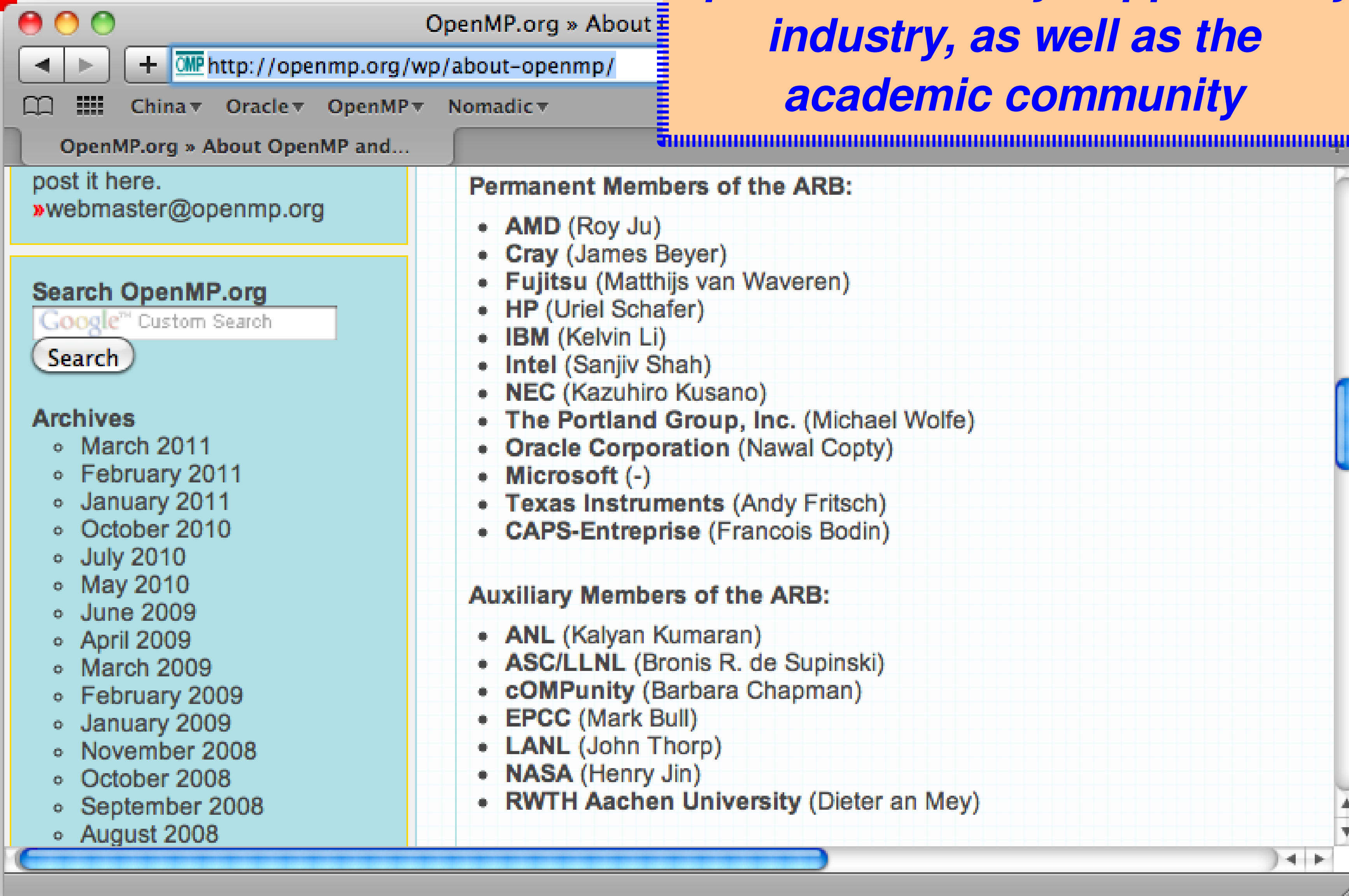
Download the examples and discuss in forum:

http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss

What is OpenMP?

- *De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran*
- *Consists of Compiler Directives, Run time routines and Environment variables*
- *Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)*
- *Version 3.0 has been released May 2008*
  - *The upcoming 3.1 release is out for public comment*

OpenMP is widely supported by industry, as well as the academic community



OpenMP.org » About

http://openmp.org/wp/about-openmp/

China ▾ Oracle ▾ OpenMP ▾ Nomadic ▾

OpenMP.org » About OpenMP and...

post it here.
»webmaster@openmp.org

Search OpenMP.org

Google™ Custom Search

Search

Archives

- March 2011
- February 2011
- January 2011
- October 2010
- July 2010
- May 2010
- June 2009
- April 2009
- March 2009
- February 2009
- January 2009
- November 2008
- October 2008
- September 2008
- August 2008

Permanent Members of the ARB:

- **AMD** (Roy Ju)
- **Cray** (James Beyer)
- **Fujitsu** (Matthijs van Waveren)
- **HP** (Uriel Schafer)
- **IBM** (Kelvin Li)
- **Intel** (Sanjiv Shah)
- **NEC** (Kazuhiro Kusano)
- **The Portland Group, Inc.** (Michael Wolfe)
- **Oracle Corporation** (Nawal Copty)
- **Microsoft** (-)
- **Texas Instruments** (Andy Fritsch)
- **CAPS-Entreprise** (Francois Bodin)

Auxiliary Members of the ARB:

- **ANL** (Kalyan Kumaran)
- **ASC/LLNL** (Bronis R. de Supinski)
- **cOMPunity** (Barbara Chapman)
- **EPCC** (Mark Bull)
- **LANL** (John Thorp)
- **NASA** (Henry Jin)
- **RWTH Aachen University** (Dieter an Mey)

When to consider OpenMP?

- ***Using an automatically parallelizing compiler:***
  - ***It can not find the parallelism***
    - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize (or not)***
  - ***The granularity is not high enough***
    - ✔ ***The compiler lacks information to parallelize at the highest possible level***
- ***Not using an automatically parallelizing compiler:***
  - ***No choice, other than doing it yourself***

Advantages of OpenMP

- *Good performance and scalability*
  - *If you do it right ....*
- *De-facto and mature standard*
- *An OpenMP program is portable*
  - *Supported by a large number of compilers*
- *Requires little programming effort*
- *Allows the program to be parallelized incrementally*

OpenMP and Multicore

OpenMP is ideally suited for multicore architectures

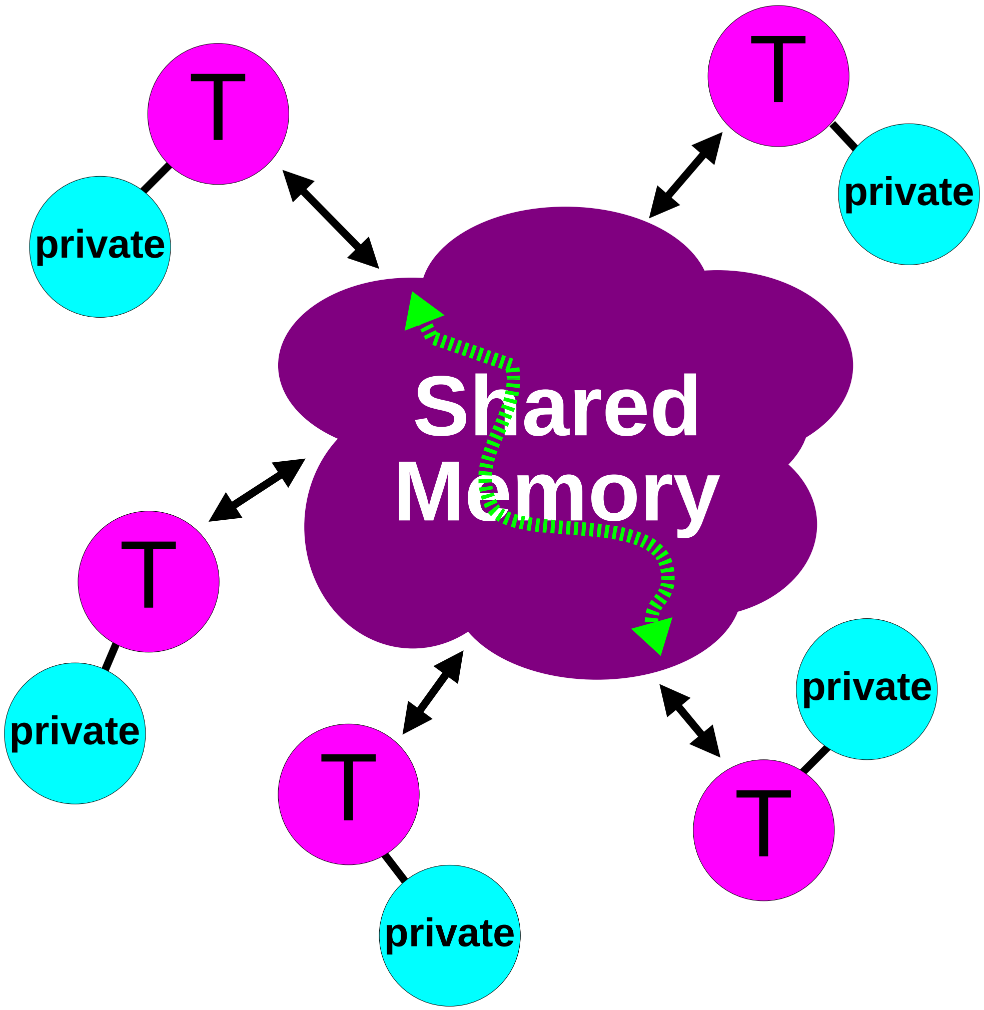
Memory and threading model map naturally

Lightweight

Mature

Widely available and used

The OpenMP Memory Model



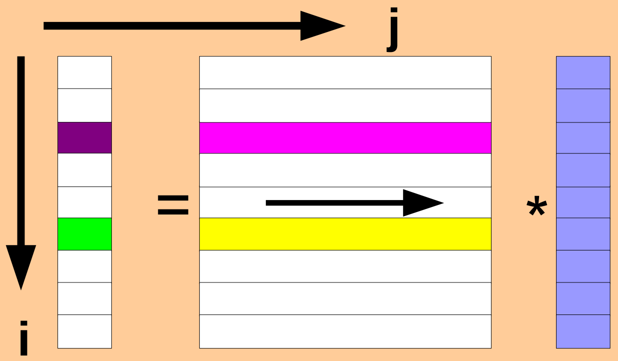
- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place,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

Data-sharing Attributes

- In an OpenMP program, data needs to be "labeled"
- Essentially there are two basic types:
  - Shared - There is only one instance of the data
    - ✔ Threads can read and write the data simultaneously unless protected through a specific construct
    - ✔ All changes made are visible to all threads
      - ◆ But not necessarily immediately, unless enforced ......
  - Private - Each thread has a copy of the data
    - ✔ No other thread can access this data
    - ✔ Changes only visible to the thread owning the 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

