Introduction to Parallel Programming in OpenMP

David Colignon, ULiège

CÉCI - Consortium des Équipements de Calcul Intensif

http://www.ceci-hpc.be

Slides and source code of examples available on every CÉCI cluster in:

`/CECI/proj/OpenMP/`

Main References

- “Parallel Programming with GCC”,
  Diego Novillo, Red Hat
  Red Hat Summit, Nashville, May 2006
  http://www.airs.com/dnovillo/Papers/rhs2006.pdf
- "An Overview of OpenMP",
  Ruud van der Pas, Oracle
  IWOMP 2010, Tsukuba, 14-16 June 2010
  http://www.compunity.org/training/tutorials/3 Overview_OpenMP.pdf
  and http://openmp.org/wp/2010/07/iwomp-2010-material-available/

More References:

Specification

OpenMP, The OpenMP API specification for parallel programming http://openmp.org/

Articles

Wikipedia (good summary) http://en.wikipedia.org/wiki/Openmp
32 OpenMP traps for C++ developers http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers/
Common Mistakes in OpenMP and How To Avoid Them http://www.michaelsuess.net/.../suess_leopold_common_mistakes_06.pdf
IWOMP 2009, The 2009 International Workshop on OpenMP (Slides) http://openmp.org/wp/2009/06/iwomp2009/
IWOMP 2010, The 2010 International Workshop on OpenMP (Slides) http://openmp.org/wp/2010/07/iwomp-2010-material-available/
Avoiding and Identifying False Sharing Among Threads http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing

Tutorials

Parallel Programming with GCC, D. Novillo, Red Hat Summit, Nashville, May 2006 http://www.airs.com/dnovillo/Papers/rhs2006.pdf
An Overview of OpenMP, IWOMP 2010, Ruud van der Pas, Oracle http://www.compunity.org/training/tutorials/3 Overview_OpenMP.pdf
A "Hands-on" Introduction to OpenMP, SC08, Mattsonand Meadows, Intel http://www.openmp.org/mp-documents/omp-hands-on-SC08.pdf
Cours OpenMP (**en français !**) de l'IDRIS http://www.idris.fr/data/cours/parallel/openmp/
Using OpenMP, SC09, Hartman-Baker R., ORNL, NCCS http://www.greatlakesconsortium.org/events/scaling/files/openmp09.pdf
OpenMP Tutorial, Barney B., LLNL https://computing.llnl.gov/tutorials/openMP/

Books

Using OpenMP - Portable Shared Memory Parallel Programming, by Chapman *et al.* https://mitpress.mit.edu/books/using-openmp
(Download Book Examples and Discuss) http://openmp.org/wp/2009/04/download-book-examples-and-discuss/

Parallel Programming in OpenMP, by Rohit Chandra *et al.* http://www.elsevier.com/wps/find/bookdescription.cws_home/677929/description
(Google Preview) http://books.google.be/books?id=18CmnqIhbhUC

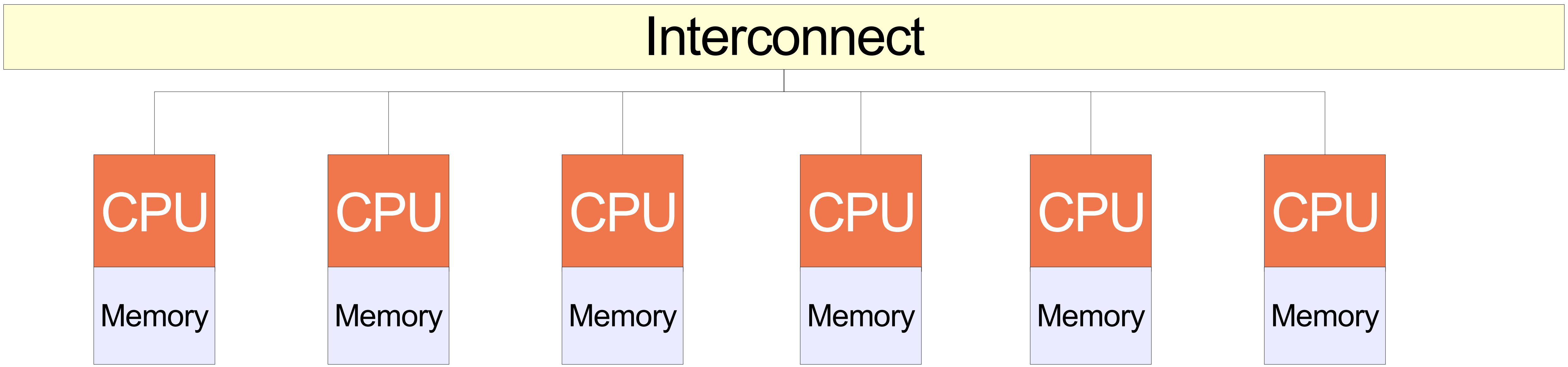
Outline

- Introduction to parallel computing
- Parallel programming models
  - Shared memory
  - Message passing
- OpenMP
  - Guided tour
  - In depth overview

Parallel Computing

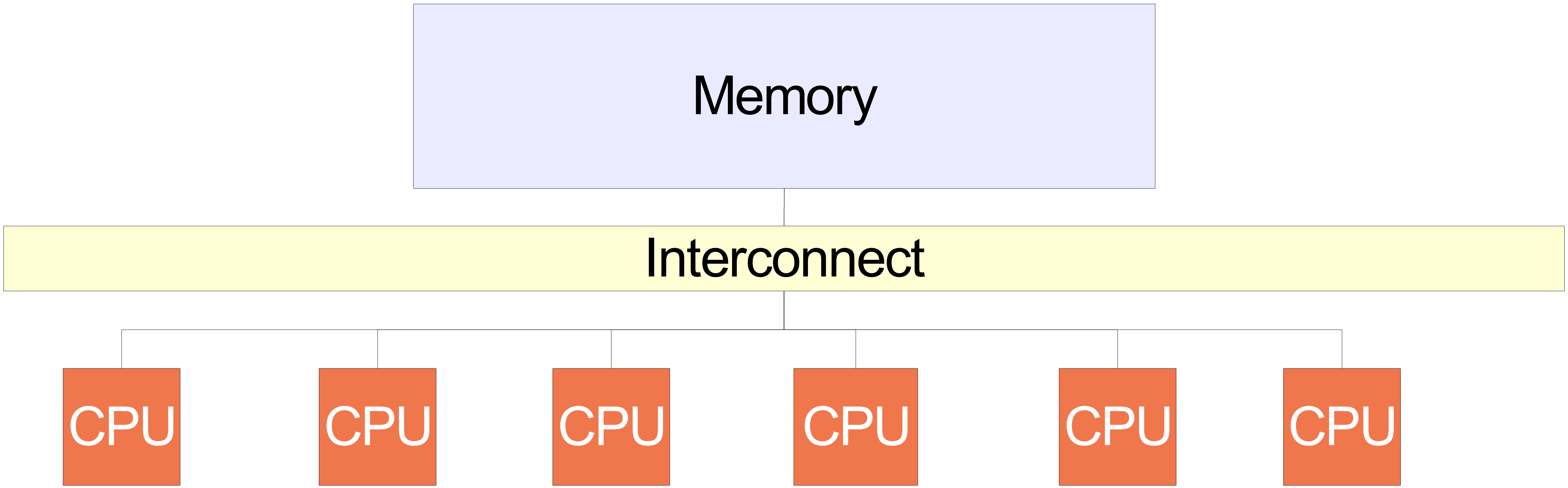
- Use hardware concurrency for increased
  - Performance
  - Problem size
- Two main models
  - Shared memory
  - Distributed memory
- Nature of problem dictates
  - Computation/communication ratio
  - Hardware requirements

Distributed Memory



- Each processor has its own private memory
- Explicit communication
- Explicit synchronization
- Difficult to program but no/few hidden side-effects

Shared Memory



- Processors share common memory
- Implicit communication
- Explicit synchronization
- Simple to program but hidden side-effects

Programming Models

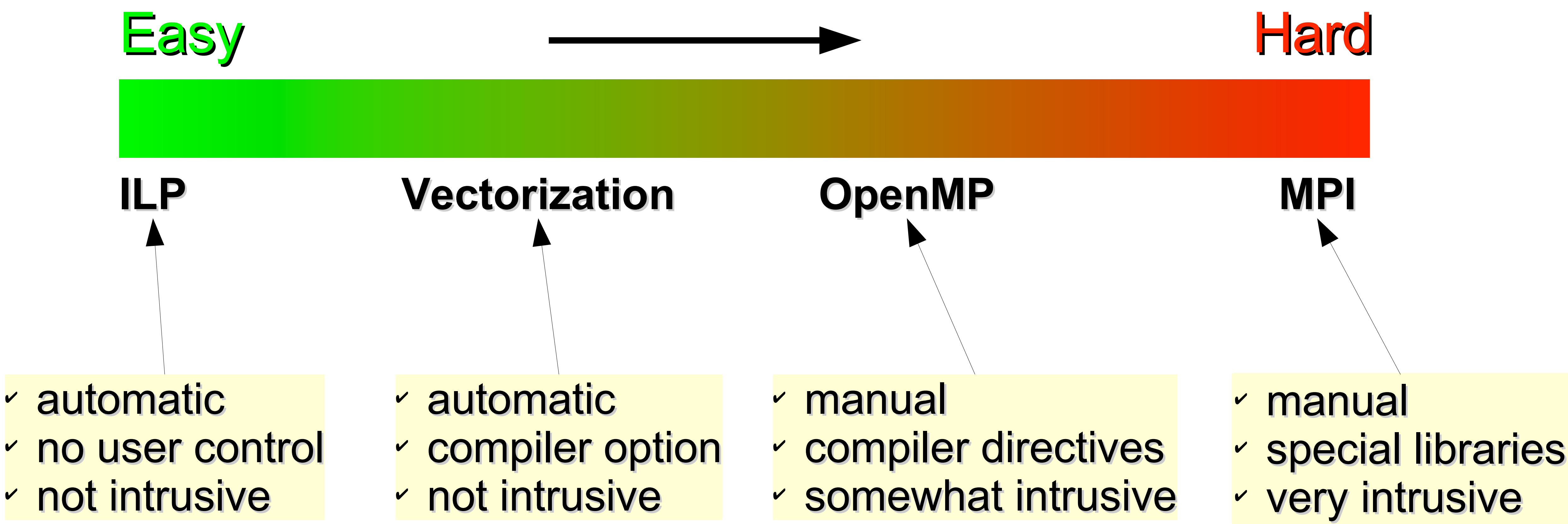
- Shared/Distributed memory often combined
  - Networks of multi-core nodes
  - Parallelism available at various levels
- Additional requirements over sequential
  - Task creation
  - Communication
  - Synchronization
- How do we program these systems?

Explicit Parallelism

- User controls: Tasks, communication and synchronization
- Increased programming complexity
  - Often require different algorithms
- Many different approaches
  - Parallel languages or language extensions: HPF, Occam, Java
  - Compiler annotations: OpenMP
  - Libraries: Pthreads, MPI

Parallelism in GCC

GCC supports four concurrency models



Ease of use not necessarily related to speedups!

Message Passing

- Completely library based
- No special compiler support required
- The "assembly language" of parallel programming
  - Ultimate control
  - Ultimate pain when things go wrong
  - Computation/communication ratio must be high
- Message Passing Interface (MPI) most popular model

Message Passing

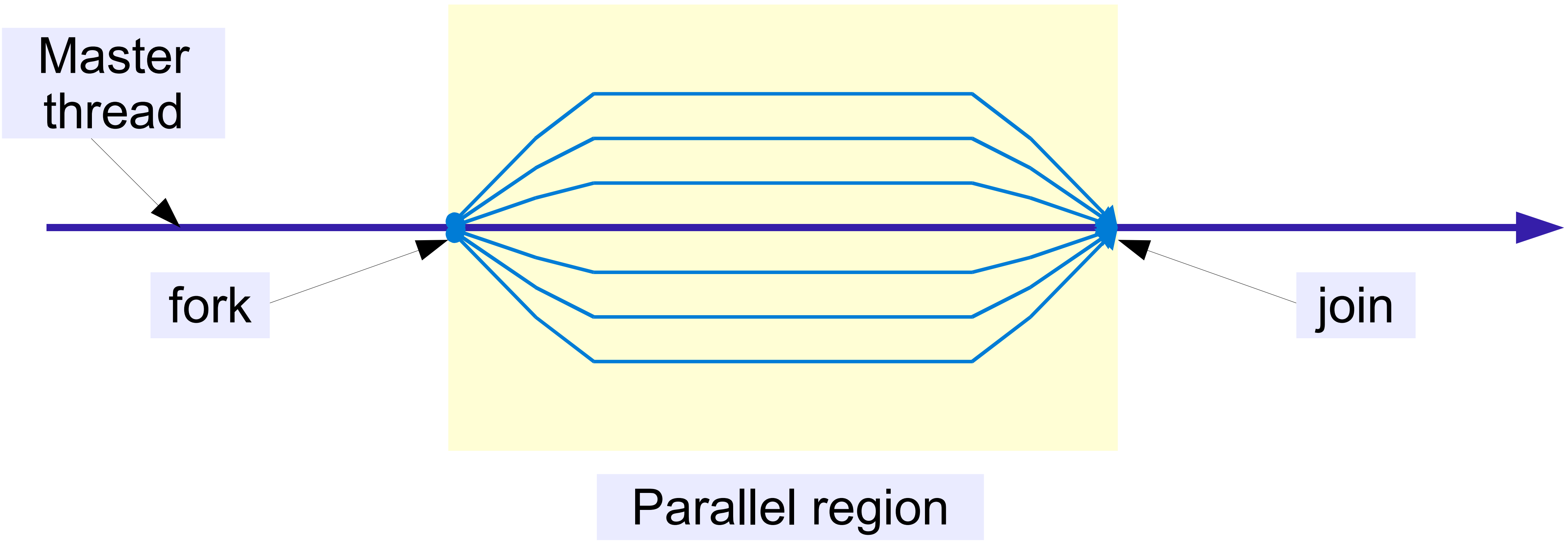
- Separate address spaces
  - It may also be used on a shared memory machine
- Heavy weight processes
- Communication explicit via network messages
  - User responsible for marshalling, sending and receiving

OpenMP - Introduction

- Language extensions for shared memory concurrency
- Supports C, C++ and Fortran
- Embedded directives specify
  - Parallelism
  - Data sharing semantics
  - Work sharing semantics
- Standard and increasingly popular

OpenMP – Programming Model

- Based on fork/join semantics
  - Master thread spawns teams of children threads
  - All threads share common memory
- Allows sequential and parallel execution



OpenMP - Programming Model

- Compiler directives via pragmas (C, C++) or comments (Fortran).
- Compiler replaces directives with calls to runtime library (`libgomp`)
- Runtime controls available via library API and environment variables
- Environment variables control parallelism

OMP_NUM_THREADS	**OMP_SCHEDULE**
OMP_DYNAMIC	**OMP_NESTED**

OpenMP – Programming Model

- Explicit sharing and synchronization
- Threads interact via shared variables
  - Several ways for specifying shared data
  - Sharing always at the variable level
- Programmer responsible for synchronization
  - Unintended sharing leads to “data races”
  - Use synchronization directives and library API
  - Synchronization is expensive

An Overview of OpenMP

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

Getting Started with OpenMP

OpenMP™

http://www.openmp.org

cOMPunity

http://www.compunity.org

OpenMP.org

http://openmp.org/wp/

China ▾ OpenMP ▾ Nomadic ▾

OpenMP.org

http://www.openmp.org

OpenMP™

THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

- »» OpenMP Specifications
- »About OpenMP
- »Compilers
- »Resources
- »Discussion Forum

Events

»IWOMP 2010 - 6th International Workshop on OpenMP, June 14-16, 2010, Tsukuba, Japan

Input Register

Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org

Google™ Custom Search

Search

Archives

- May 2010
- April 2010
- June 2009
- April 2009
- March 2009

OpenMP News

»SPEC Looking For A Few Good Applications

spec®

SPEC, the **Standard Performance Evaluation Corporation**, is looking for realistic OpenMP applications to include in the next version of the SPEC CPU and SPEC OMP benchmark suites.