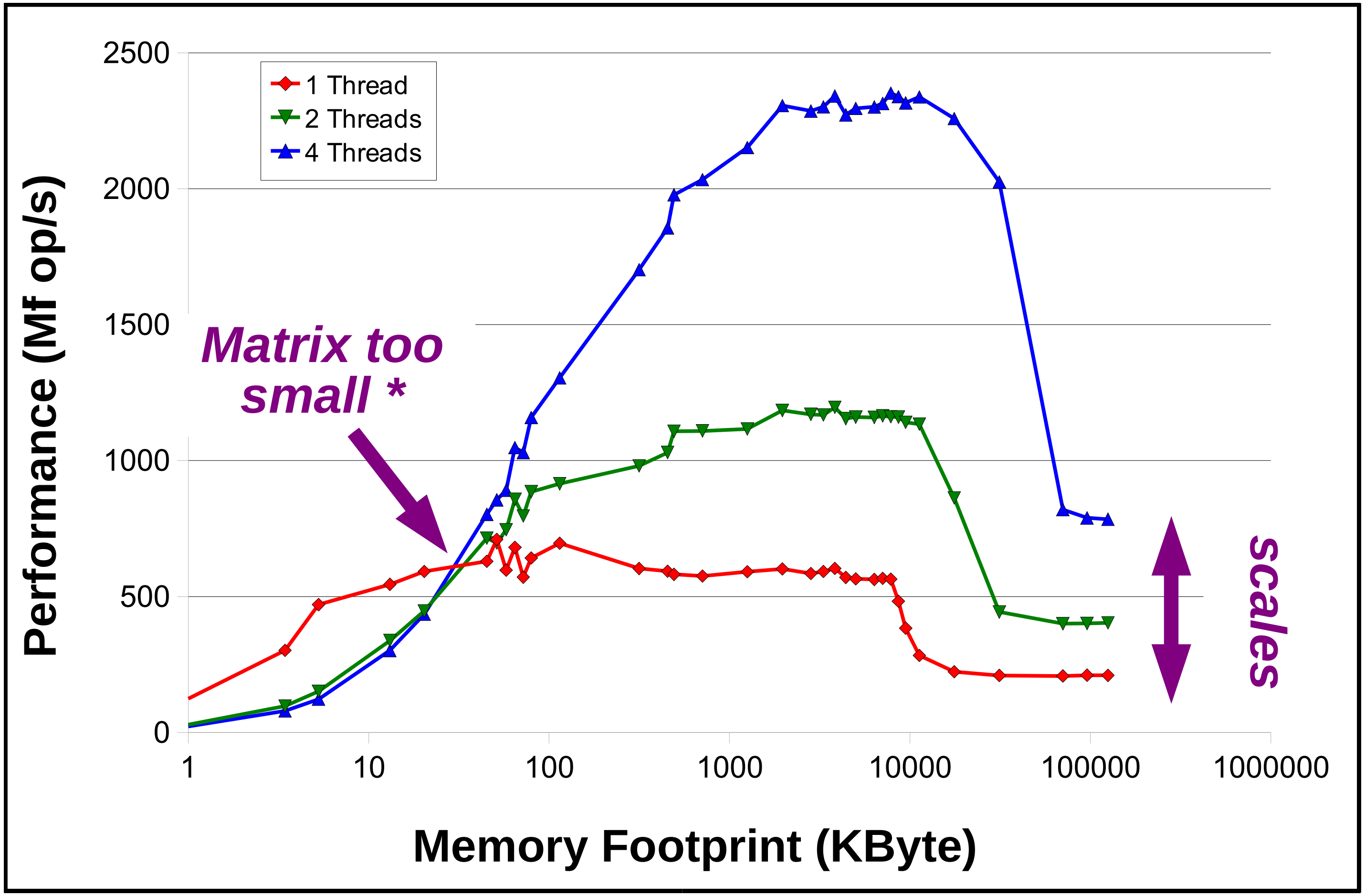
```
for (i=0,1,2,3,4)
 i = 0
  sum =    b[i=0][j]*c[j]
    a[0] = sum
 i = 1
  sum =    b[i=1][j]*c[j]
    a[1] = sum
```

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 ...

OpenMP Performance Example



) With the IF-clause in OpenMP this performance degradation can be avoided

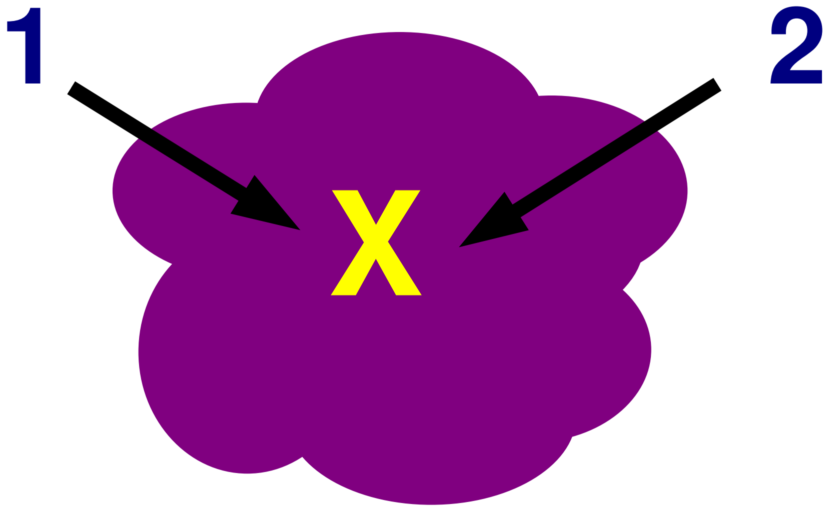
Intermezzo - Data Races

In a shared memory program, care needs to be taken when updating a shared variable

Example:

Thread 0: X = 1

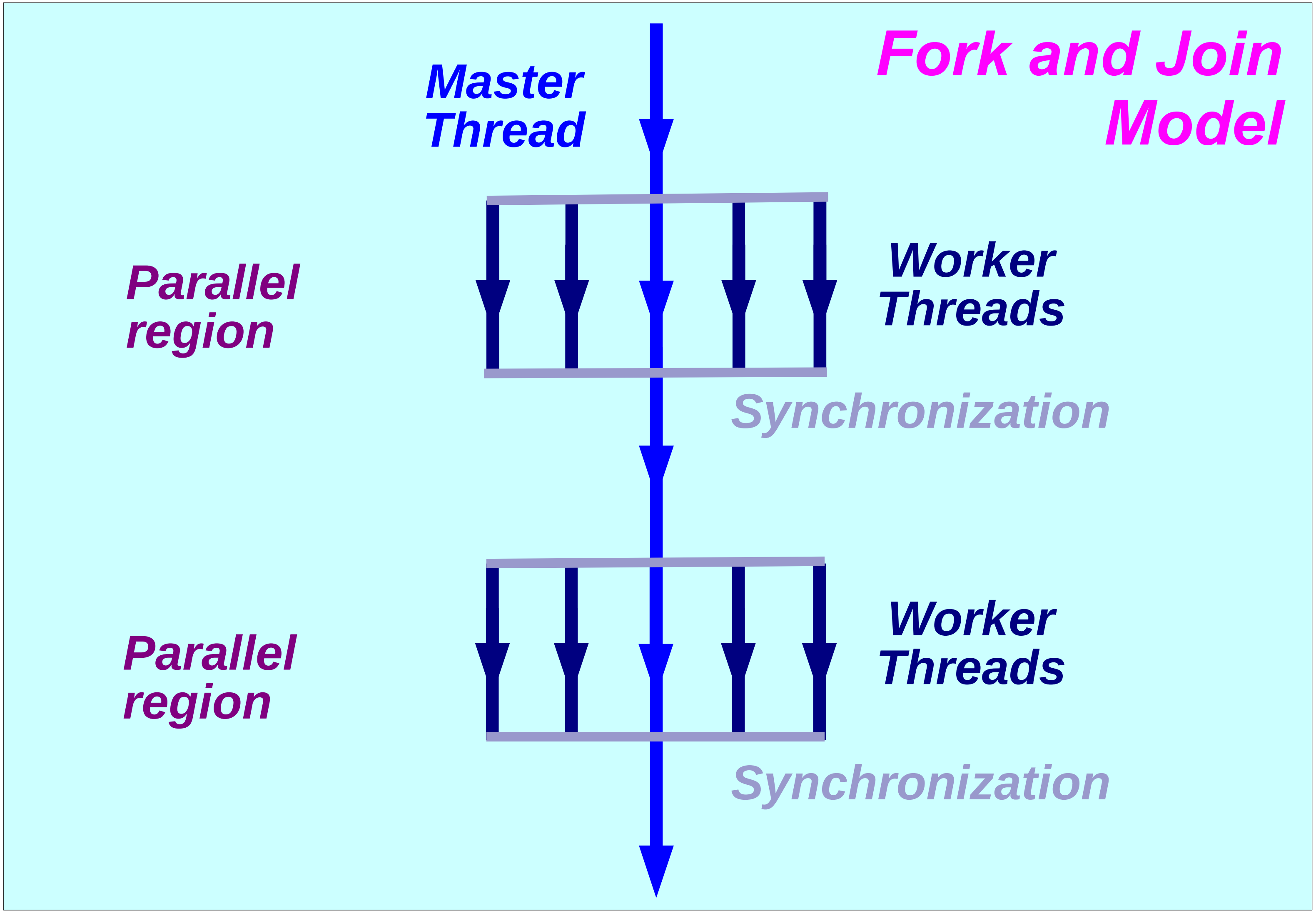
Thread 1: X = 2



The resulting value for "X" is undefined

This is an example of a "data race"

The OpenMP Execution Model



Defining Parallelism in OpenMP

- ***OpenMP Team := Master + Workers***
- ***A <u>Parallel Region</u> is a block of code executed by all threads simultaneously***
  - ☞ *The master thread always has thread ID 0*
  - ☞ *Thread adjustment (if enabled) is only done before entering a parallel region*
  - ☞ *Parallel regions can be nested, but support for this is implementation dependent*
  - ☞ *An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially*

The Parallel Region

A parallel region is a block of code executed by multiple threads simultaneously

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this code is executed in parallel"

} (implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]

    "this code is executed in parallel"

!$omp end parallel (implied barrier)
```

Parallel Region - An Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {



          printf("Hello World\n");



   return(0);
}
```

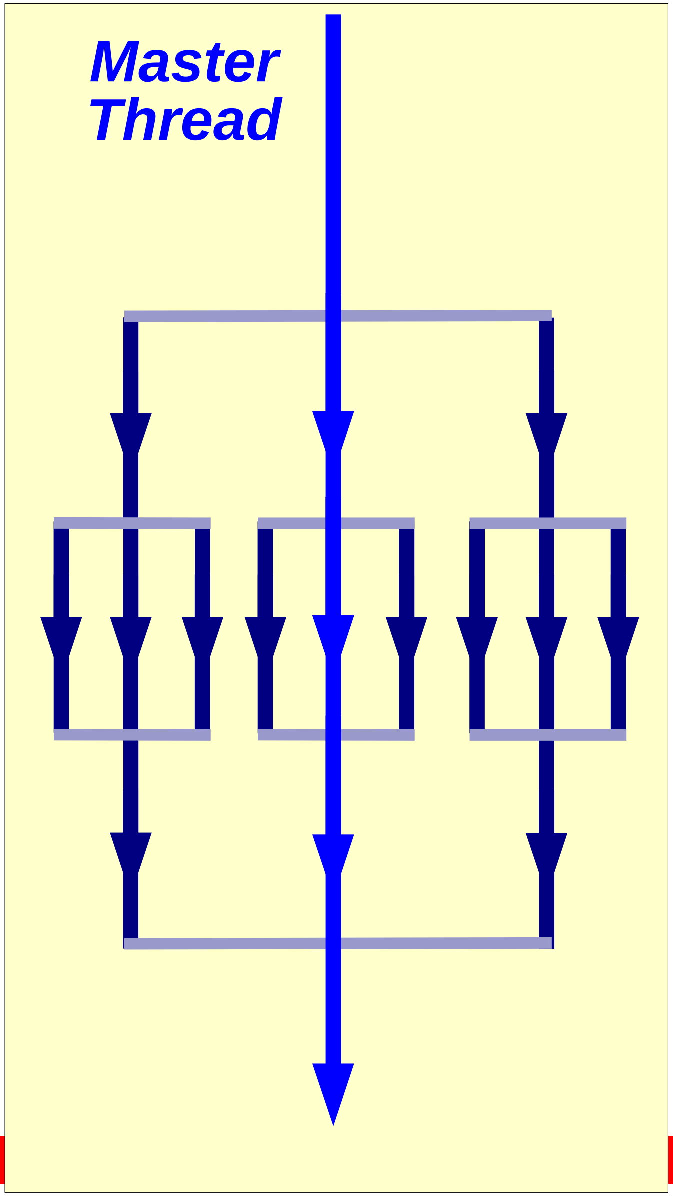
Parallel Region - An Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

Parallel Region - An Example/2

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

Nested Parallelism



3-way parallel

9-way parallel

3-way parallel

Outer parallel region

Nested parallel region

Outer parallel region

Note: nesting level can be arbitrarily deep

Remember The Example?

`T1 = data[0] +...+ data[24]`	`A = (T1+T2+T3+T4)/100`
`T2 = data[25]+...+ data[49]`	
`T3 = data[50]+...+ data[74]`	
`T4 = data[75]+...+ data[99]`	

How Can We Parallelize The Example ?