SPEC is sponsoring a search program, and for each step of the process that a submission passes, SPEC will compensate the Program Submitter (in recognition of the Submitter's effort and skill). A submission that passes all of the steps and is included in the next SPEC CPU benchmark suite will receive $5000 US overall and a license for the new benchmark suite when released. Details on the Benchmark Search Program at: **http://www.spec.org/cpuv6/**.

Posted on May 20, 2010

»IWOMP 2010: International Workshop on OpenMP

6th International Workshop on OpenMP, June 14-16, 2010, Tsukuba, Japan

"Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More"

The International Workshop on OpenMP is an annual series of workshops dedicated to the promotion and advancement of all aspects focusing on parallel programming with OpenMP. OpenMP is now a major programming model for shared memory systems from multi-core machines to large scale servers. Recently, new ideas and challenges are proposed to extend OpenMP framework for adopting accelerators and also exploiting parallelism beyond loop levels. The workshop serves as a forum to present the latest research ideas and results related to this shared memory programming model. It also offers the opportunity to interact with OpenMP users, developers and the people working on the next release of the standard. The 2010 International Workshop on OpenMP **IWOMP 2010** will be held in the high-tech city of Tsukuba, Japan.

The workshop **IWOMP 2010** will be a three-day event. In the first day, tutorials are provided for focusing on topics of interest to current and prospective OpenMP developers, suitable for both

The OpenMP API

supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»Read about OpenMP.org

Get

»OpenMP specs

Use

»OpenMP Compilers

Learn

»Using OpenMP – the book
»Using OpenMP – the examples

ORACLE

Shameless Plug - “Using OpenMP”

“Using OpenMP”

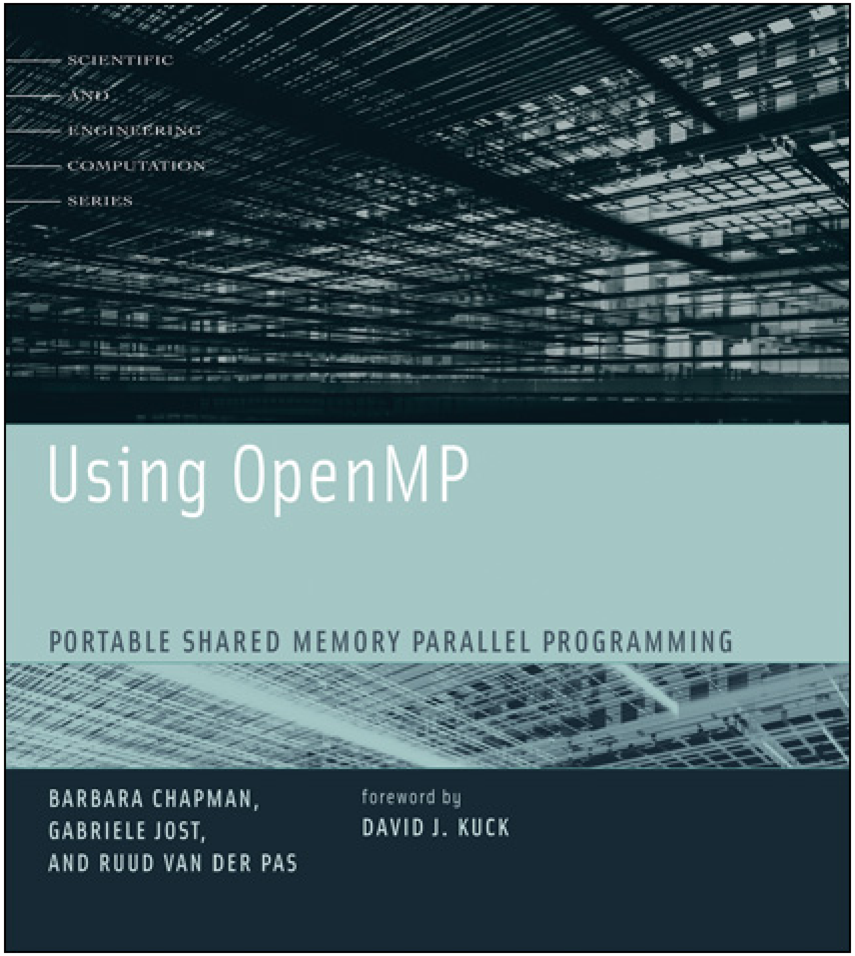
Portable Shared Memory Parallel Programming

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



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***
  - ***Environment variables***
- ***Specif cation maintained by the OpenMP Architecture Review Board (http://www.openmp.org)***
- ***Version 3.0 has been released May 2008***

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 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

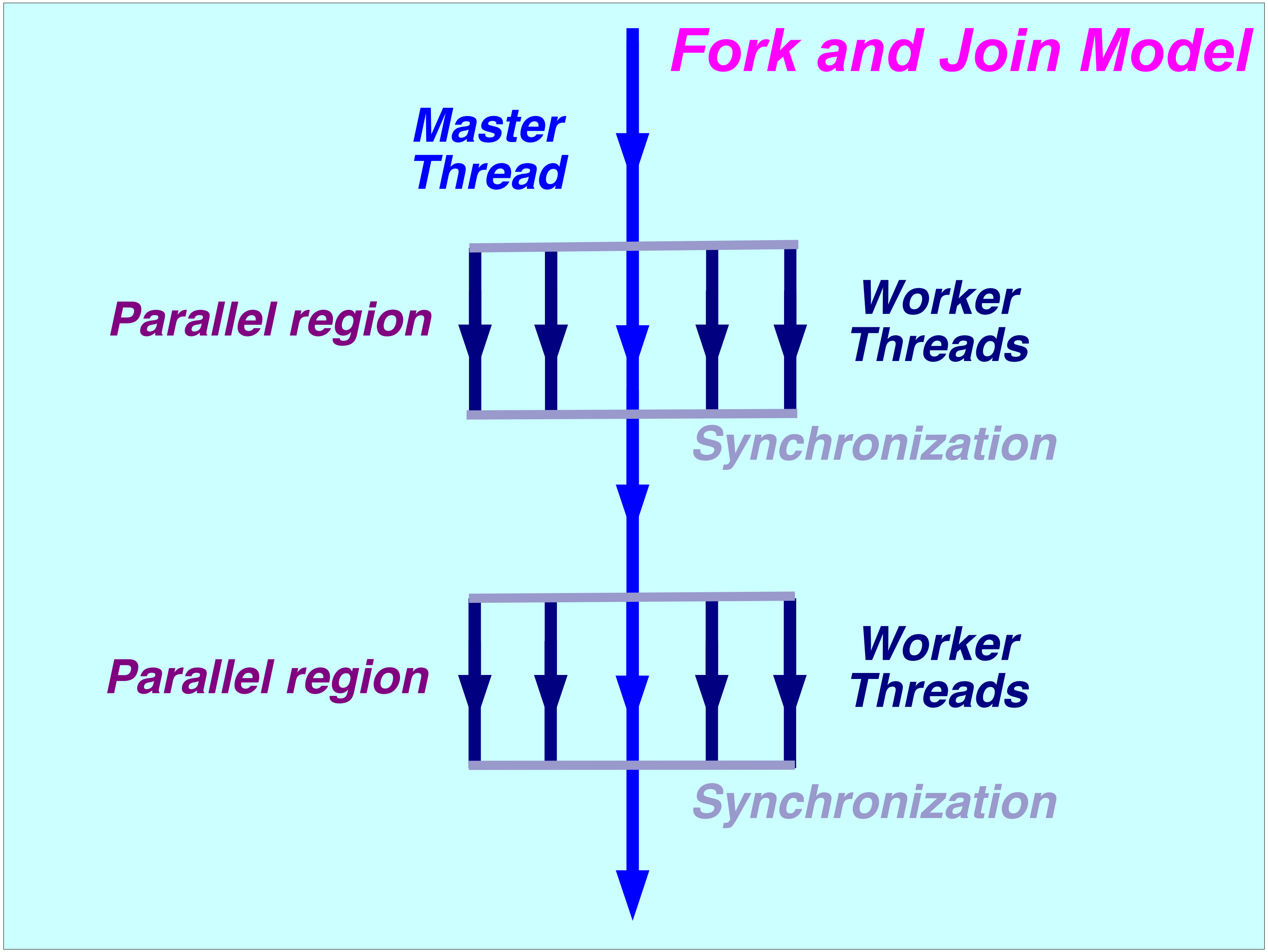
Memory and threading model map naturally

Lightweight

Mature

Widely available and used

The OpenMP Execution Model



Process

A process is created by the operating system, and requires a fair amount of "overhead".

Processes contain information about program resources and program execution state, including:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Thread

A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

Threads use and exist within the process resources
- are able to be scheduled by the operating system
- run as independent entities
- they duplicate only the bare essential resources that enable them to exist as executable code.

This independent flow of control is accomplished because a thread maintains its own:
- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.

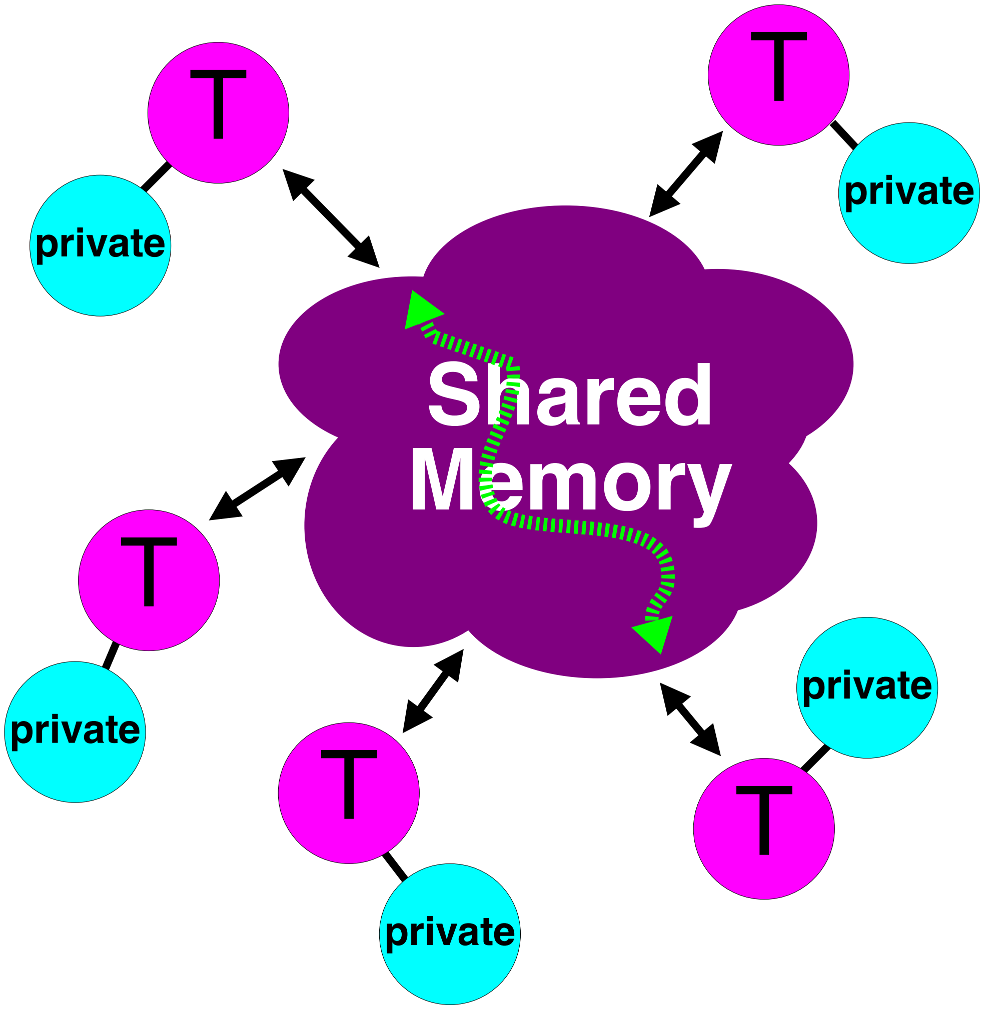
Threads may share the process resources with other threads that act equally independently (and dependently)

Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Thread die if the parent process dies

Thread Iis "lightweight" because most of the overhead has already been accomplished through the creation of its process.

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*
    - *All threads can read and write the data simultaneously, unless protected through a specif c OpenMP 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*

The private and shared clauses

private (list)

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

shared (list)

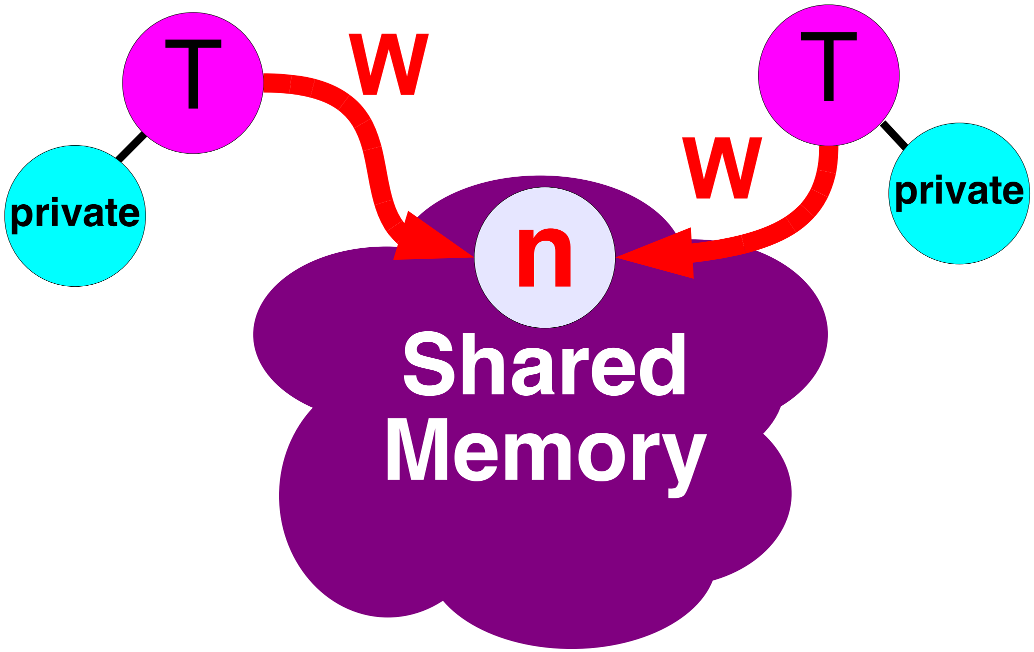
- *Data is accessible by all threads in the team*
- *All threads access the same address space*

What is a Data Race?