1. *Inform each thread what part of the data to work on*
2. *Make sure the thread has access to the data it needs*
3. *Each thread computes the sum of its part of the data*
4. *This partial sum is accumulated into the total sum*
5. *One thread computes the final result by dividing the sum by the number of data points*

OpenMP Strategy

1. *Have OpenMP decide for a thread what part of the data to work on.*
2. *Make array "data" shared, so all threads can read what they need*
3. *Each thread computes the "local" sum of the part of the data it has to work on.*
4. *This partial sum can be accumulated into the total "global" sum by using an OpenMP critical section*
5. *The master thread uses this value to compute the average.*

The Example - Main Program

```
void main()
{
   int n = 51;
   double a, data[n];

   printf("Please give the number of data points: ");
   scanf("%d",&n);

   printf("Number of threads used: %d\n",
           get_num_threads());

   for (int i=0; i<n; i++)
       data[i] = i+1;

   a = average(n,data);

   printf("n = %d a = %f\n",n,a);
}
```

The Example with OpenMP*

```
double average(int n, double data[])
{
     double sum = 0.0;

 #pragma omp parallel default(none) \
         shared(n,sum,data)
     {
        double Lsum = 0.0;
        #pragma omp for
        for (int i=0; i<n; i++)
            Lsum = Lsum + data[i];

        printf("\tThread %d: has computed its
               local sum: %.2f\n",
               omp_get_thread_num(), Lsum);

        #pragma omp critical
        {sum = sum + Lsum;}
     } // End of parallel region

     return(sum/n);
 }
```

****) This example can be done more easily with the reduction clause***

How Does This Work?

Assumptions: n=10 and we use 2 threads

Thread 0

```
Lsum = 0.0
for (i=0; i<5; i++)
 Lsum = Lsum+data[i]
sum = sum + Lsum
```

Thread 1

```
Lsum = 0.0
for (i=5; i<10; i++)
 Lsum = Lsum+data[i]

sum = sum + Lsum
```

Wait !

Build The Example

```
$ cc -c -fast -g main.c
$ cc -c -fast -g -xopenmp get_num_threads.c
$ cc -c -fast -g -xopenmp -xvpara -xloopinfo
average_omp.c
"average_omp.c", line 18: PARALLELIZED, user
pragma used
$ cc -o main_omp.exe main.o get_num_threads.o
average_omp.o -fast -g -xopenmp
$
```

Run The Example (2 threads)

```
$ export OMP_NUM_THREADS=2      ← set number of threads
$ ./main_omp.exe
Please give the number of data points: 51
Number of threads used: 2       ← check # of threads
     Thread 0: has computed its local sum: 351.00
     Thread 1: has computed its local sum: 975.00
n = 51 a = 26.000000            ← numerical result
$
```

Run The Example (4 threads)

```
$ export OMP_NUM_THREADS=4          ← set number of threads
$ ./main_omp.exe
Please give the number of data points: 51
Number of threads used: 4           ← check # of threads
     Thread 3: has computed its local sum: 546.00
     Thread 0: has computed its local sum: 91.00
     Thread 1: has computed its local sum: 260.00
     Thread 2: has computed its local sum: 429.00
n = 51 a = 26.000000                ← numerical result
$
```

MPI The Message Passing Interface

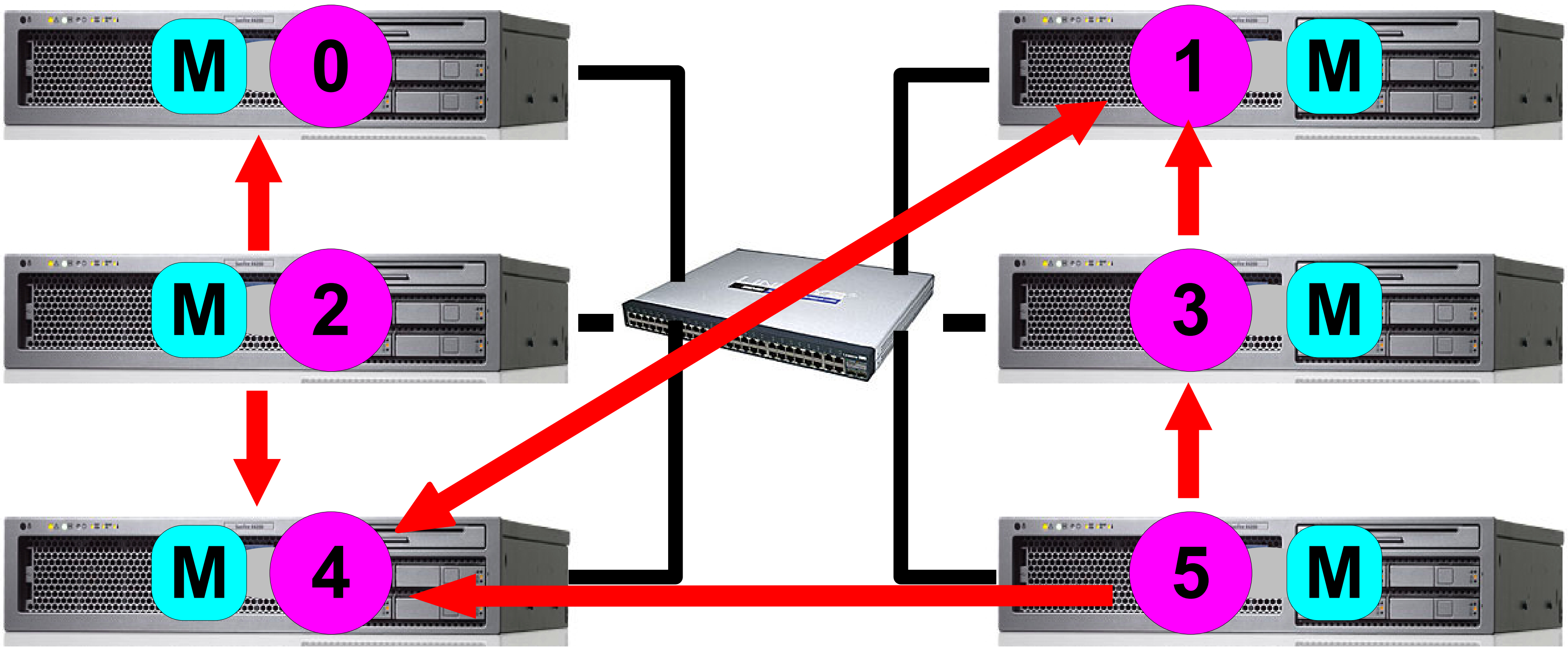
What is MPI?

- ***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 specification: 1994***
  - ***Major enhancements in MPI-2 (1997)***
    - ✔ ***Remote memory management, Parallel I/O and Dynamic process management***
  - ***MPI 2.2 was released September 2009***

More about MPI

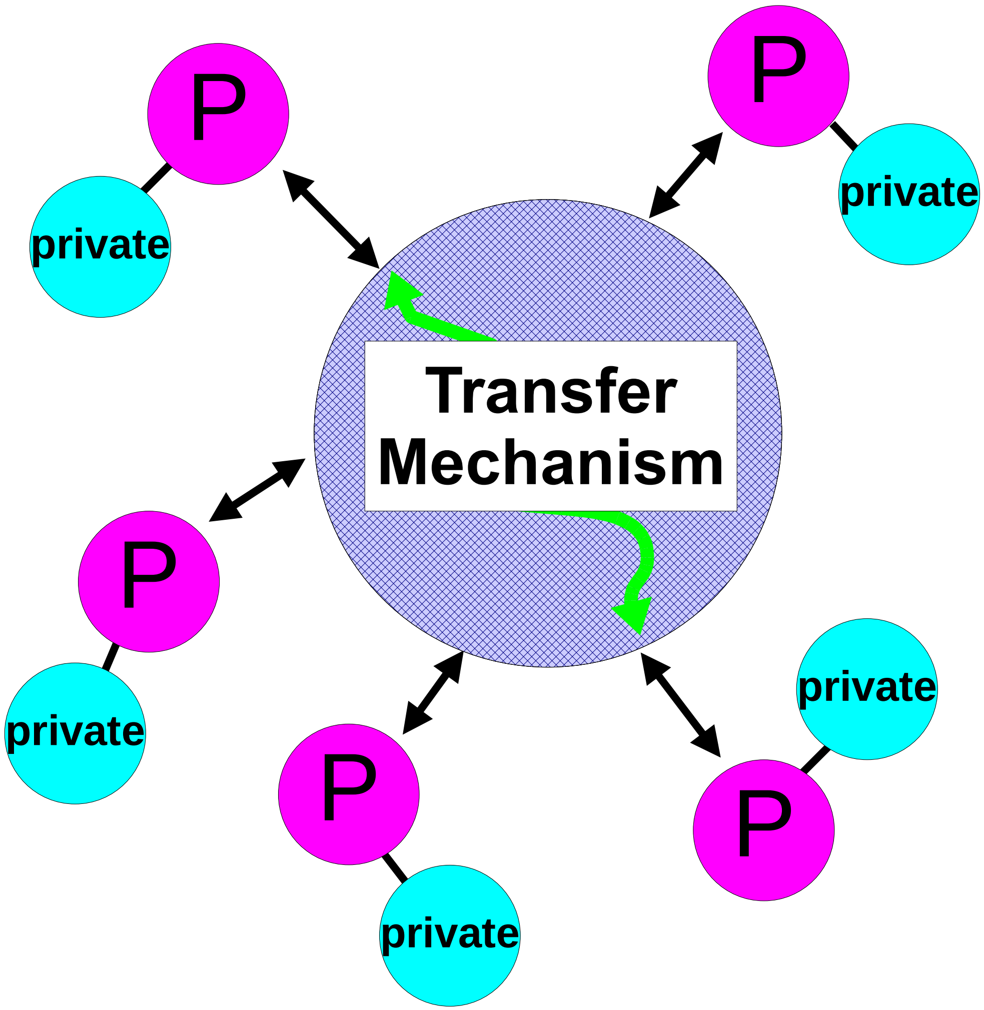
- MPI has its own data types (e.g. MPI_INT)
  - User defined data types are supported as well
- MPI supports C, C++ and Fortran
  - Include file <mpi.h> in C/C++ and “mpif.h” in Fortran
- An MPI environment typically consists of at least:
  - A library implementing the API
  - A compiler and linker that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