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

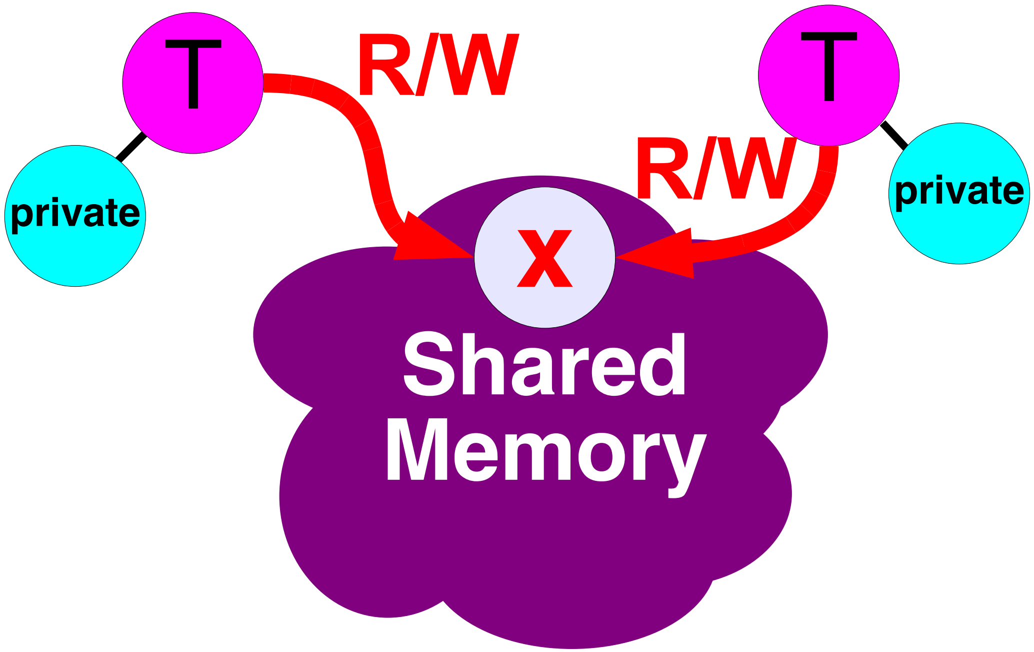
Example of a data race

```
#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
```

An OpenMP example

For-loop with independent iterations

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

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

```
$ cc -xopenmp source.c
$ export OMP_NUM_THREADS=5
$ ./a.out
```

Example Parallel Execution

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
i=0-199	i=200-399	i=400-599	i=600-799	i=800-999
a[i]	a[i]	a[i]	a[i]	a[i]
+	+	+	+	+
b[i]	b[i]	b[i]	b[i]	b[i]
=	=	=	=	=
c[i]	c[i]	c[i]	c[i]	c[i]

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*
- ***A <u>work-sharing construct</u> divides the execution of the enclosed code region among the members of the team; in other words: they split the work***

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*

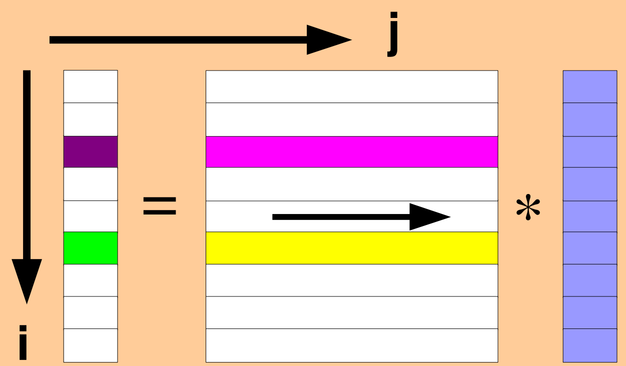
- ❑ *Fortran: directives are case insensitive*
  - *Syntax:* sentinel directive [clause [[,] clause]...]
  - *The sentinel is one of the following:*
    - ✔ *!$OMP or C$OMP or *$OMP* *(f xed format)*
    - ✔ *!$OMP* *(free format)*
- ❑ *Continuation: follows the language syntax*
- ❑ *Conditional compilation:* !$ *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 specif c clause(s) that can be used, depend on the directive*

Example 2 - 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

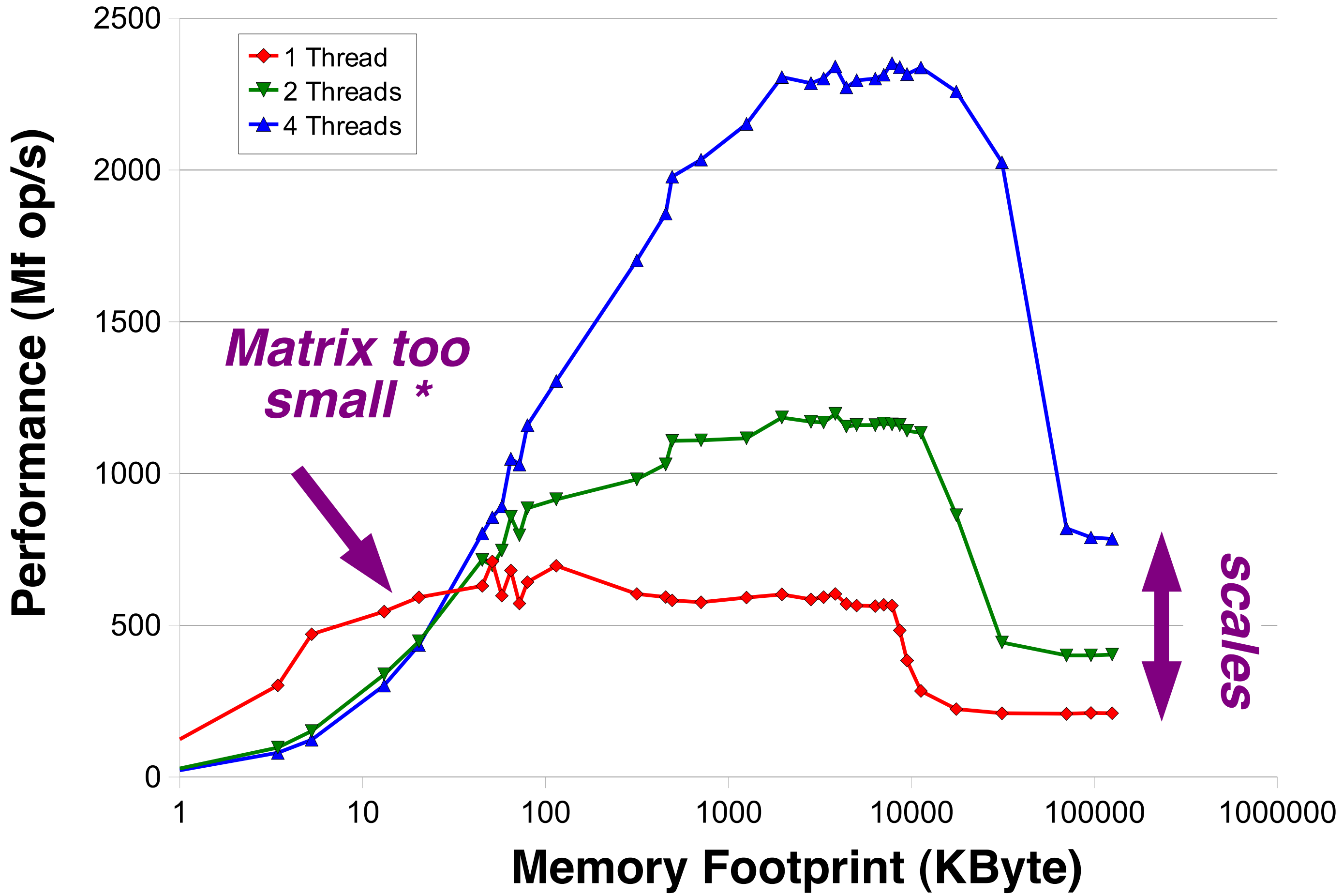
```
for (i=0,1,2,3,4)
 i = 0
sum =    b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
   a[1] = sum
```

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*

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 --*/
```

IWOMP International Workshop on OpenMP

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 updated all of a[ ] f rst, before using a[ ] **

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

wait !

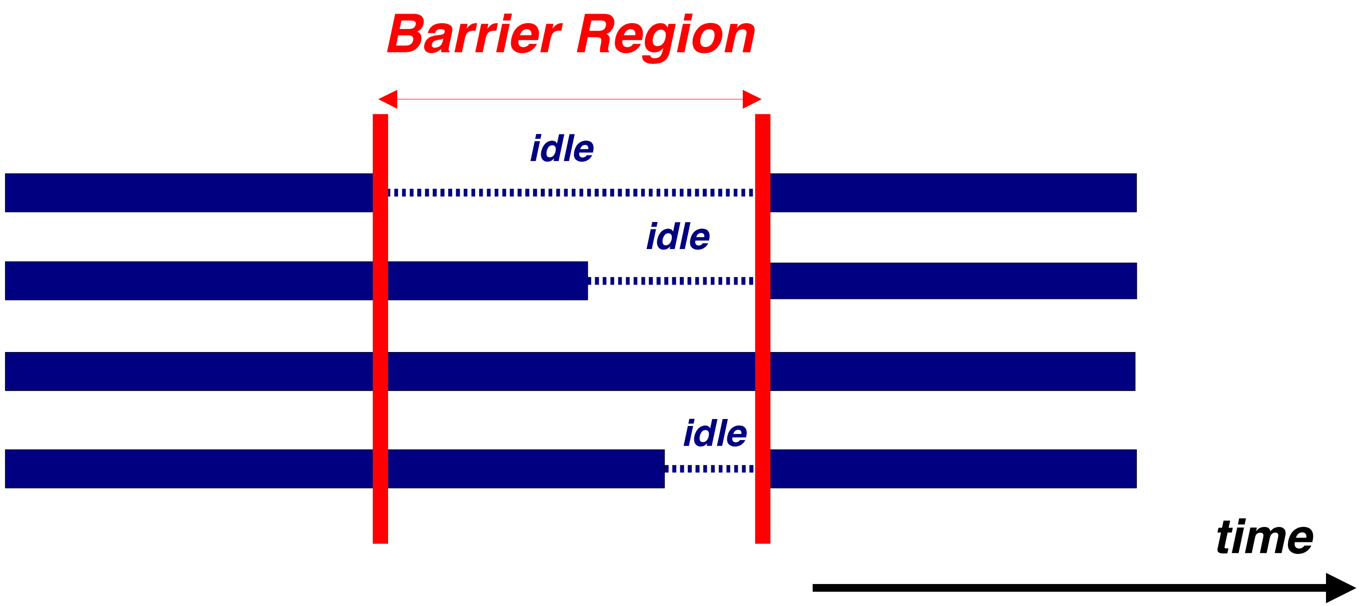
barrier

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

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

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

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

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

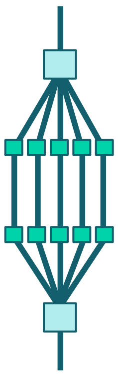
The Parallel Region

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)
```

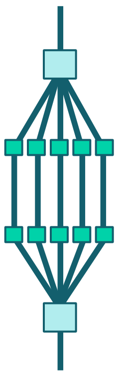
```
!$omp parallel [clause[[,] clause] ...]

    "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/2



```
#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/3

```
$ 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
$
```

The Worksharing Constructs

The OpenMP 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 specif ed)*
- ☞ *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]
```

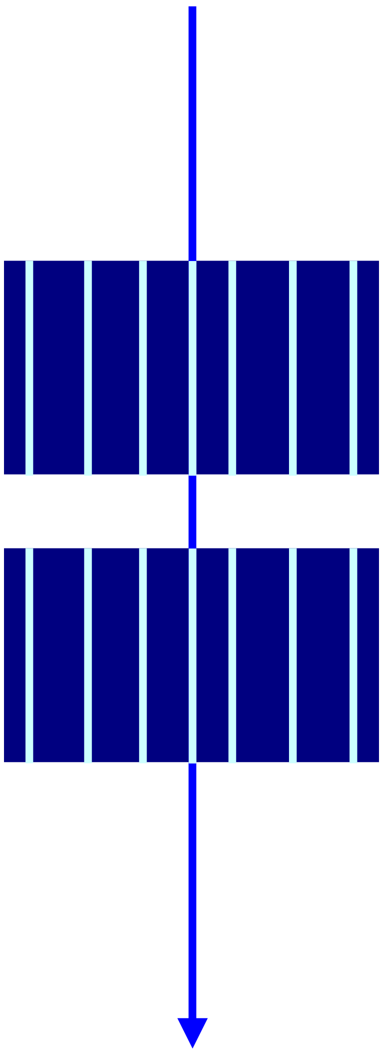
Example:

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

The omp for directive - Example

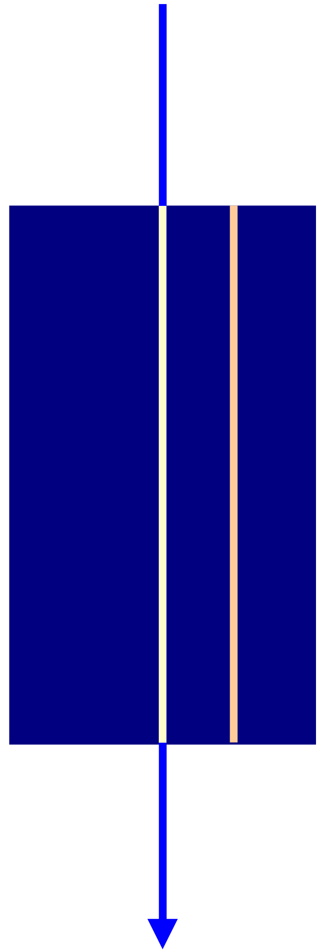
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

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

(implied barrier)

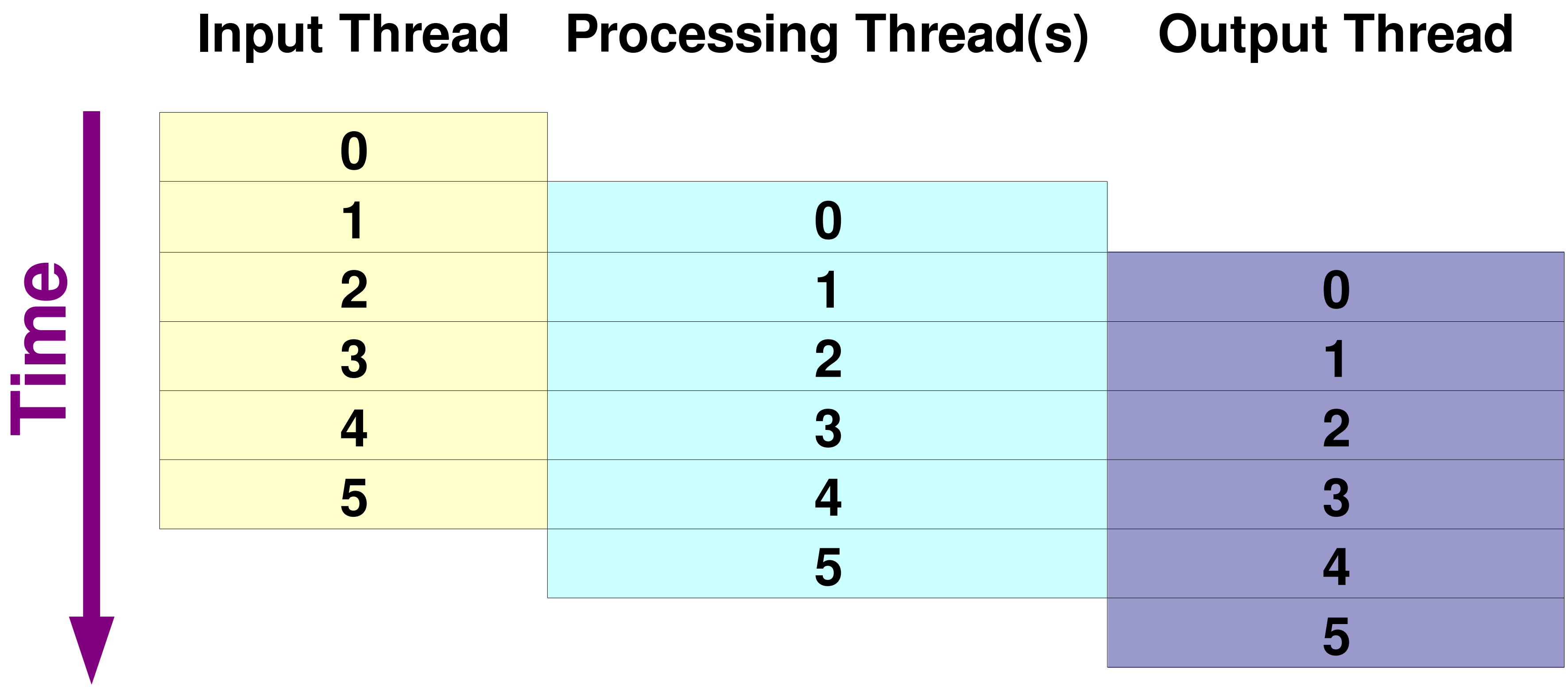


The sections directive - Example