The MPI Programming Model



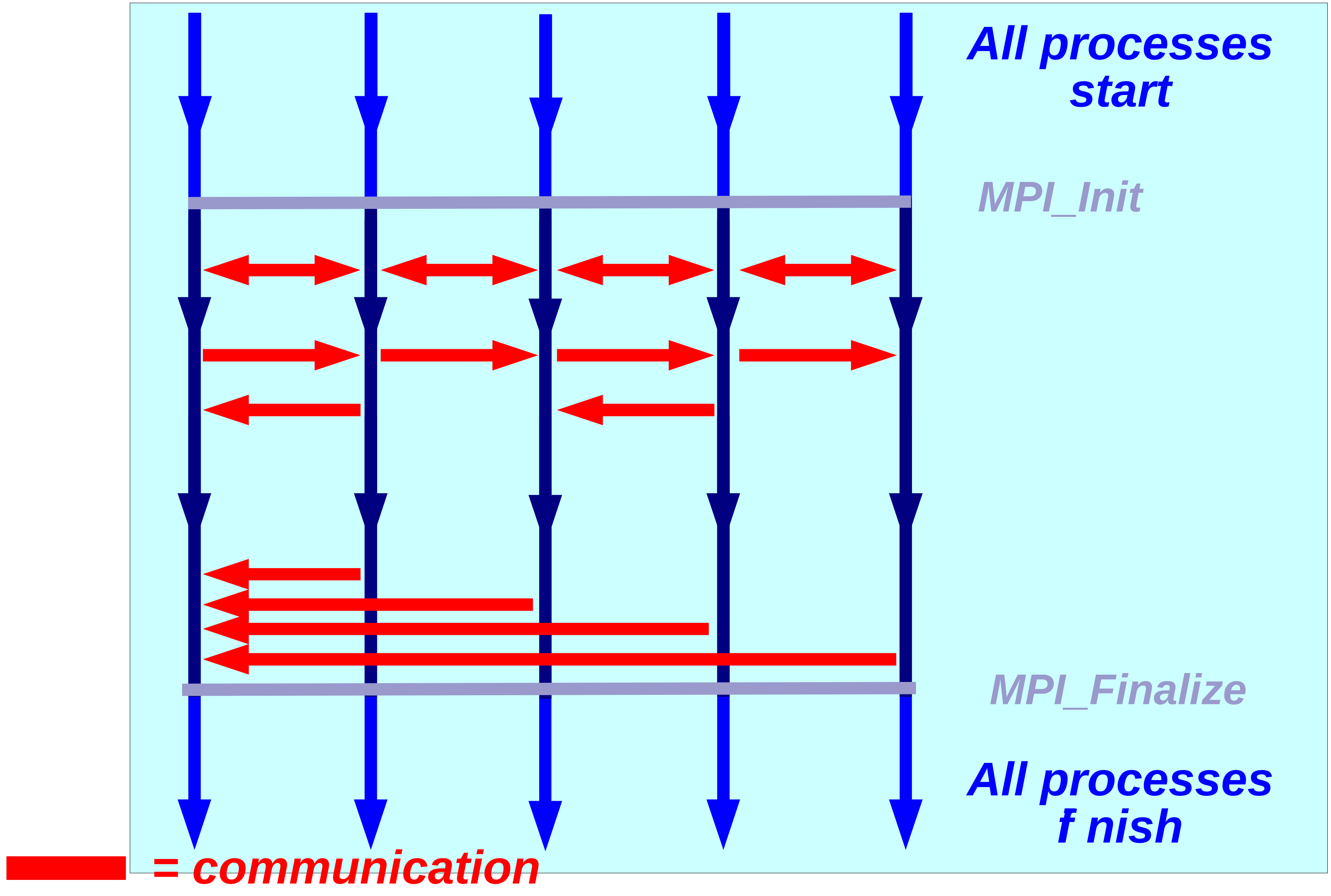
A Cluster Of Systems

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*

The MPI Execution Model



The Six Basic MPI Functions/1

1. Initialize MPI environment (mandatory)

```
int MPI_Init(int *argc, char ***argv)
```

2. Clean up all MPI states (mandatory)

```
int MPI_Finalize()
```

Example - “Hello World” *

```
#include <stdio.h>
#include <stdlib.h>

#include <mpi.h>

int main (int argc, char **argv)
{

  MPI_Init(&argc, &argv);

  printf("Hello Parallel World\n");

  MPI_Finalize();

}
```

```
$ mpicc hello-world.c
$ mpirun -np 4 ./a.out
Hello Parallel World
Hello Parallel World
Hello Parallel World
Hello Parallel World
$
```

****) Handling of I/O is implementation dependent (outside using MPI I/O)***

The Six Basic MPI Functions/2

3. Returns the number of MPI processes in "size"

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

4. Returns the MPI process ID ("the rank") in "rank"

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Example - “Hello World”

```
#include <stdio.h>
#include <stdlib.

#include <mpi.h>

int main (int arg
{

  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();

}
```

```
$ mpicc hello-world.c
$ 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
$
```

The Six Basic MPI Functions/3

5. Send a message to “dest”

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

6. Receive a message from “source”

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm,
             MPI_Status *status)
```

How Can We Parallelize The Example ?

1. ***Inform each thread what part of the data to work on***
2. ***Make sure the thread has access to the data it needs***
3. ***Each thread computes the sum of its part of the data***
4. ***This partial sum is accumulated into the total sum***
5. ***One thread computes the final result by dividing the sum by the number of data points***

MPI Strategy

1. *Master process reads input and defines what part of the data each process has to work on.*
2. *Master process sends the size of this chunk and the relevant part of the data to each process.*
3. *Each process receives this information.*
4. *Each process computes the sum of its part of the data and stores it in a local variable.*
5. *Master process collects these partial sums and accumulates it into the global sum.*
6. *Master process uses this value to compute the average.*

MPI Example - Get Started

```
#include <mpi.h>

int main (int argc, char **argv)
{

  int master = 0, msg_tag1 = 1117, msg_tag2 = 2009;

  if (ier = MPI_Init(&argc, &argv)) != 0 ) {
      .....
  }

  if ( (ier = MPI_Comm_size(MPI_COMM_WORLD,&nproc)) !=0 )

  {
      .....
  }

  if ( (ier = MPI_Comm_rank(MPI_COMM_WORLD,&me)) !=0 )
  {
      .....
  }
```

MPI - Set Up Phase

```
if ( me == 0 ) {
    printf("Please give the number of data points: ");
    fflush(stdout); scanf("%d",&n);

    printf("There are %d MPI processes\n",nproc);
    printf("Number of data points: %d\n",n);

     if ( (data = (double *) malloc(n*sizeof(double)))
                                              == NULL )
     {
         ...........
     } else {
        for (int i=0; i<n; i++) data[i] = i+1;
     }

     int irem   = n%nproc;
     int nchunk = (n-irem)/nproc;
     int istart = 0;
     int iend   = 0;
           ................ (continued on next slide)
```

MPI - Define and Assign Work

```
      for (int p=1; p<nproc; p++)
      {
          if (p < irem) {
             istart=(nchunk+1)*p; iend=istart+nchunk;
           } else {
             istart=nchunk*p+irem; iend=istart+nchunk-1;
           }
           vlen = iend-istart+1;
           if ( (ier = MPI_Send(&vlen,1, MPI_INT, p,
                       msg_tag1, MPI_COMM_WORLD)) != 0 )
{
              .....
           }
           if ( (ier = MPI_Send(&data[istart], vlen,
                       MPI_DOUBLE_PRECISION, p, msg_tag2,
                       MPI_COMM_WORLD)) != 0 ) {
              .....
           }
       } // End of for loop, still in “if” branch
       vlen = ( irem > 0 ) ? nchunk+1 : nchunk;

  } else {
```

MPI - Process Receives Info

```
  if ( (ier = MPI_Recv(&vlen, 1, MPI_INT, master,
              msg_tag1, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE)) != 0 ) {
              .....
  }

  if ( (data=(double *) malloc(vlen*sizeof(double)))
                                        == NULL ) {
              .....
  }
  if ( (ier = MPI_Recv(data, vlen,
              MPI_DOUBLE_PRECISION, master, msg_tag2,
              MPI_COMM_WORLD,
              MPI_STATUS_IGNORE)) != 0 ) {
              .....
  }
} // End of "if (me == 0) then .... else ...."
```

MPI - Computation And Final Result

```
Lsum = 0.0;
for (int i=0; i<vlen; i++)
{
    Lsum = Lsum + data[i];
}
printf("\tProcess %d: has computed its local sum:
                             %.2f\n",me,Lsum);

if ( (ier = MPI_Reduce(&Lsum,&sum,1,
            MPI_DOUBLE_PRECISION,
            MPI_SUM,master,MPI_COMM_WORLD)) !=0 ) {
           .....
}

if ( me == 0 ) {
   average = sum / n;
   printf("n = %d a = %.2f\n",n,average);
}

free(data);

if ( (ier = MPI_Finalize()) != 0 ) {.....}
```

Build And Run The Example

```
$ mpicc -c -fast -g average_mpi.c
$ mpicc -o main_mpi.exe average_mpi.o -fast -g

$ mpirun -np 2 ./main_mpi.exe        ← set number of procs
Please give the number of data points: 51
There are 2 MPI processes            ← check # of procs
Number of data points: 51
    Process 0: has computed its local sum: 351.00
    Process 1: has computed its local sum: 975.00
n = 51 a = 26.00                     ← numerical result
$
```

Run The Example (4 processes)

```
$ mpirun -np 4 ./main_mpi.exe
Please give the number of data points: 51
There are 4 MPI processes
Number of data points: 51
    Process 0: has computed its local sum: 91.00
    Process 2: has computed its local sum: 429.00
    Process 3: has computed its local sum: 546.00
    Process 1: has computed its local sum: 260.00
n = 51 a = 26.00
$
```

set number of procs

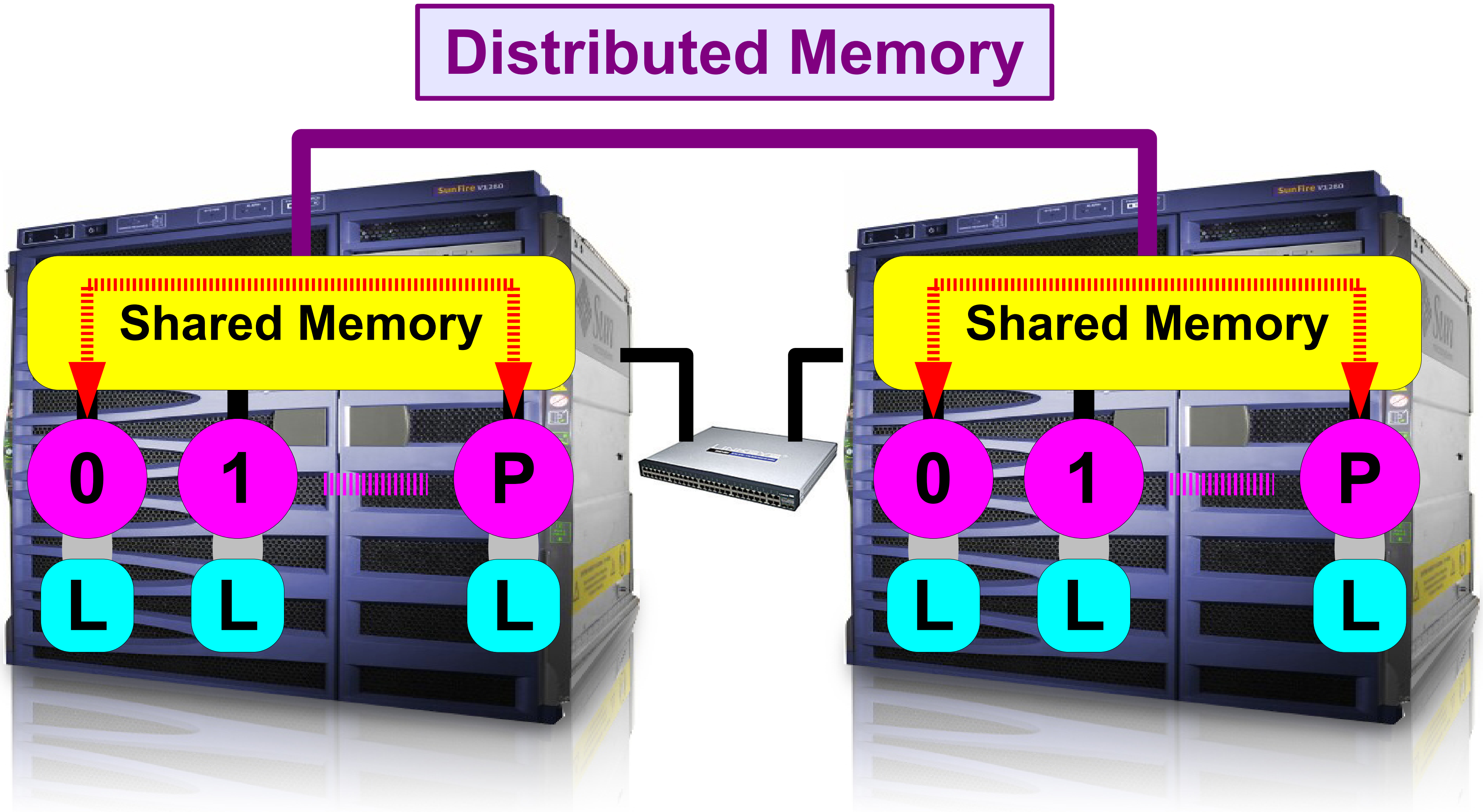
check # of procs

numerical result

Intermezzo
MPI or OpenMP ?

An Answer
The Hybrid Parallel Programming Model

The Hybrid Programming Model



MPI Example - Compute Part