```
#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



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
{
     .....
```

one volunteer requested

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

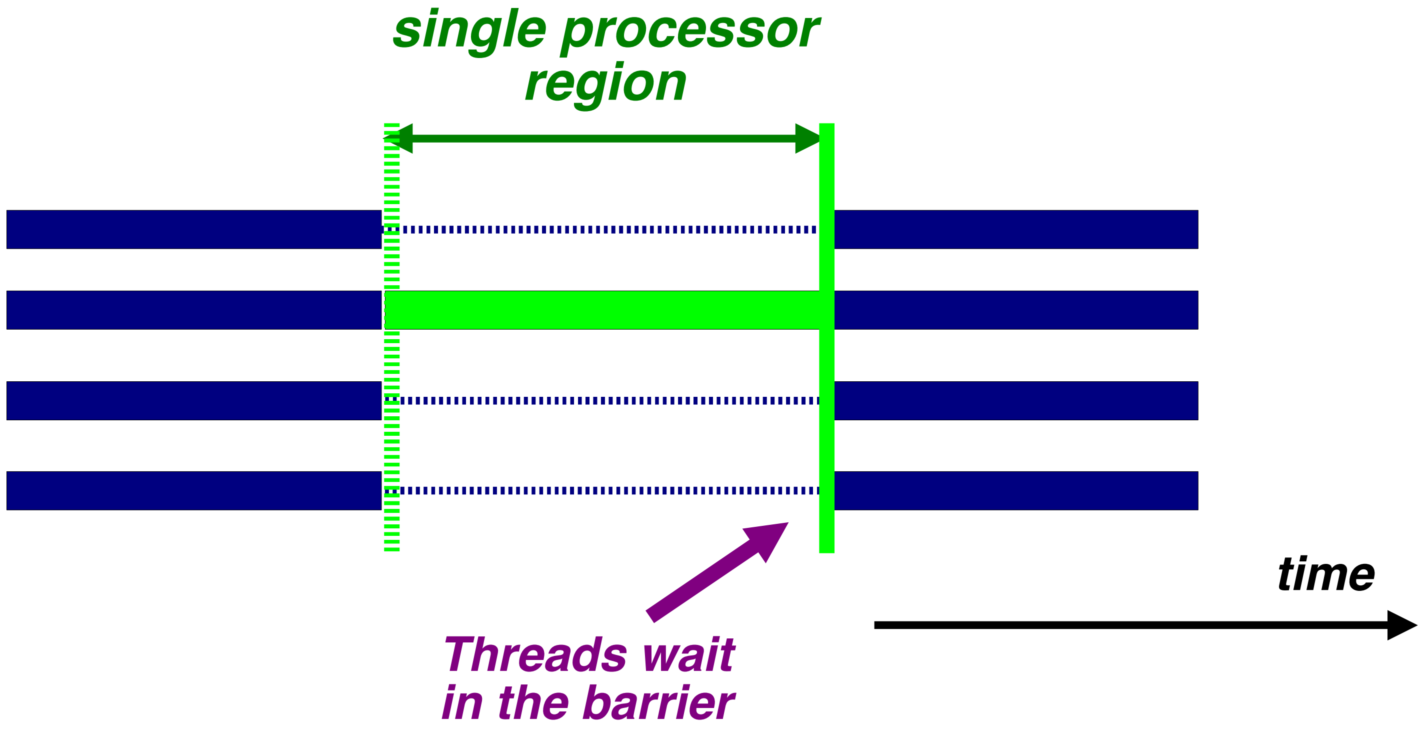
thanks, we're done

```
     .....
}
```

Parallel Version

May have to insert a barrier here

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

Single PARALLEL loop

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

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

```
!$omp parallel
!$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
```

Single PARALLEL sections

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

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

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

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

Orphaning

```
          :
#pragma omp parallel
{
          :
    (void) dowork();
          :
}
          :
```

```
void dowork()
{
          :
  #pragma omp for
   for (int i=0;i<n;i++)
   {
          :
   }
          :
}
```

orphaned work-sharing directive

- *The OpenMP specif cation does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned*
- *That is, they can appear outside the lexical extent of a parallel region*

More on orphaning

```
(void) dowork(); !- Sequential FOR

#pragma omp parallel
{
   (void) dowork(); !- Parallel FOR
}
```

```
void dowork()
{
#pragma omp for
   for (i=0;....)
   {
      :
   }
}
```

- ***When an orphaned worksharing or synchronization directive is encountered in the <u>sequential part</u> of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored***

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 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

Using OpenMP

Using OpenMP

- *We have already seen many features of OpenMP*
- *We will now cover*
  - *Additional language constructs*
  - *Features that may be useful or needed when running an OpenMP application*
- *The tasking concept is covered in separate section*

About storage association

- ***Private variables are undefined on entry and exit of the parallel region***
- ***A private variable within a parallel region has no storage association with the same variable outside of the region***
- ***Use the first/last private clause to override this behavior***
- ***We illustrate these concepts with an example***

Example private variables

```
main()
{
  A = 10;

 #pragma omp parallel
 {
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
   for (i=0; i<n; i++)
   {
       ....
       B = A + i;            /*-- A undefined, unless declared
                                  firstprivate --*/

       ....
   }

   C = B;                    /*-- B undefined, unless declared
                                  lastprivate --*/


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

Disclaimer: This code fragment is not very meaningful and only serves to demonstrate the clauses

The first/last private clauses

firstprivate (list)

- ✔ ***All variables in the list are initialized with the value the original object had before entering the parallel construct***

lastprivate (list)

- ✔ ***The thread that executes the <u>sequentially last</u> iteration or section updates the value of the objects in the list***

The default clause

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

default (none | shared) — *C/C++*

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*

firstprivate

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

The reduction clause - Example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

- *Care needs to be taken when updating shared variable SUM*
- *With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided*

The reduction clause

reduction ([operator | intrinsic]) : list) *Fortran*

reduction (operator : list) *C/C++*

- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is def ned as:*

Check the docs for details

Fortran

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr_list)
x = intrinsic (expr_list, x)
```

C/C++

```
x = x operator expr
x = expr operator x
x++, ++x, x--, --x
x <binop> = expr
```

- ✔ *Note that the value of a reduction variable is undef ned from the moment the f rst thread reaches the clause till the operation has completed*
- ✔ *The reduction can be hidden in a function call*

Fortran - Allocatable Arrays

- *Fortran allocatable arrays whose status is "currently allocated" are allowed to be specif ed as private, lastprivate, f rstprivate, reduction, or copyprivate*

```
integer, allocatable,dimension (:) :: A
integer i

allocate (A(n))

!$omp parallel private (A)
      do i = 1, n
         A(i) = i
      end do
      ...
!$omp end parallel
```

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 def ned*
- ✔ *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 f nishes, it starts on the next portion of work***

guided [, chunk]

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

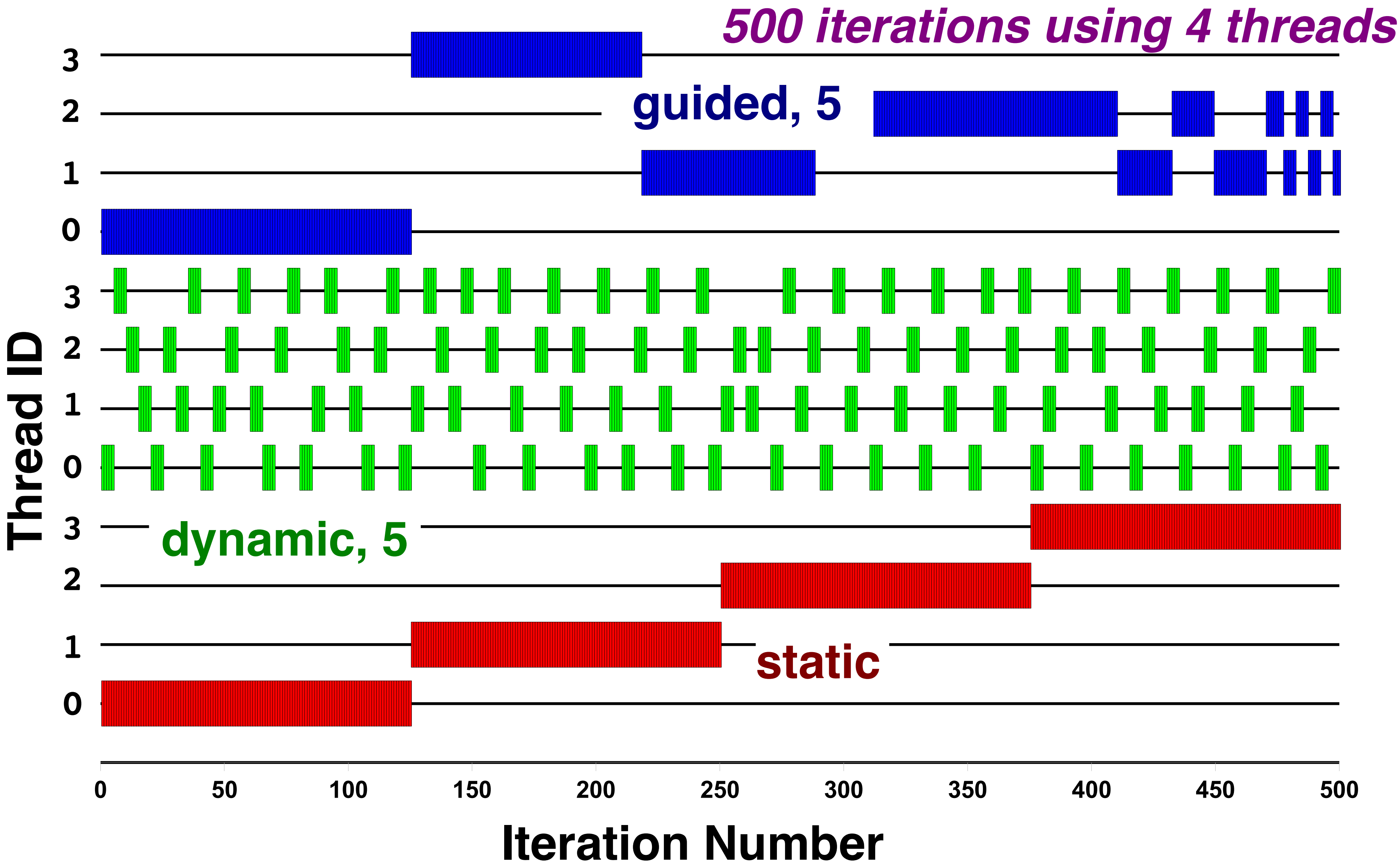
auto

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

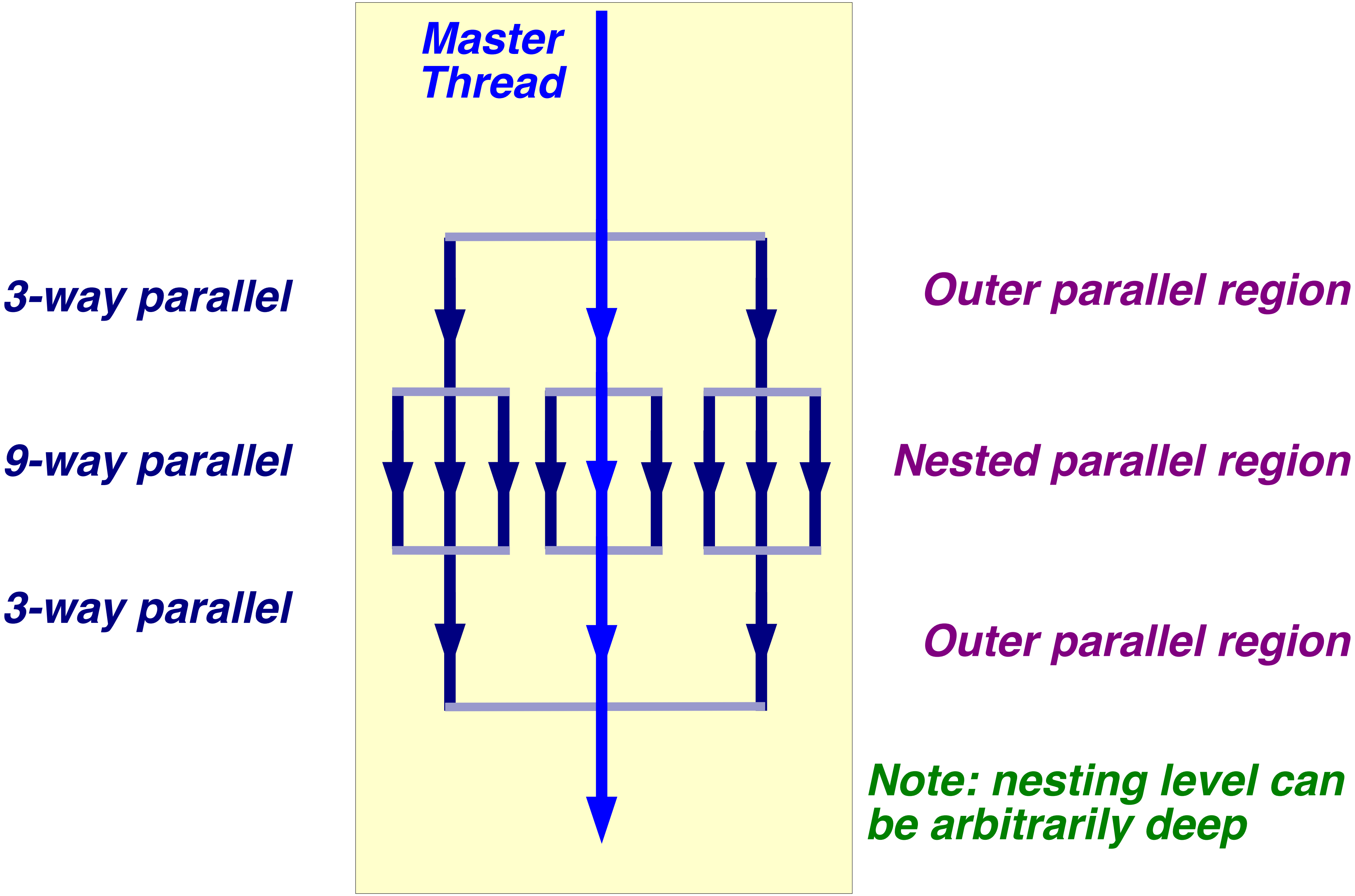
runtime

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

The Experiment



Nested Parallelism



Note: nesting level can be arbitrarily deep

Loop Collapse

- *Allows parallelization of perfectly nested loops without using nested parallelism*
- **collapse** *clause on for/do loop indicates how many loops should be collapsed*
- *Compiler forms a single loop and then parallelizes this*

```
!$omp parallel do collapse(2) ...
  do i = il, iu, is
     do j = jl. ju. js
        do k = kl, ku, ks
           .....
        end do
     end do
  end do
!$omp end parallel do
```

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
```

Additional Directives/2

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

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

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

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

The Master Directive

Only the <u>master thread</u> executes the code block:

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

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

There is no implied barrier on entry or exit !

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:

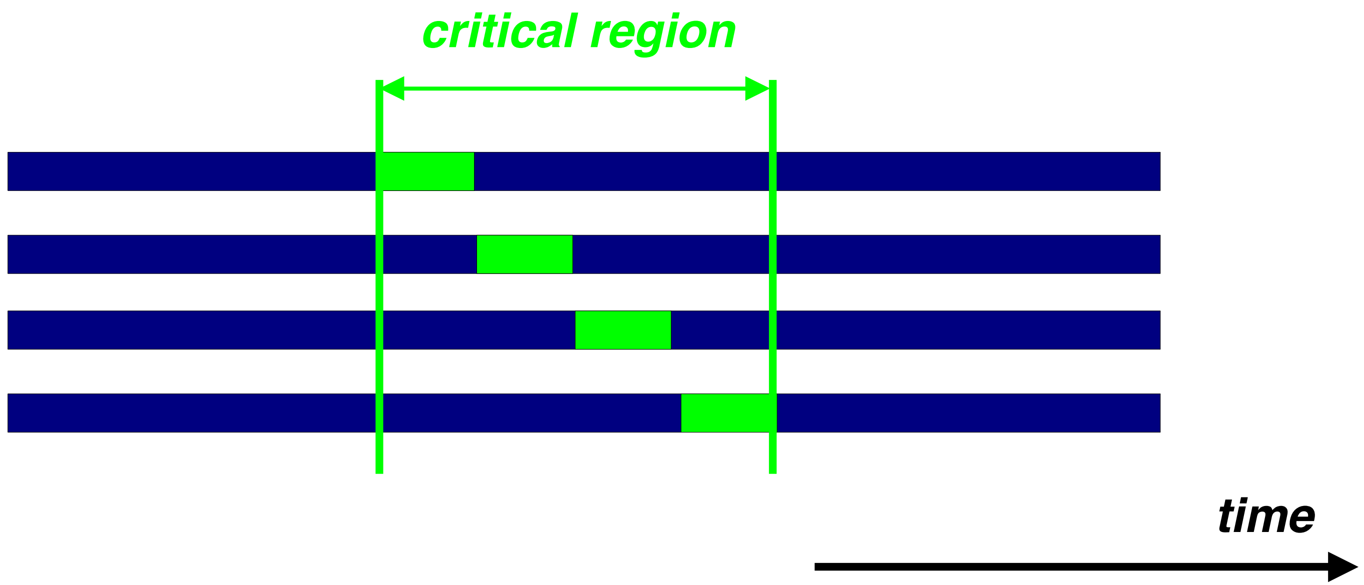
```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

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];
```

More synchronization constructs

The enclosed block of code is executed in the order in which iterations would be executed sequentially:

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

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

May introduce serialization (could be expensive)

Ensure that all threads in a team have a consistent view of certain objects in memory:

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

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

In the absence of a list, all visible variables are f ushed

Implied f ush regions

- *During a* **barrier** *region*
- *At exit from* **worksharing** *regions, unless a* **nowait** *is present*
- *At entry to and exit from* **parallel, critical, ordered** *and* **parallel worksharing** *regions*
- *During* **omp_set_lock** *and* **omp_unset_lock** *regions*
- *During* **omp_test_lock, omp_set_nest_lock, omp_unset _nest_lock** *and* **omp_test_nest_lock** *regions, if the region causes the lock to be set or unset*
- *Immediately before and after every* **task scheduling** *point*
- *At entry to and exit from* **atomic** *regions, where the list contains only the variable updated in the atomic construct*
- *A f ush region is <u>not implied</u> at the following locations:*
  - *At entry to a* **worksharing** *region*
  - *At entry to or exit from a* **master** *region*

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***
- ***Oracle Solaris Studio has extensive support for OpenMP developers***

Example: Serial PI Program

```
static long num_steps = 100000;
double step;
void main ()
{           int i;    double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
    }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

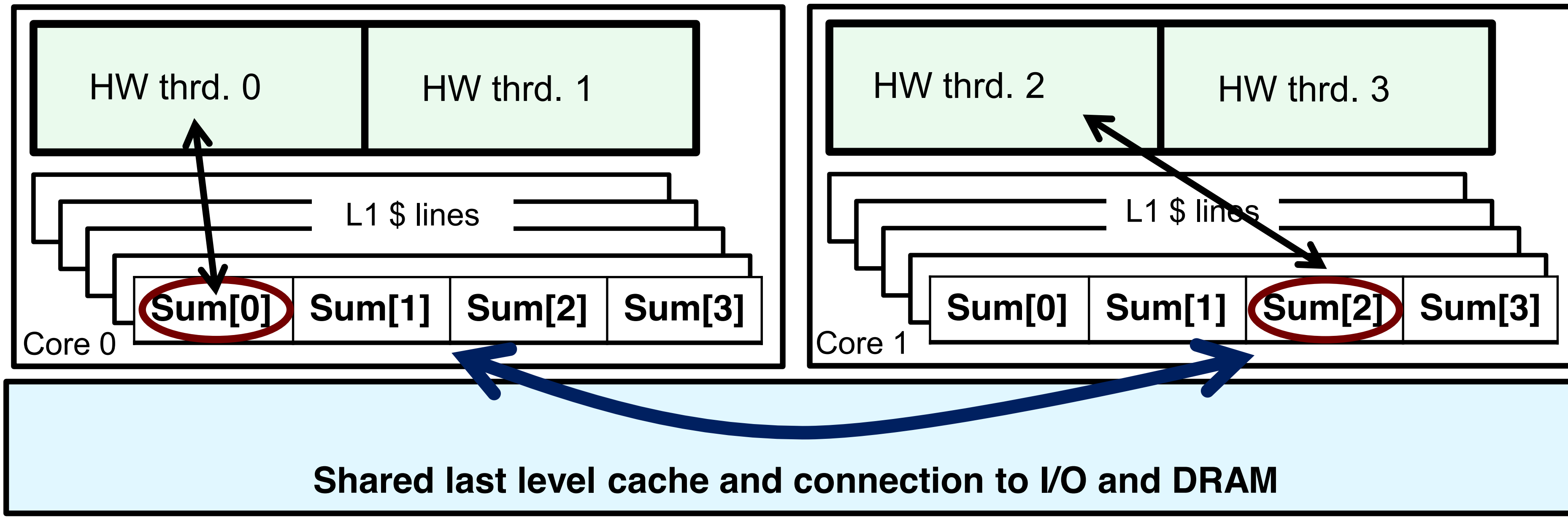
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS];
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
             int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)  nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id] += 4.0/(1.0+x*x);
             }
   }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Why such poor scaling? False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads … This is called **“false sharing”**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD     8           // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS][PAD};
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {        int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id][0] += 4.0/(1.0+x*x);
             }
   }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

Results*: pi program padded accumulator

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double  step;
#define   PAD     8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{            int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id][0] += 4.0/(1.0+x*x);
             }
   }
             for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1^{st} SPMD	1^{st} SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Getting OpenMP Up To Speed

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

- ❑ *The Myth*
- ❑ *Deep Trouble*
- ❑ *Get Real*
- ❑ *The Wrapping*

File Edit View History Bookmarks Tools Help

Getting Started

myth - Informat

"A myth, in popular use, is something that is widely believed but false." ......

Myth

Wikipedia, the free encyclopedia – Cite This Source

Myth may refer to:

Mythology, mythography, or folkloristics. In these academic fields, a myth (*mythos*) is a sacred story concerning the origins of the world or how the world and the creatures in it came to have their present form. The active beings in myths are generally gods and heroes. Myths often are said to take place before recorded history begins. In saying that a myth is a sacred narrative, what is meant is that a myth is believed to be true by people who attach religious or spiritual significance to it. Use of the term by scholars does not imply that the narrative is either true or false. See also legend and tale.

A myth, in popular use, is something that is widely believed but false. This usage, which is often pejorative, arose from labeling the religious stories and beliefs of other cultures as being incorrect, but it has spread to cover non-religious beliefs as well. Because of this usage, many people take offense when the religious narratives they believe to be true are called myths (see Religion and mythology for more information). This usage is frequently confused with fiction, legend, fairy tale, folklore, fable, and urba[n] ... distinct meaning in academia.

- Phoenix Myth
- Myth Nighclub
- Golf Myth
- Atlantis Myth
- The Beauty Myth

Indicates premium content, which is available only to subscribers.

(source: www.reference.com)

Done

The Myth

“OpenMP Does Not Scale”

Hmmm .... What Does That Really Mean ?

Some Questions I Could Ask

"Do you mean you wrote a parallel program, using OpenMP and it doesn't perform?"

"I see. Did you make sure the program was fairly well optimized in sequential mode?"

"Oh. You didn't. By the way, why do you expect the program to scale?"

"Oh. You just think it should and you used all the cores. Have you estimated the speed up using Amdahl's Law?"

"No, this law is not a new EU environmental regulation. It is something else."

"I understand. You can't know everything. Have you at least used a tool to identify the most time consuming parts in your program?"

Some More Questions I Could Ask

“Oh. You didn't. You just parallelized all loops in the program. Did you try to avoid parallelizing innermost loops in a loop nest?”

“Oh. You didn't. Did you minimize the number of parallel regions then?”

“Oh. You didn't. It just worked fine the way it was.

“Did you at least use the nowait clause to minimize the use of barriers?”

“Oh. You've never heard of a barrier. Might be worth to read up on.”

“Do all processors roughly perform the same amount of work?”

“You don't know, but think it is okay. I hope you're right.”

International Workshop IWOMP on OpenMP

I Don't Give Up That Easily

"Did you make optimal use of private data, or did you share most of it?"

"Oh. You didn't. Sharing is just easier. I see.

"You seem to be using a cc-NUMA system. Did you take that into account?"

"You've never heard of that either. How unfortunate. Could there perhaps be any false sharing affecting performance?"

"Oh. Never heard of that either. May come handy to learn a little more about both."

"So, what did you do next to address the performance ?"

"Switched to MPI. Does that perform any better then?"

"Oh. You don't know. You're still debugging the code."

Going Into Pedantic Mode

"While you're waiting for your MPI debug run to finish (are you sure it doesn't hang by the way), please allow me to talk a little more about OpenMP and Performance."

International Workshop IWOMP on OpenMP

Deep Trouble

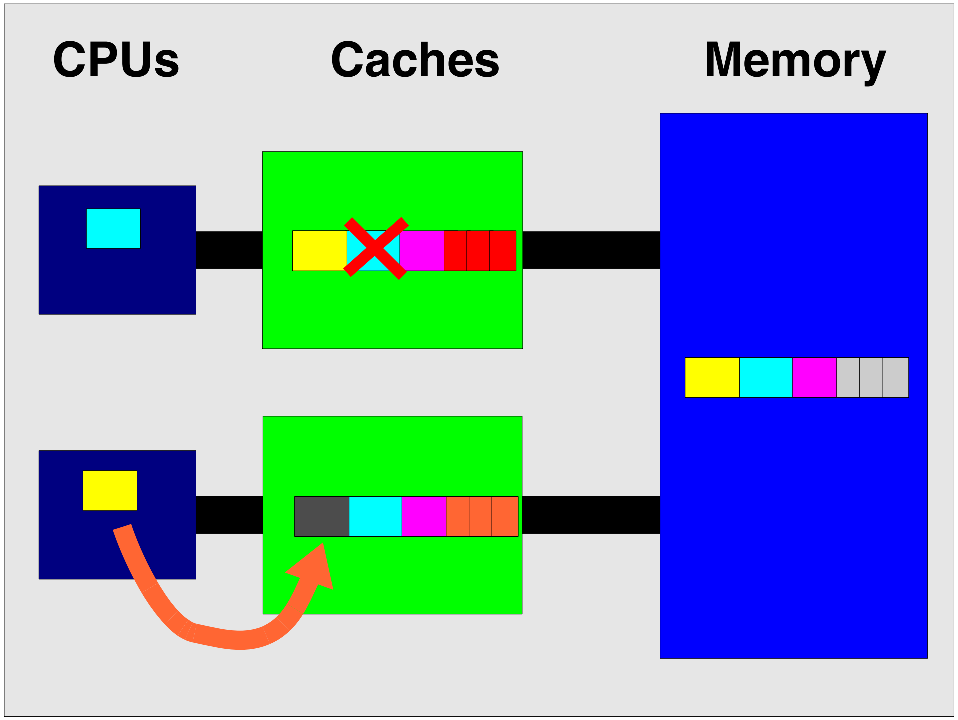
OpenMP and Performance

- ***The transparency of OpenMP is a mixed blessing***
  - ***Makes things pretty easy***
  - ***May mask performance bottlenecks***
- ***In the ideal world, an OpenMP application just performs well***
- ***Unfortunately, this is not the case***
- ***Two of the more obscure effects that can negatively impact performance are cc-NUMA behavior and False Sharing***
- ***Neither of these are restricted to OpenMP, but they are important enough to cover in some detail here***

False Sharing

False Sharing

A store into a shared cache line invalidates the other copies of that line:



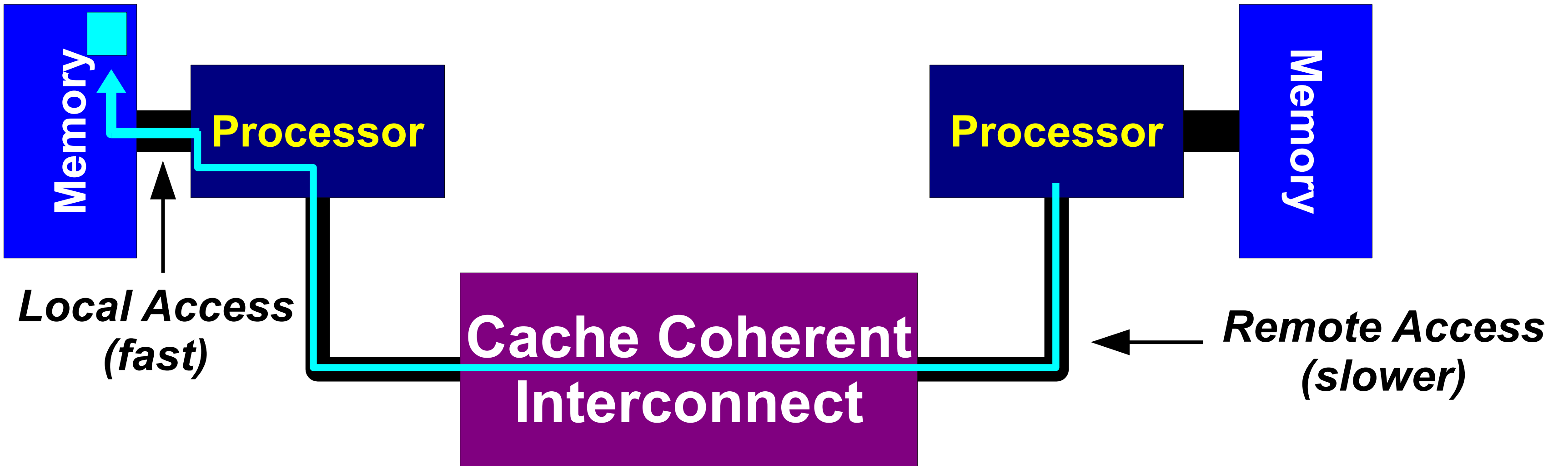
The system is not able to distinguish between changes within one individual line

False Sharing Red Flags

- ***Be alert, when all of these three conditions are met:***
  - ***Shared data is modif ed by multiple processors***
  - ***Multiple threads operate on the same cache line(s)***
  - ***Update occurs simultaneously and very frequently***
- ***Use local data where possible***
- ***Shared read-only data does not lead to false sharing***

Considerations for cc-NUMA

A generic cc-NUMA architecture

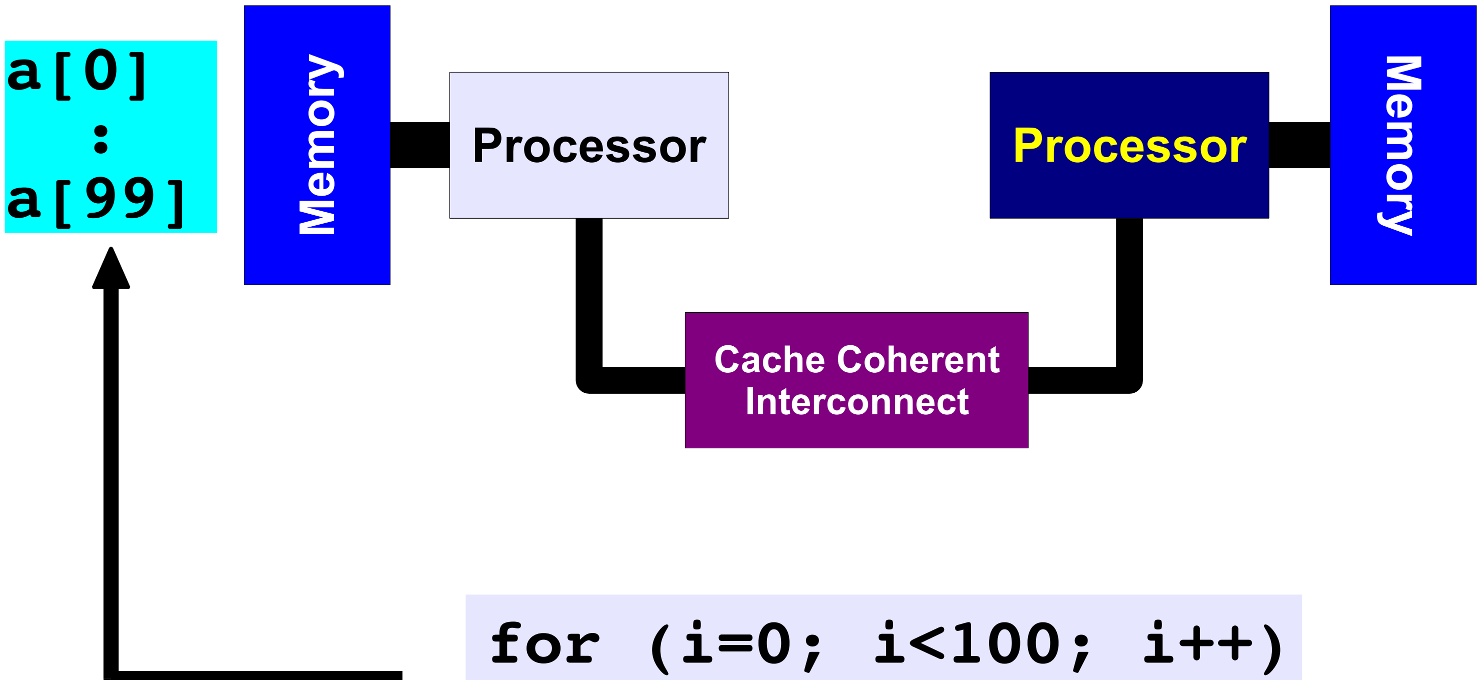


Main Issue: How To Distribute The Data ?

About Data Distribution

- ***Important aspect on a cc-NUMA system***
  - ***If not optimal - longer access times, memory hotspots***
- ***OpenMP does not provide support for cc-NUMA***
- ***Placement comes from the Operating System***
  - ***This is therefore Operating System dependent***
- ***Solaris, Linux and Windows use "First Touch" to place data***

About “First Touch” placement/1

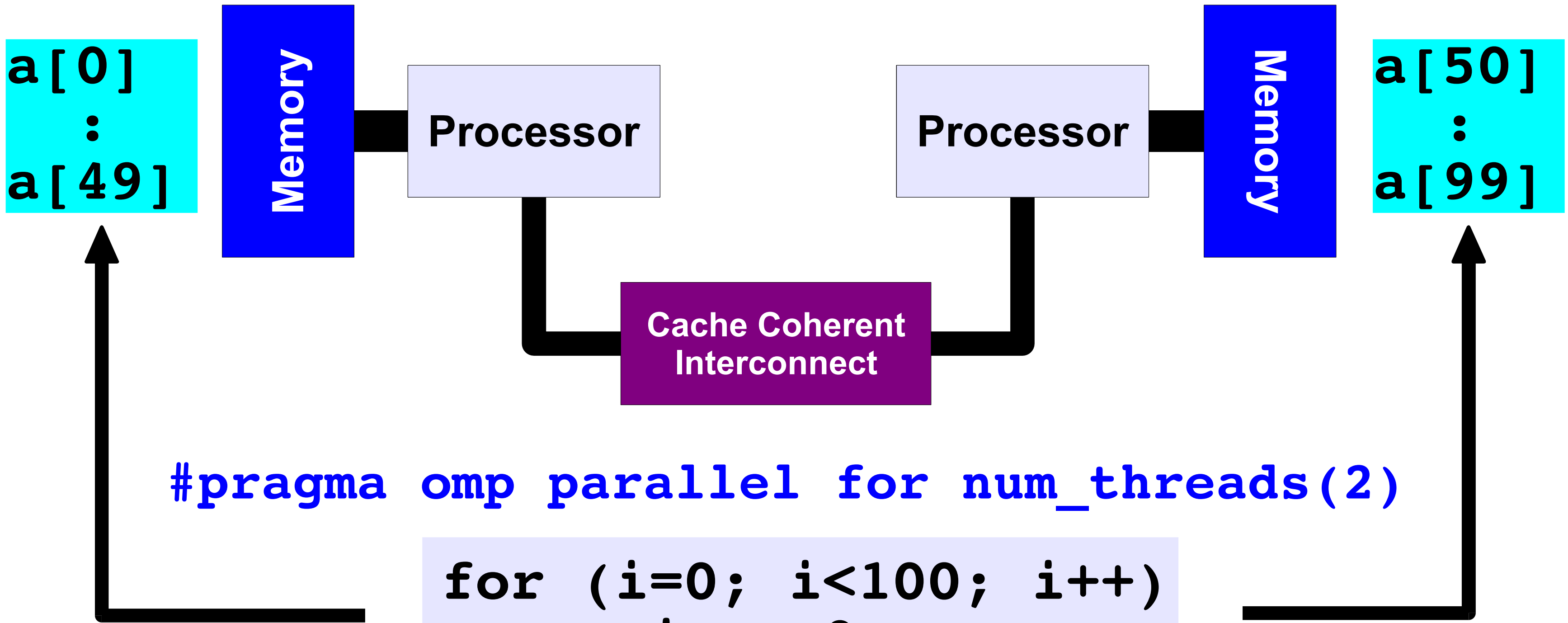