```
Lsum = 0.0;
for (int i=0; i<vlen; i++)
{
    Lsum = Lsum + data[i];
}
```

Wait a minute, this is exactly the same computation as we started with, only on a subset of the data

But that means we could use OpenMP to parallelize this computation within each MPI process ....

The Hybrid Example *

```
Lsum = 0.0;
#pragma omp parallel default(none) \
            shared(me,vlen,data,Lsum)
{
   #pragma omp single
   {printf("\tMPI process %d uses %d OpenMP threads\n",
            me,omp_get_num_threads());}

   double ThreadSum = 0.0;
   #pragma omp for
   for (int i=0; i<vlen; i++)
       ThreadSum = ThreadSum + data[i];

   #pragma omp critical
   {Lsum = Lsum + ThreadSum;}

} // End of parallel region
```

****) This example can be done more easily with the reduction clause***

ORACLE

Using OpenMP

Using OpenMP

- ***We have just seen a glimpse of OpenMP***
- ***To be practically useful, much more functionality is needed***
- ***Covered in this section:***
  - ***Many of the language constructs***
  - ***Features that may be useful or needed when running an OpenMP application***
- ***Note that the tasking concept is covered in a separate section***

Components of OpenMP

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Wallclock timer
- Locking

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

Directive format

- ❑ *C: directives are case sensitive*
  - *Syntax:* **#pragma omp directive [clause [clause] ...]**
- ❑ *Continuation: use \ in pragma*
- ❑ *Conditional compilation:* **_OPENMP** *macro is set*
- ❑ *Fortran: directives are case insensitive*
  - *Syntax:* **sentinel directive [clause [[,] clause]...]**
  - *The sentinel is one of the following:*
    - ✓ ***!$OMP*** *or* ***C$OMP*** *or* ****$OMP*** *(fixed format)*
    - ✓ ***!$OMP*** *(free format)*
- ❑ *Continuation: follows the language syntax*
- ❑ *Conditional compilation:* **!$** *or* **C$ -> 2 spaces**

OpenMP clauses

- *Many OpenMP directives support clauses*
  - *These clauses are used to provide additional information with the directive*
- *For example,* private(a) *is a clause to the "for" directive:*
  - #pragma omp for private(a)
- *The specific clause(s) that can be used, depend on the directive*

The if clause

if (scalar expression)

- ✔ *Only execute in parallel if expression evaluates to true*
- ✔ *Otherwise, execute serially*

```
#pragma omp parallel if (n > some_threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

Private and shared clauses

private (list)

- ✔ ***No storage association with original object***
- ✔ ***All references are to the local object***
- ✔ ***Values are undefined on entry and exit***

shared (list)

- ✔ ***Data is accessible by all threads in the team***
- ✔ ***All threads access the same address space***

The default clause

default (none | shared) — C/C++

default (none | shared | private | threadprivate) — Fortran

none

- *No implicit defaults; have to scope all variables explicitly*

shared

- *All variables are shared*
- *The default in absence of an explicit "default" clause*

private

- *All variables are private to the thread*
- *Includes common block data, unless THREADPRIVATE*

f rstprivate

- *All variables are private to the thread; pre-initialized*

Barrier/1

Suppose we run each of these two loops in parallel over i:

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

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

This may give us a wrong answer (one day)

Why ?

Barrier/2

We need to have <u>updated all of a[ ]</u> first, before using a[ ] *

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

wait !

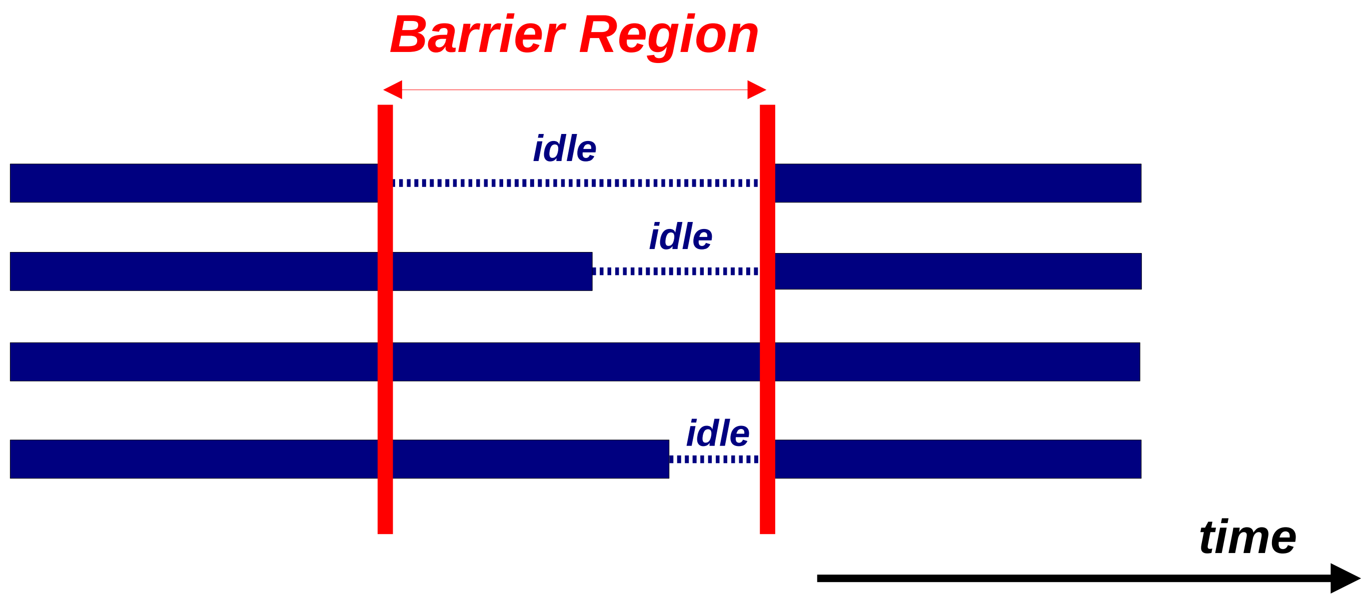
barrier

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

All threads wait at the barrier point and only continue when all threads have reached the barrier point

****) If there is the <u>guarantee</u> that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case***

Barrier/3



Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

When to use barriers ?

- ***If data is updated asynchronously and data integrity is at risk***
- ***Examples:***
  - ***Between parts in the code that read and write the same section of memory***
  - ***After one timestep/iteration in a solver***
- ***Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors***
- ***Therefore, use them with care***

The nowait clause

- *To minimize synchronization, some directives support the optional* ***nowait*** *clause*
  - *If present, threads do not synchronize/wait at the end of that particular construct*
- *In C, it is one of the clauses on the pragma*
- *In Fortran, it is appended at the closing part of the construct*

```c
#pragma omp for nowait
{
        :
}
```

```fortran
!$omp do
      :
      :
!$omp end do nowait
```

The Worksharing Constructs

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- *The work is distributed over the threads*
- *Must be enclosed in a parallel region*
- *Must be encountered by all threads in the team, or none at all*
- *No implied barrier on entry; implied barrier on exit (unless nowait is specifed)*
- *A work-sharing construct does not launch any new threads*

The Workshare construct

Fortran has a fourth worksharing construct:

```
!$OMP WORKSHARE

    <array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

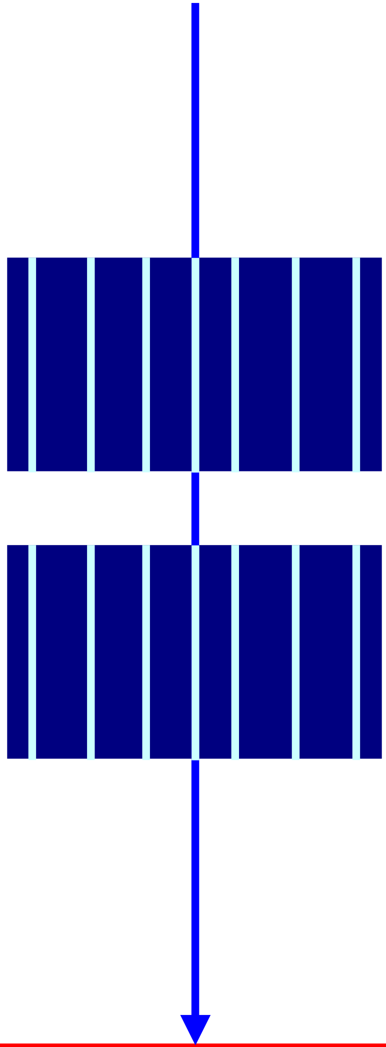
Example:

```
!$OMP WORKSHARE
    A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

The omp for directive - Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



A more elaborate example

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];

#pragma omp for nowait

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

            ....
#pragma omp barrier

    scale = sum(a,0,n) + sum(z,0,n) + f;
            ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

Statement is executed by all threads

parallel region

ORACLE

The schedule clause/1

schedule (static | dynamic | guided | auto [, chunk])
schedule (runtime)

static [, chunk]

- ✔ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✔ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*
  - *Details are implementation defined*
- ✔ *Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region*

The schedule clause/2

Example static schedule

Loop of length 16, 4 threads:

Thread	0	1	2	3
*no chunk**	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

**) The precise distribution is implementation def ned*

The schedule clause/3

dynamic [, chunk]

- ✔ *Fixed portions of work; size is controlled by the value of chunk*
- ✔ *When a thread finishes, it starts on the next portion of work*

guided [, chunk]