```
for (i=0; i<100; i++)
      a[i] = 0;
```

First Touch

All array elements are in the memory of the processor executing this thread

About “First Touch” placement/2




```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
     a[i] = 0;
```

First Touch
Both memories each have “their half” of the array

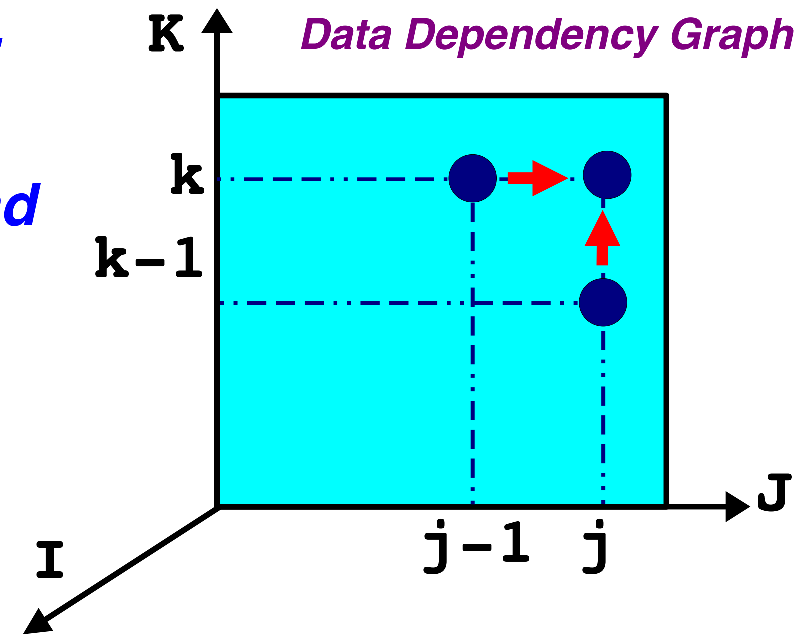
Get Real

Block Matrix Update

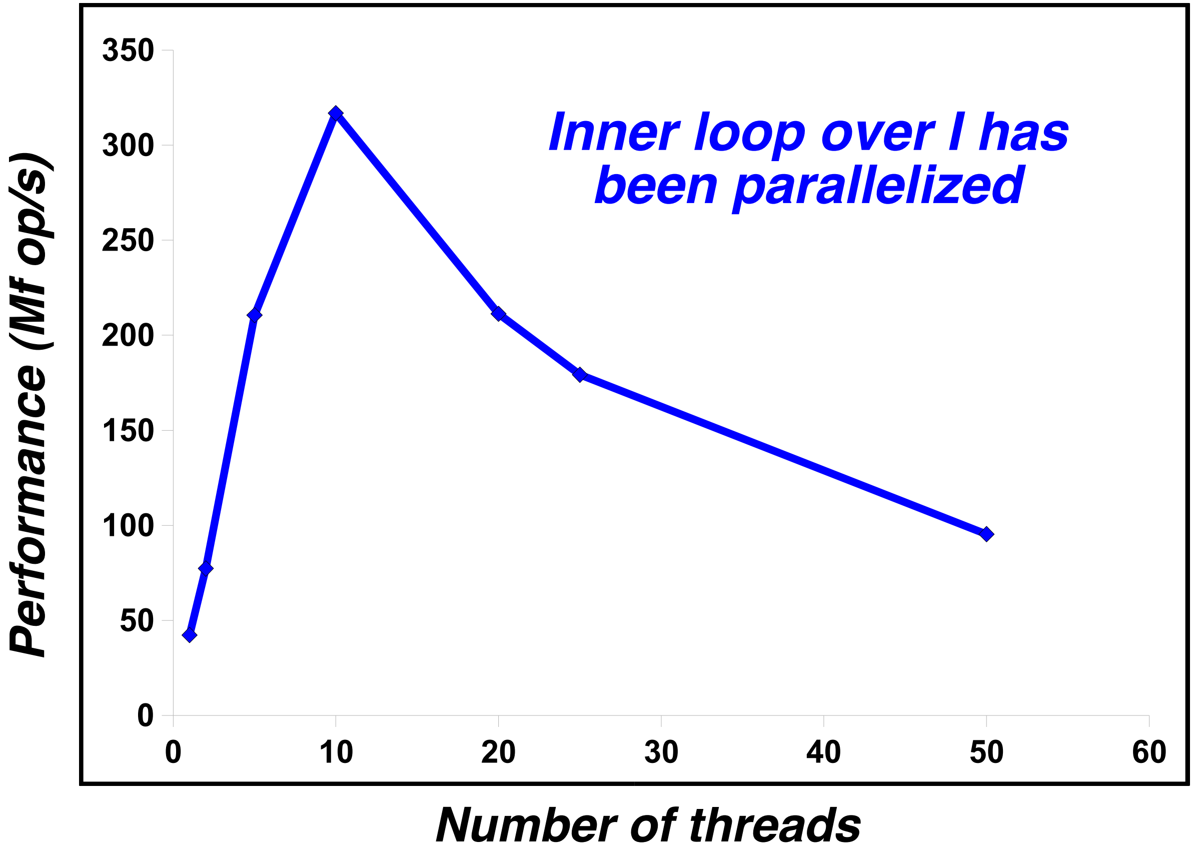
A 3D matrix update