- ✔ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

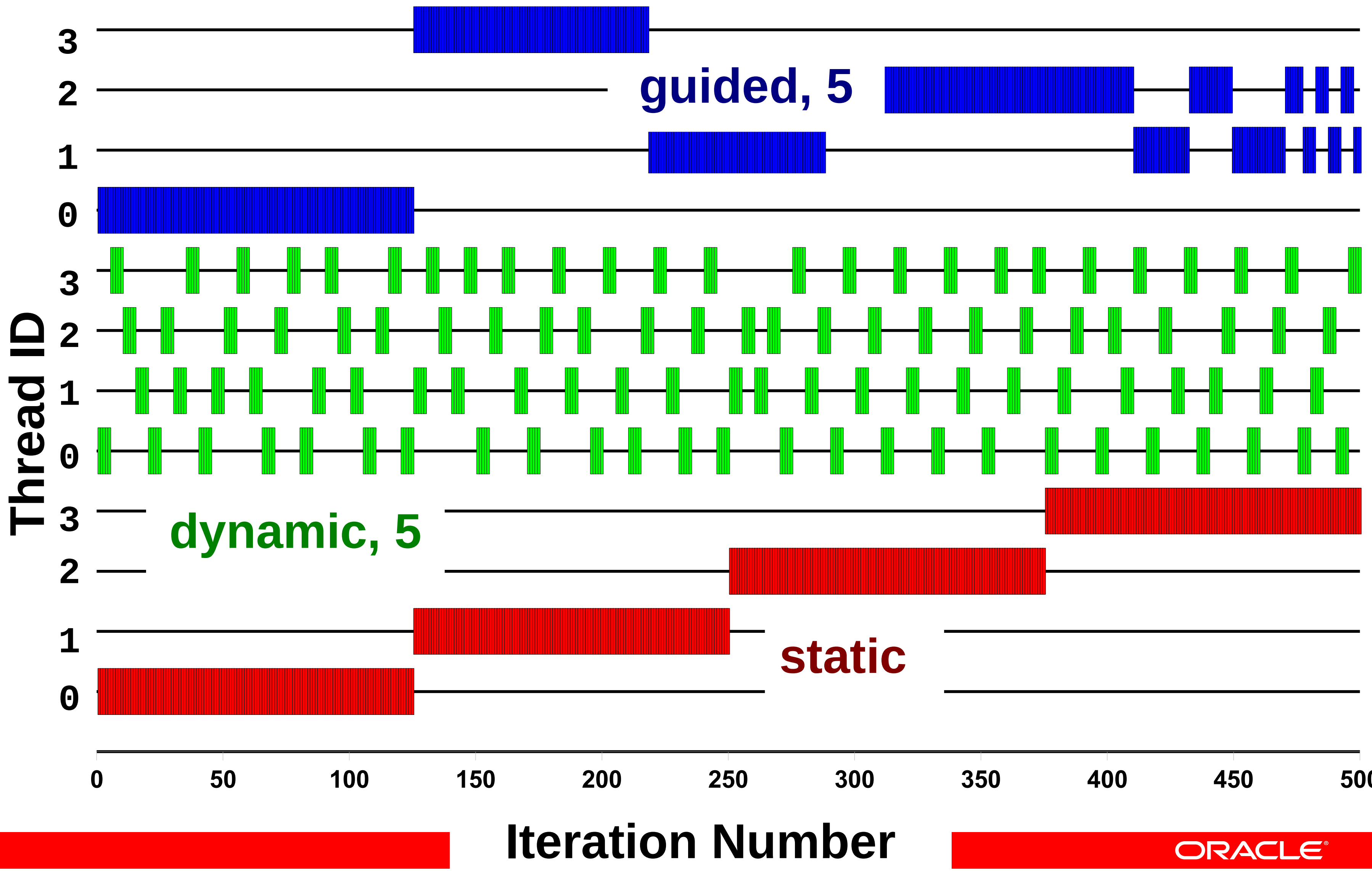
auto

- ✔ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

runtime

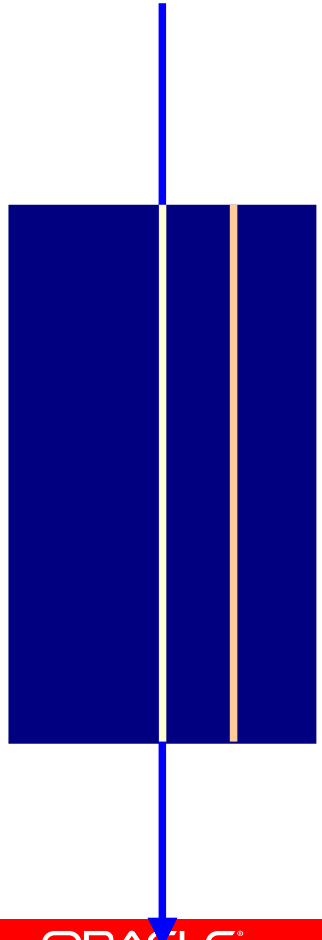
- ✔ *Iteration scheduling scheme is set at runtime through environment variable* ***OMP_SCHEDULE***

Experiment - 500 iterations, 4 threads



The Sections Directive - Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



Overlap I/O and Processing/1

Time

Input Thread	Processing Thread(s)	Output Thread
0		
1	0	
2	1	0
3	2	1
4	3	2
5	4	3
	5	4
		5

Overlap I/O and Processing/2

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) read_input(i);
         (void) signal_read(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_read(i);
         (void) process_data(i);
         (void) signal_processed(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_processed(i);
         (void) write_output(i);
     }
   }
} /*-- End of parallel sections --*/
```

Input Thread

Processing Thread(s)

Output Thread

Single processor region/1

This construct is ideally suited for I/O or initializations

Original Code

```
.....
"read a[0..N-1]";
.....
```

"declare A to be be shared"

```
#pragma omp parallel
{
    .....
    "read a[0..N-1]";
    .....
}
```

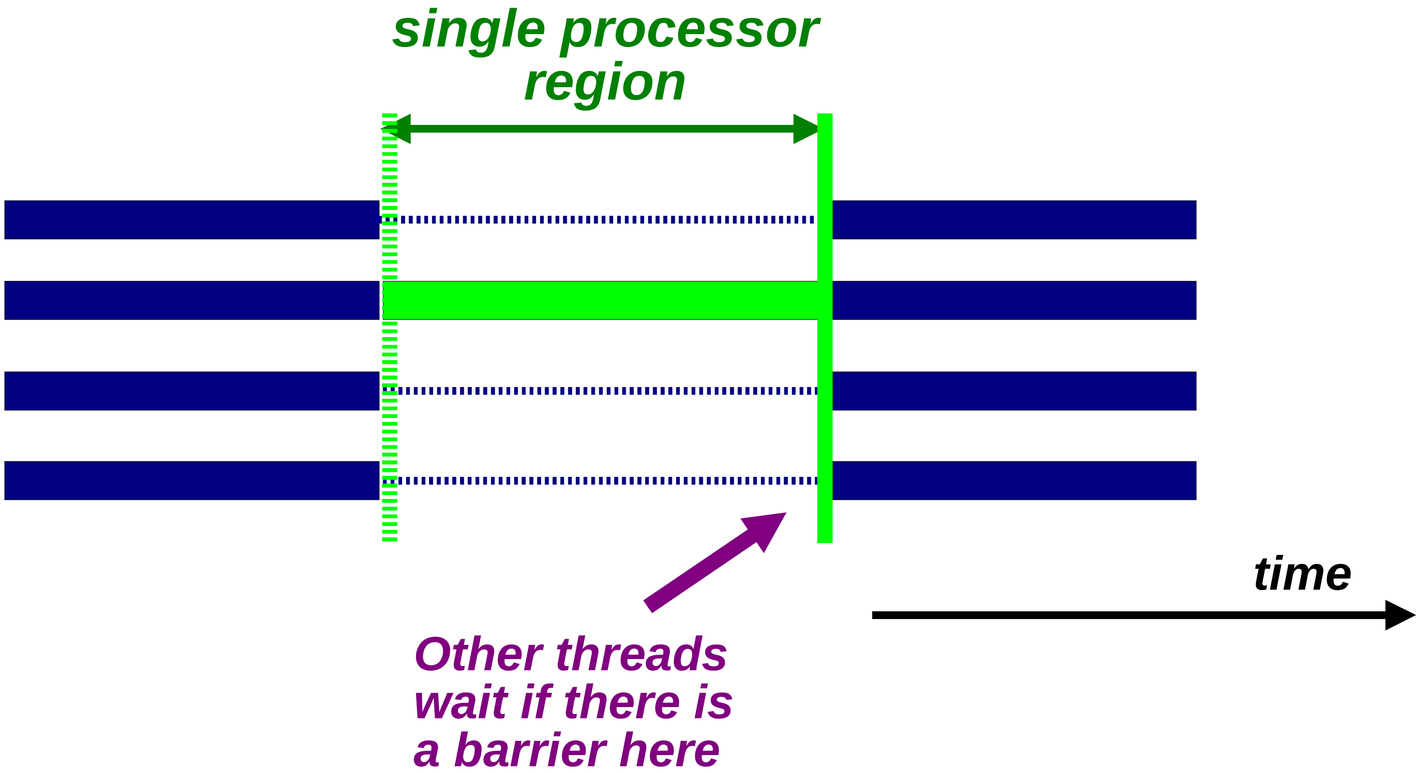
one volunteer requested

thanks, we're done

May have to insert a barrier somewhere here

Parallel Version

Single processor region/2



The Single Directive

Only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
       <code-block>
}
```

```
!$omp single [private][firstprivate]
   <code-block>
!$omp end single [copyprivate][nowait]
```

Combined work-sharing constructs

```
#pragma omp parallel
#pragma omp for
    for (...)
```

```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
!$omp parallel
!$omp do
        ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
        ...
!$omp end parallel do
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
        ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
        ...
!$omp end parallel workshare
```

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel
sections
{ ... }
```

Single PARALLEL sections

```
!$omp parallel
!$omp sections
        ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
        ...
!$omp end parallel sections
```

Additional Directives/1

```
#pragma omp master
{<code-block>}
```

```
!$omp master
          <code-block>
!$omp end master
```

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
          <code-block>
!$omp end critical [(name)]
```

```
#pragma omp atomic
```

```
!$omp atomic
```

Critical Region/1

If sum is a shared variable, this loop can not run in parallel

```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

We can use a critical region for this:

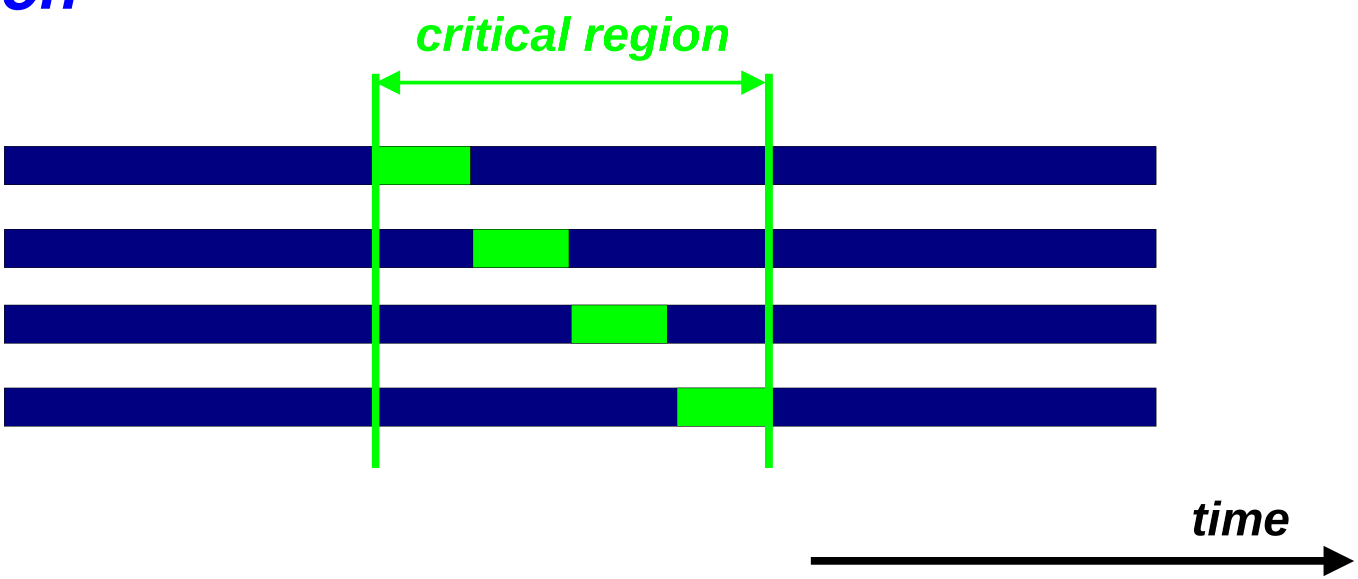
```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

one at a time can proceed

next in line, please

Critical Region/2

- ***Useful to avoid a race condition, or to perform I/O (but that still has random order)***
- ***Be aware that there is a cost associated with a critical region***



Critical and Atomic constructs

Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

Atomic: only the loads and store are atomic ....

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
      a[indx[i]] += b[i];
```

Additional Directives/2

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
        <code-block>
!$omp end ordered
```

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

OpenMP Runtime Routines

OpenMP Runtime Functions/1

Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Number of threads in team
omp_get_max_threads	Max num of threads for parallel region
omp_get_thread_num	Get thread ID
omp_get_num_procs	Maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment (but implementation is free to ignore this)
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism (but implementation is free to ignore this)
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

C/C++ : Need to include file <omp.h>
Fortran : Add “use omp_lib” or include file “omp_lib.h”

OpenMP Runtime Functions/2

Name	Functionality
omp_set_schedule	Set schedule (if “runtime” is used)
omp_get_schedule	Returns the schedule in use
omp_get_thread_limit	Max number of threads for program
omp_set_max_active_levels	Set number of active parallel regions
omp_get_max_active_levels	Number of active parallel regions
omp_get_level	Number of nested parallel regions
omp_get_active_level	Number of nested active par. regions
omp_get_ancestor_thread_num	Thread id of ancestor thread
omp_get_team_size (level)	Size of the thread team at this level

C/C++ : Need to include file <omp.h>
Fortran : Add “use omp_lib” or include file “omp_lib.h”