```
        do k = 2, n
           do j = 2, n
!$omp parallel do default(shared) private(i) &
!$omp schedule(static)
              do i = 1, m
                 x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
              end do
!$omp end parallel do
           end do
        end do
```

- ***The loops are correctly nested for serial performance***
- ***Due to a data dependency on J and K, only the inner loop can be parallelized***
- ***This will cause the barrier to be executed (N-1)*** ^{2} ***times***



The performance



Scaling is very poor (as to be expected)

Dimensions : M=7,500 N=20
Footprint : ~24 MByte

Performance Analyzer data

Using 10 threads

Name	Excl. User CPU sec.	Excl. User CPU %	Incl. User CPU sec.	Excl. Wall sec.
<Total>	10.590	100.0	10.590	1.550
__mt_EndOfTask_Barrier_	5.740	54.2	5.740	0.240
__mt_WaitForWork_	3.860	36.4	3.860	0.
__mt_MasterFunction_	0.480	4.5	0.680	0.480
MAIN_	0.230	2.2	1.200	0.470
block_3d_ -- MP doall from line 14 [_$d1A14 … 3d_]	0.170	1.6	5.910	0.170
block_3d_	0.040	0.4	6.460	0.040
memset	0.030	0.3	0.030	0.080

Using 20 threads

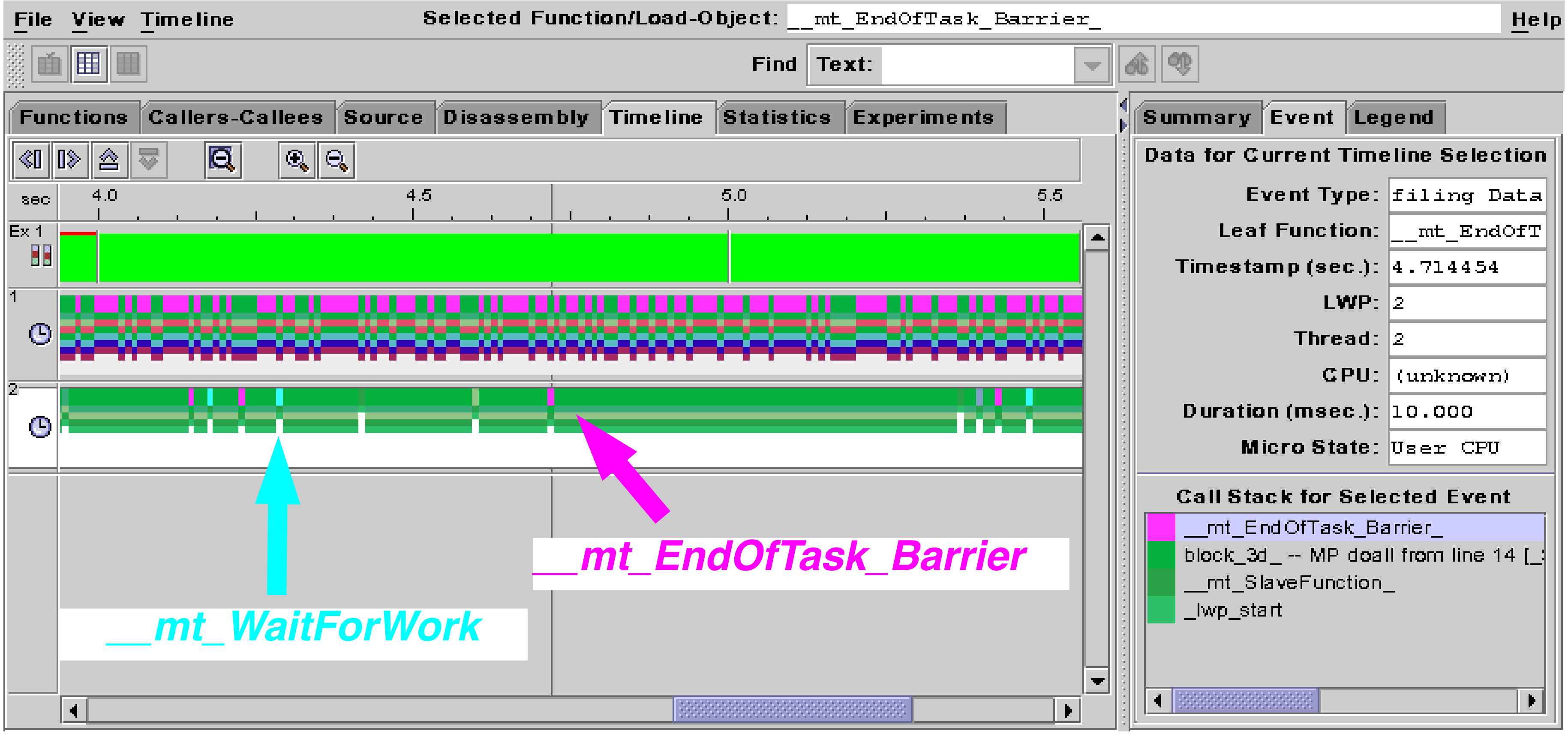
Name	Excl. User CPU sec.	Excl. User CPU %	Incl. User CPU sec.	Excl. Wall sec.
<Total>	47.120	100.0	47.120	2.900
__mt_EndOfTask_Barrier_	25.700	54.5	25.700	0.980
__mt_WaitForWork_	19.880	42.2	19.880	0.
__mt_MasterFunction_	1.100	2.3	1.320	1.100
MAIN_	0.190	0.4	2.520	0.470
block_3d_ -- MP doall from line 14 [_$d1A14.block_3d_]	0.100	0.2	25.800	0.100
__mt_setup_doJob_int_	0.080	0.2	0.080	0.080
__mt_setup_job_	0.020	0.0	0.020	0.020
block_3d_	0.010	0.0	27.020	0.010

do not scale at all

scales somewhat

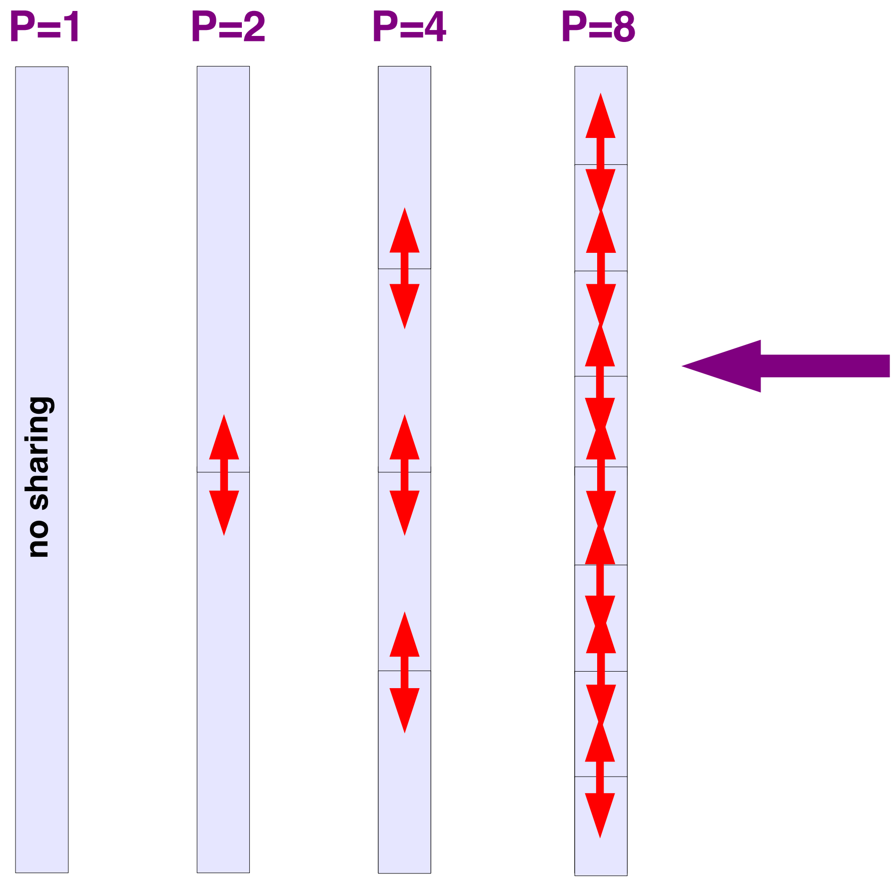
Question: Why is __mt_WaitForWork so high in the prof le ?

The Analyzer Timeline overview



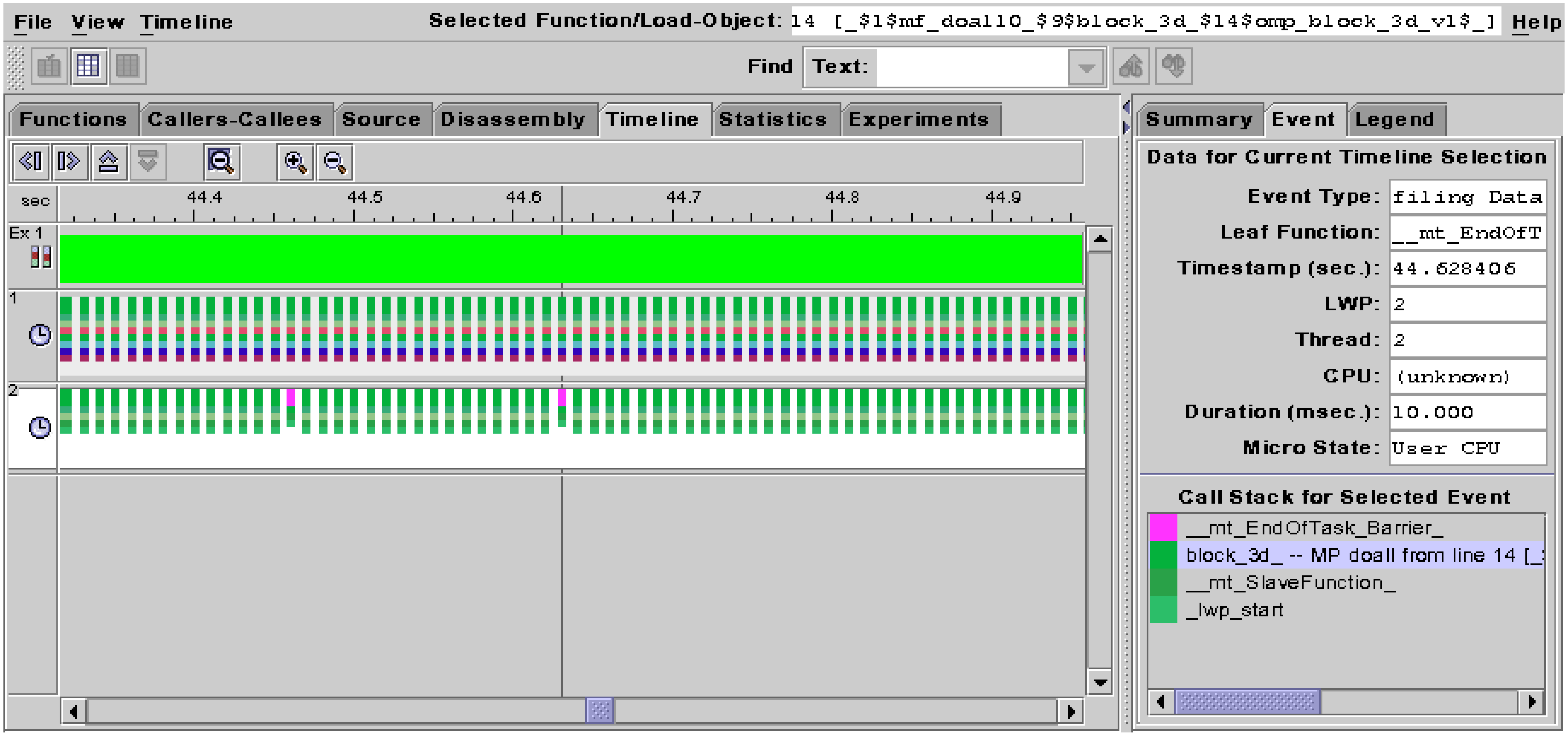
This is False Sharing at work !

```
!$omp parallel do default(shared) private(i) &
!$omp schedule(static)
          do i = 1, m
             x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
          end do
!$omp end parallel do
```



False sharing increases as we increase the number of threads

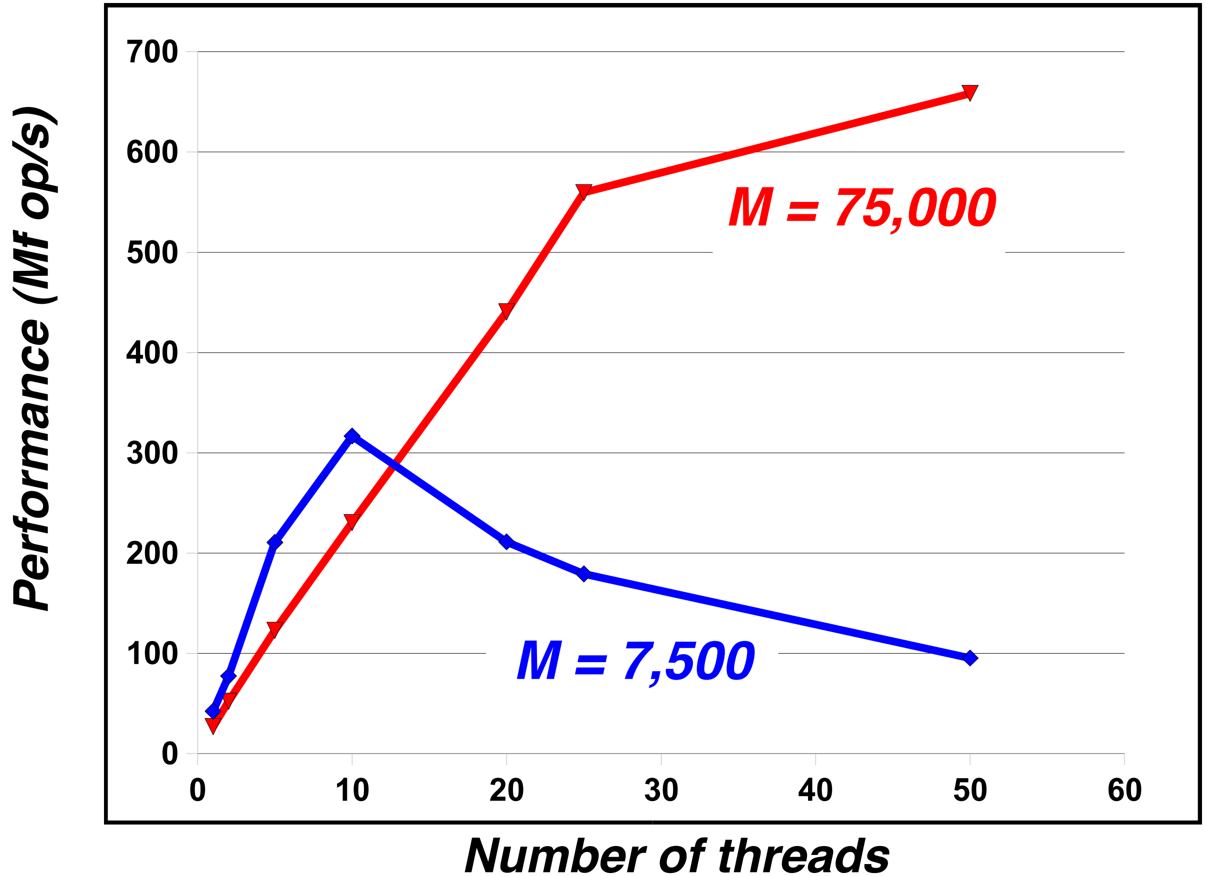
Sanity Check: Setting M=75000*



Only a very few barrier calls now

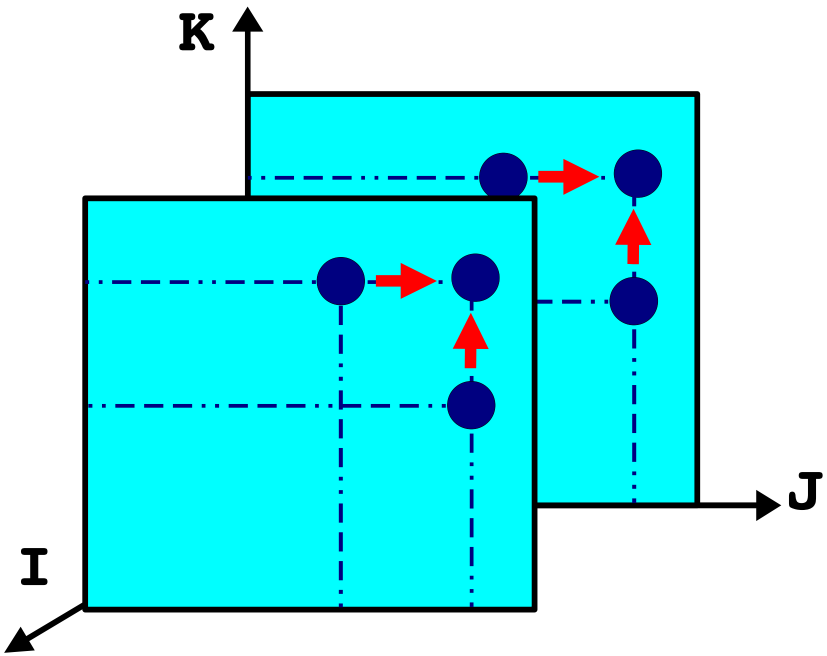
****) Increasing the length of the loop should decrease false sharing***

Performance comparison



For a higher value of M, the program scales better

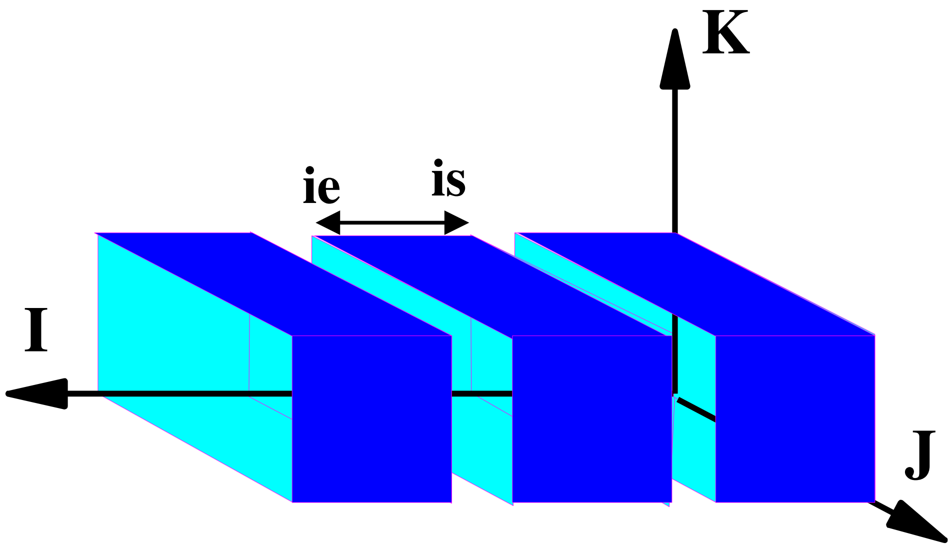
Observation



- *No data dependency on 'I'*
- *Therefore we can split the 3D matrix in larger blocks and process these in parallel*