OpenMP locking routines

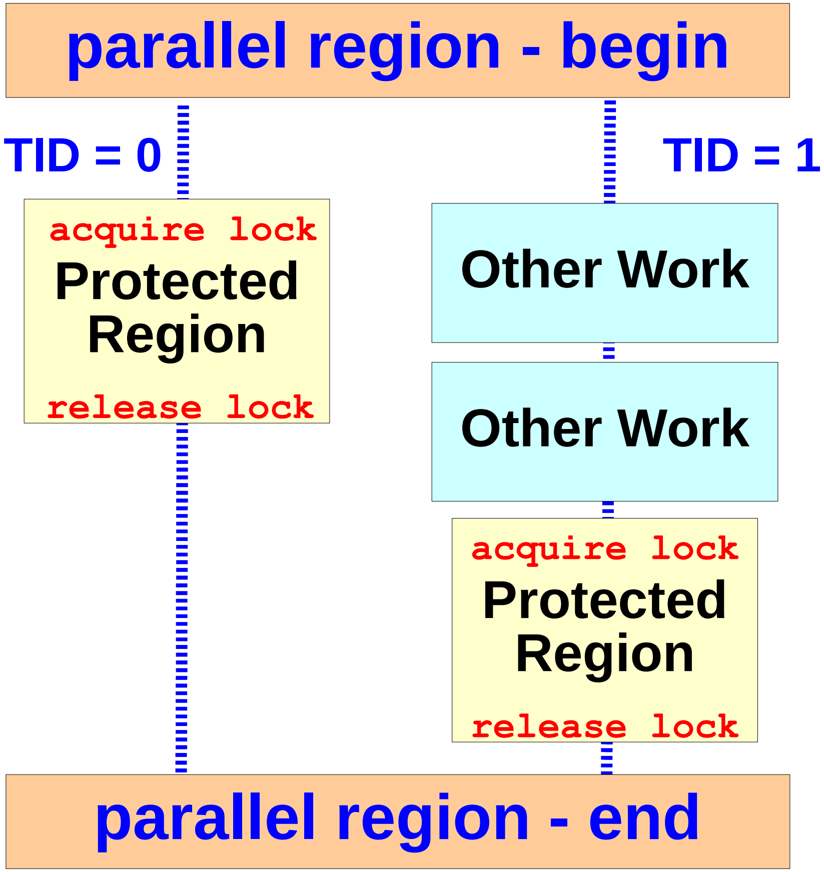
- ***Locks provide greater flexibility over critical sections and atomic updates:***
  - *Possible to implement asynchronous behavior*
  - *Not block structured*
- ***The so-called lock variable, is a special variable:***
  - *C/C++: type omp_lock_t and omp_nest_lock_t for nested locks*
  - *Fortran: type INTEGER and of a KIND large enough to hold an address*
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, <u>and behavior is undefined</u>, in case a lock variable is used without the appropriate initialization***

Nested locking

- ***Simple locks:*** *may not be locked if already in a locked state*
- ***Nestable locks:*** *may be locked multiple times by the same thread before being unlocked*
- ***In the remainder, we discuss simple locks only***
- ***The interface for functions dealing with nested locks is similar (but using nestable lock variables):***

Simple locks	Nestable locks
`omp_init_lock`	`omp_init_nest_lock`
`omp_destroy_lock`	`omp_destroy_nest_lock`
`omp_set_lock`	`omp_set_nest_lock`
`omp_unset_lock`	`omp_unset_nest_lock`
`omp_test_lock`	`omp_test_nest_lock`

OpenMP locking example



- *The protected region contains the update of a shared variable*
- *One thread acquires the lock and performs the update*
- *Meanwhile, the other thread performs some other work*
- *When the lock is released again, the other thread performs the update*

Locking Example - The Code

```
      Program Locks
            ....
      Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else()
       End Do

       Call Do_Work()

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

Example output for 2 threads

```
TID:  1 at 09:07:27 => entered parallel region
TID:  1 at 09:07:27 => done with WAIT loop and has the lock
TID:  1 at 09:07:27 => ready to do the parallel work
TID:  1 at 09:07:27 => this will take about 18 seconds
TID:  0 at 09:07:27 => entered parallel region
TID:  0 at 09:07:27 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:32 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:37 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:42 =>   WAIT for lock - will do something else for  5 seconds
TID:  1 at 09:07:45 => done with my work
TID:  1 at 09:07:45 => done with work loop - released the lock
TID:  1 at 09:07:45 => ready to leave the parallel region
TID:  0 at 09:07:47 => done with WAIT loop and has the lock
TID:  0 at 09:07:47 => ready to do the parallel work
TID:  0 at 09:07:47 => this will take about 18 seconds
TID:  0 at 09:08:05 => done with my work
TID:  0 at 09:08:05 => done with work loop - released the lock
TID:  0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```

Used to check the answer

Note: program has been instrumented to get this information

OpenMP Environment Variables

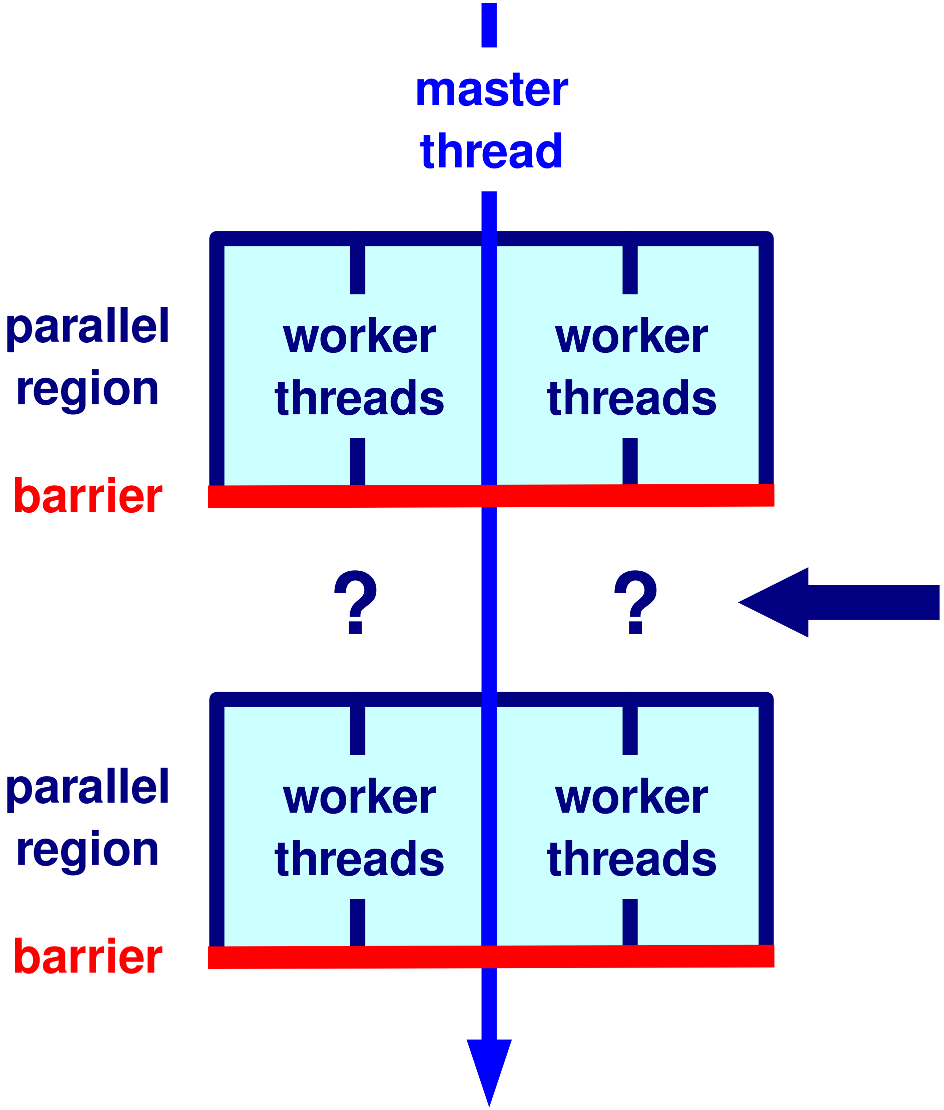
OpenMP Environment Variables

OpenMP environment variable	Default for Oracle Solaris Studio
OMP_NUM_THREADS n	1
OMP_SCHEDULE "schedule,[chunk]"	static, "N/P"
OMP_DYNAMIC { TRUE | FALSE }	TRUE
OMP_NESTED { TRUE | FALSE }	FALSE
OMP_STACKSIZE size [B|K|M|G]	4 MB (32 bit) / 8 MB (64-bit)
OMP_WAIT_POLICY [ACTIVE | PASSIVE]	PASSIVE
OMP_MAX_ACTIVE_LEVELS	4
OMP_THREAD_LIMIT	1024

Note:

The names are in uppercase, the values are case insensitive

Implementing the Fork-Join Model



Use the OMP_WAIT_POLICY environment variable to control the behaviour of idle threads

Tasking In OpenMP

What Is A Task?

A TASK

"A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct"

COMMENT: When a thread executes a task, it produces a task region

TASK REGION

"A region consisting of all code encountered during the execution of a task"

COMMENT: A parallel region consists of one or more implicit task regions

EXPLICIT TASK

"A task generated when a task construct is encountered during execution"

Tasking Directives

```
#pragma omp task
```

```
!$omp task
```

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

          printf("A ");
          printf("race ");
          printf("car ");

   printf("\n");
   return(0);
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

What will this program print ?

Example/2

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("A ");
          printf("race ");
          printf("car ");

   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/3

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note that this program could for example also print
“A A race race car car ” or
“A race A car race car”, or
“A race A race car car”, ...
although I have not observed this (yet)

Example/4

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
          printf("A ");
          printf("race ");
          printf("car ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/5

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

But now only 1 thread executes ....

Example/6

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf(“A “);
        #pragma omp task
          {printf("race ");}
        #pragma omp task
          {printf("car ");}
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/7

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$
```

Tasks can be executed in arbitrary order

Example/8

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf(“A “);
        #pragma omp task
          {printf("race ");}
        #pragma omp task
          {printf("car ");}
        printf(“is fun to watch “);
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/9

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$
```

Tasks are executed at a task execution point

Example/10

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf("A ");
        #pragma omp task
          {printf("car ");}
        #pragma omp task
          {printf("race ");}
        #pragma omp taskwait
        printf("is fun to watch ");
     }
   } // End of parallel region

   printf("\n");retu
}
```

What will this program print using 2 threads ?

Example/11

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

Tasks are executed first now

Task Construct Syntax

C/C++:

```
#pragma omp task [clause [[,]clause] ...]
    structured-block
```

Fortran:

```
!$omp task[clause [[,]clause] ...]
    structured-block
!$omp end task
```

Task Synchronization

- ***Syntax:***
  - ***C/C++:*** `#pragma omp taskwait`
  - ***Fortran:*** `!$omp taskwait`
- ***Current task suspends execution until all children tasks, generated within the current task up to this point, have completed execution***

When are Tasks Complete?

- *At an implicit thread barrier*
- *At an explicit thread barrier*
  - *C/C++:* `#pragma omp barrier`
  - *Fortran:* `!$omp barrier`
- *At a task barrier*
  - *C/C++:* `#pragma omp taskwait`
  - *Fortran:* `!$omp taskwait`

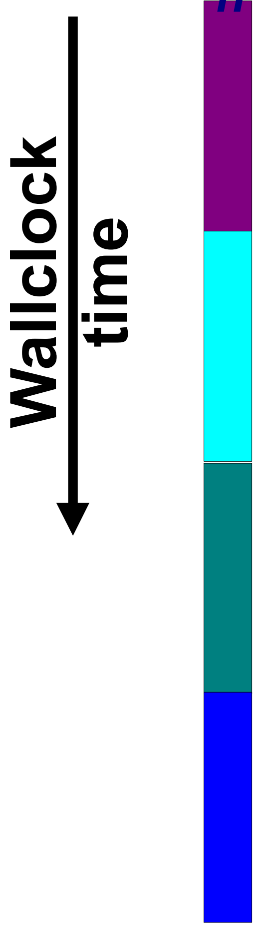
Performance Considerations in Parallel Computing

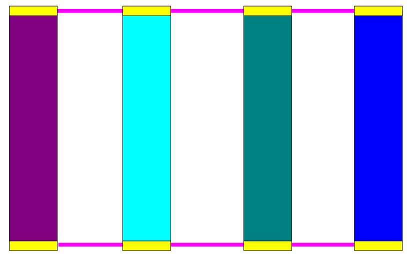
Parallel Overhead

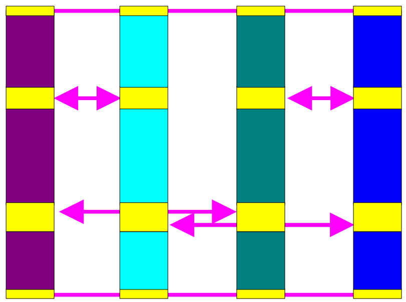
- ***The total CPU time often exceeds the serial CPU time:***
  - ***The newly introduced parallel portions in your program need to be executed***
  - ***Threads need time sending data to each other and synchronizing ("communication")***
    - ✔ ***Often the key contributor, spoiling all the fun***
-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

Communication

Serial Execution	*Parallel - Without communication*	*Parallel - With communication*



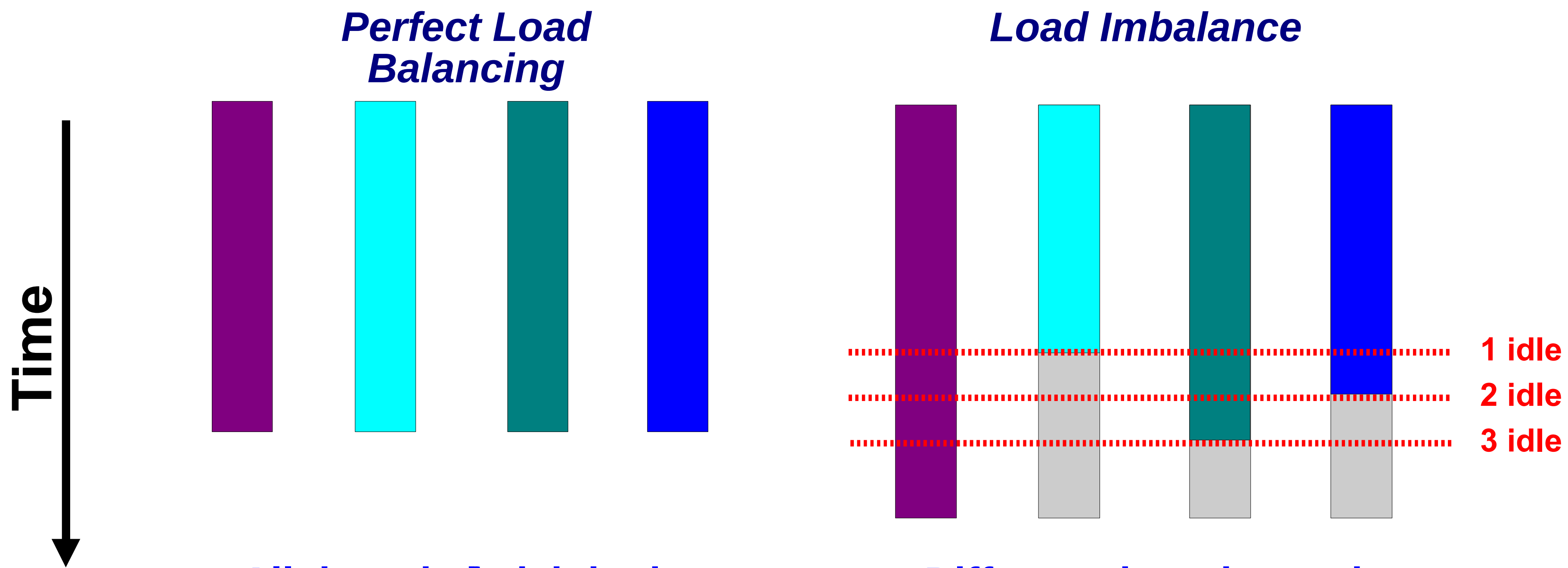




- **Embarrassingly parallel: 4x faster**
- **Wallclock time is ¼ of serial wallclock time**

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

Load Balancing



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

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

Thread is idle

Amdahl's Law/1

Decompose the execution time in 2 parts:

```
T = T(parallel) + T(non-parallel)
```

Describe this with a parameter "f" (between 0 and 1):

```
T = f * T + (1-f) * T
```

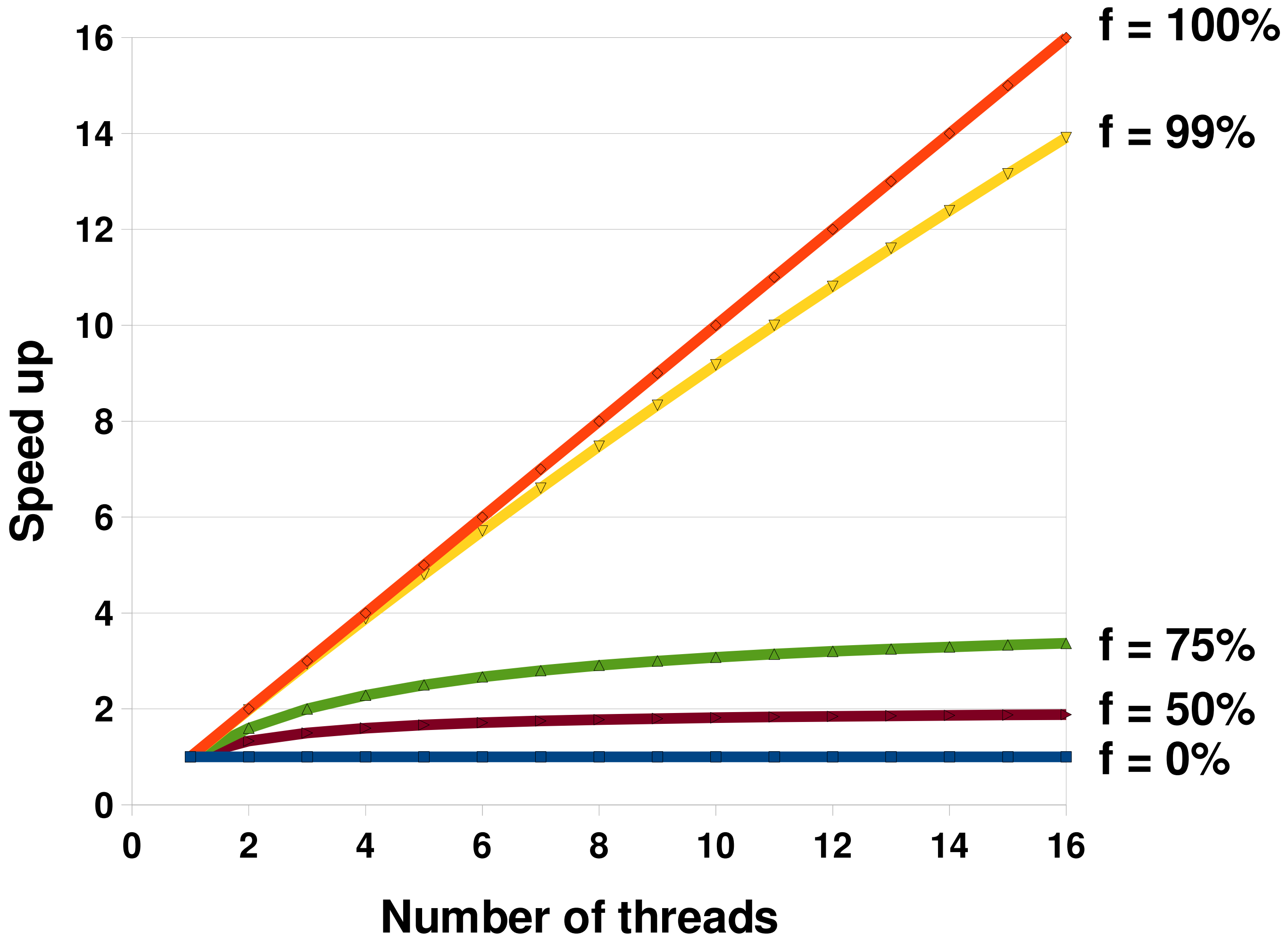
Execution time using "P" cores:

```
T(P) = (f * T)/P + (1-f) * T
```

Amdahl's Law - The Parallel Speed Up S(P) is:

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

Amdahl's Law/2



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":

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

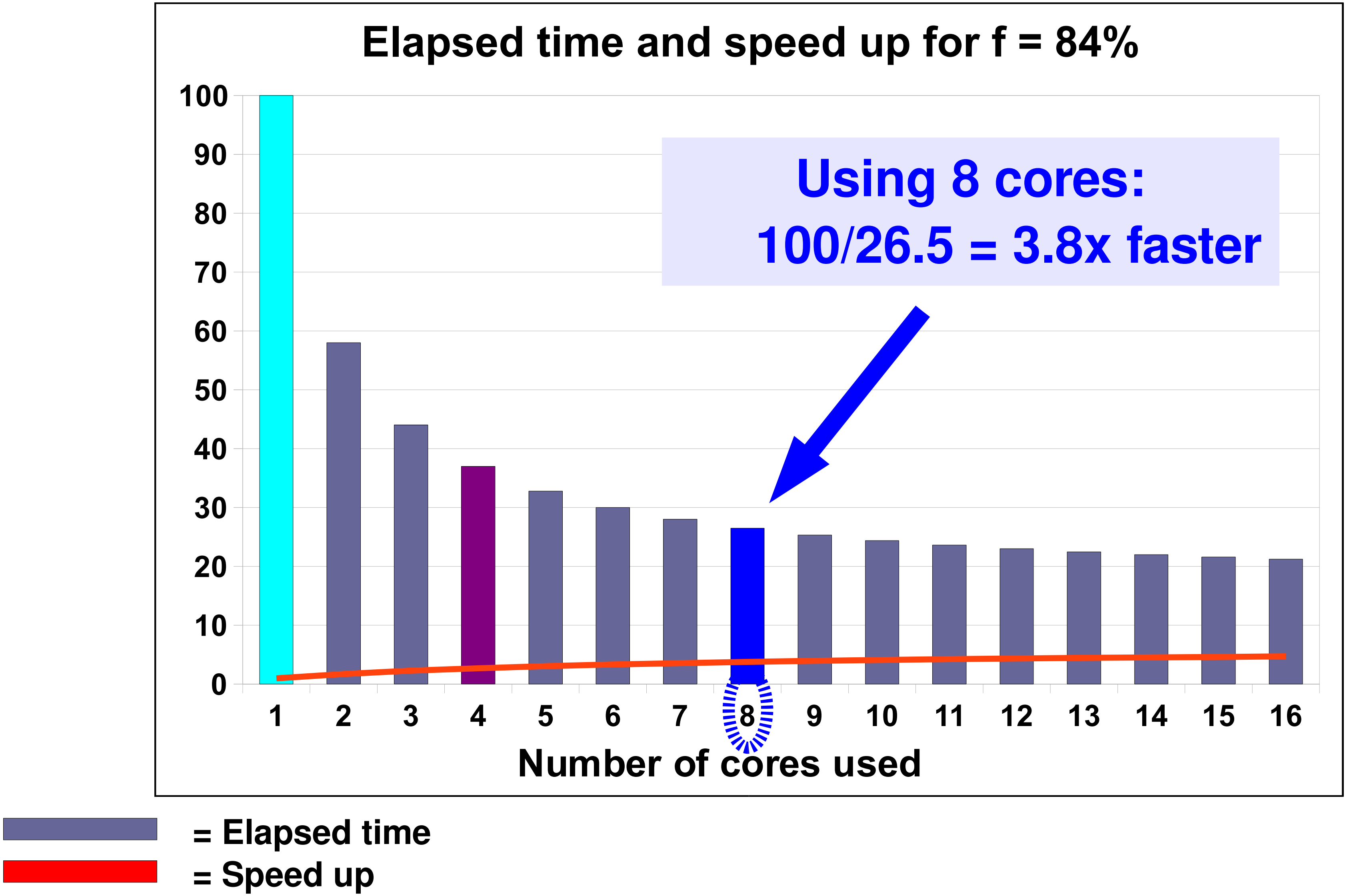
Example:

```
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 Cheaper .....



Summary OpenMP

Summary OpenMP

- ***OpenMP provides for a small, but yet powerful, programming model***
- ***It can be used on a shared memory system of any size***
  - ***This includes a single socket multicore system***
- ***Compilers with OpenMP support are widely available***
- ***The tasking concept opens up opportunities to parallelize a wider range of applications***

Thank You And .... Stay Tuned !

ruud.vanderpas@oracle.com

Hardware and Software

Engineered to Work Together

156

ORACLE®