```
do k = 2, n
   do j = 2, n
      do i = 1, m
         x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
   end do
end do
```

The Idea



- *We need to distribute the M iterations over the number of processors*
- *We do this by controlling the start (IS) and end (IE) value of the inner loop*
- *Each thread will calculate these values for it's portion of the work*

```
do k = 2, n
   do j = 2, n
      do i = is, ie
         x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
   end do
end do
```

The f rst implementation

```
      use omp_lib
          ........
      nrem   = mod(m,nthreads)
      nchunk =  (m-nrem)/nthreads

!$omp parallel default (none)&
!$omp private (P,is,ie)      &
!$omp shared  (nrem,nchunk,m,n,x,scale)

       P = omp_get_thread_num()

       if  ( P < nrem )  then
         is = 1 + P*(nchunk + 1)
         ie = is + nchunk
       else
         is = 1 + P*nchunk+ nrem
         ie = is + nchunk - 1
       end if

       call kernel(is,ie,m,n,x,scale)

!$omp end parallel
```

```
subroutine kernel(is,ie,m,n,x,scale)
         .......
do k = 2, n
 do j = 2, n
    do i = is, ie
      x(i,j,k)=x(i,j,k-1)+x(i,j-1,k)*scale
    end do
  end do
end do
```

Another Idea: Use OpenMP !

```
      use omp_lib

      implicit none
      integer      :: is, ie, m, n
      real(kind=8):: x(m,n,n), scale
      integer      :: i, j, k

!$omp parallel default(none) &
!$omp private(i,j,k) shared(m,n,scale,x)
      do k = 2, n
         do j = 2, n
!$omp do schedule(static)
            do i = 1, m
               x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
            end do
!$omp end do nowait
         end do
      end do
!$omp end parallel
```

How this works on 2 threads

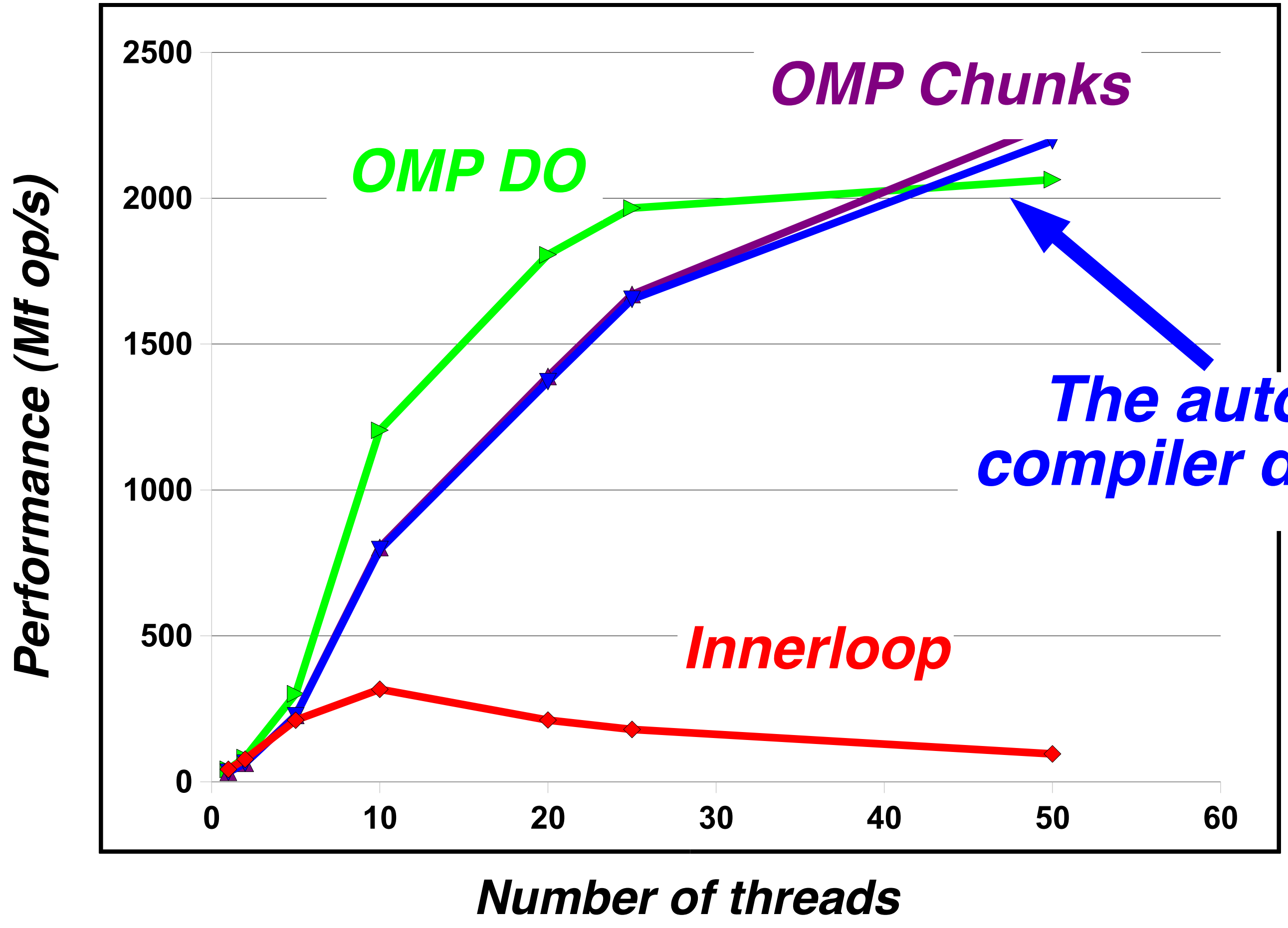
Thread 0 Executes:		Thread 1 Executes:
k=2 j=2	parallel region	k=2 j=2
do i = 1,m/2 x(i,2,2) = ... end do	work sharing	do i = m/2+1,m x(i,2,2) = ... end do
k=2 j=3	parallel region	k=2 j=3
do i = 1,m/2 x(i,3,2) = ... end do	work sharing	do i = m/2+1,m x(i,3,2) = ... end do
... etc ...		... etc ...

This splits the operation in a way that is similar to our manual implementation

Performance

- ❑ *We have set M=7500 N=20*
  - *This problem size does not scale at all when we explicitly parallelized the inner loop over 'I'*
- ❑ *We have have tested 4 versions of this program*
  - *Inner Loop Over 'I' - Our f rst OpenMP version*
  - *AutoPar - The automatically parallelized version of 'kernel'*
  - *OMP_Chunks - The manually parallelized version with our explicit calculation of the chunks*
  - *OMP_DO - The version with the OpenMP parallel region and work-sharing DO*

The performance (M=7,500)



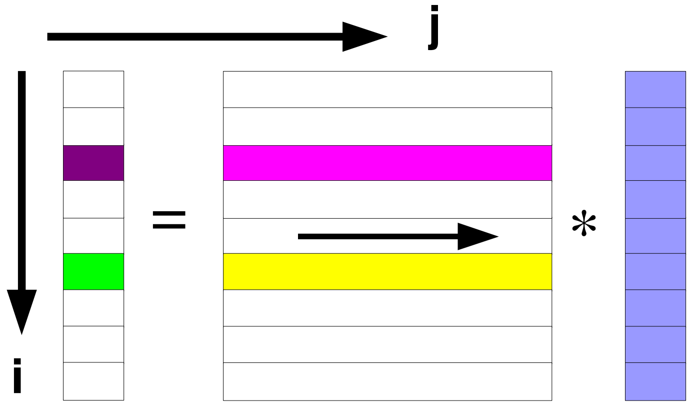
The auto-parallelizing compiler does really well !

Dimensions : M=7,500 N=20
Footprint : ~24 MByte

Matrix Times Vector

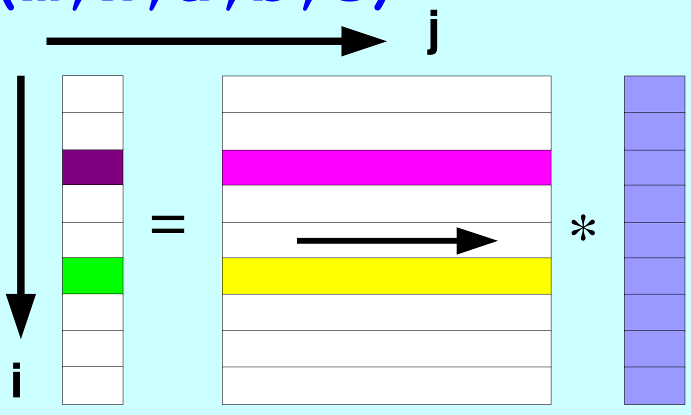
The Sequential Source

```
for (i=0; i<m; i++)
{
   a[i] = 0.0;
   for (j=0; j<n; j++)
      a[i] += b[i][j]*c[j];
}
```

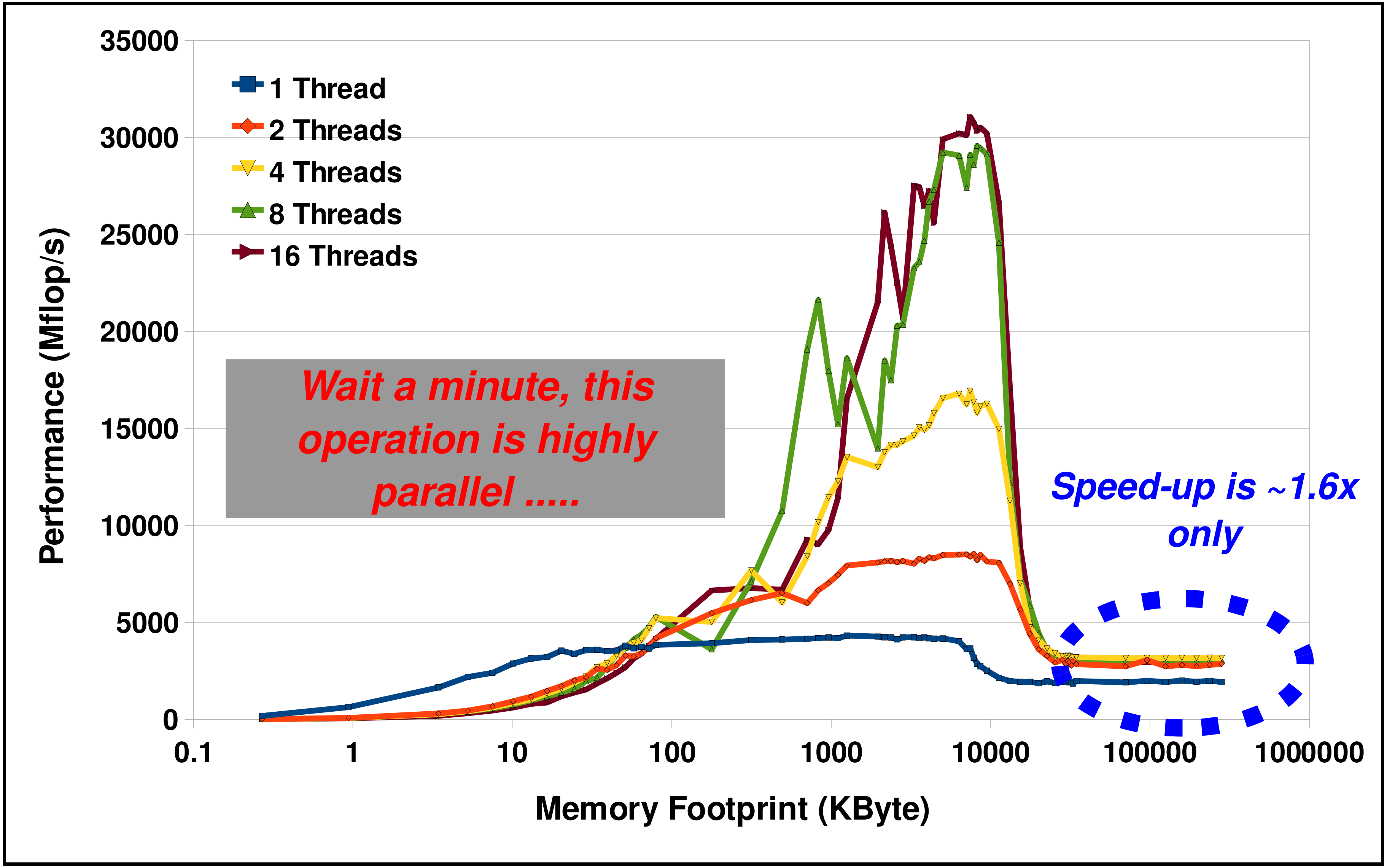


The OpenMP Source

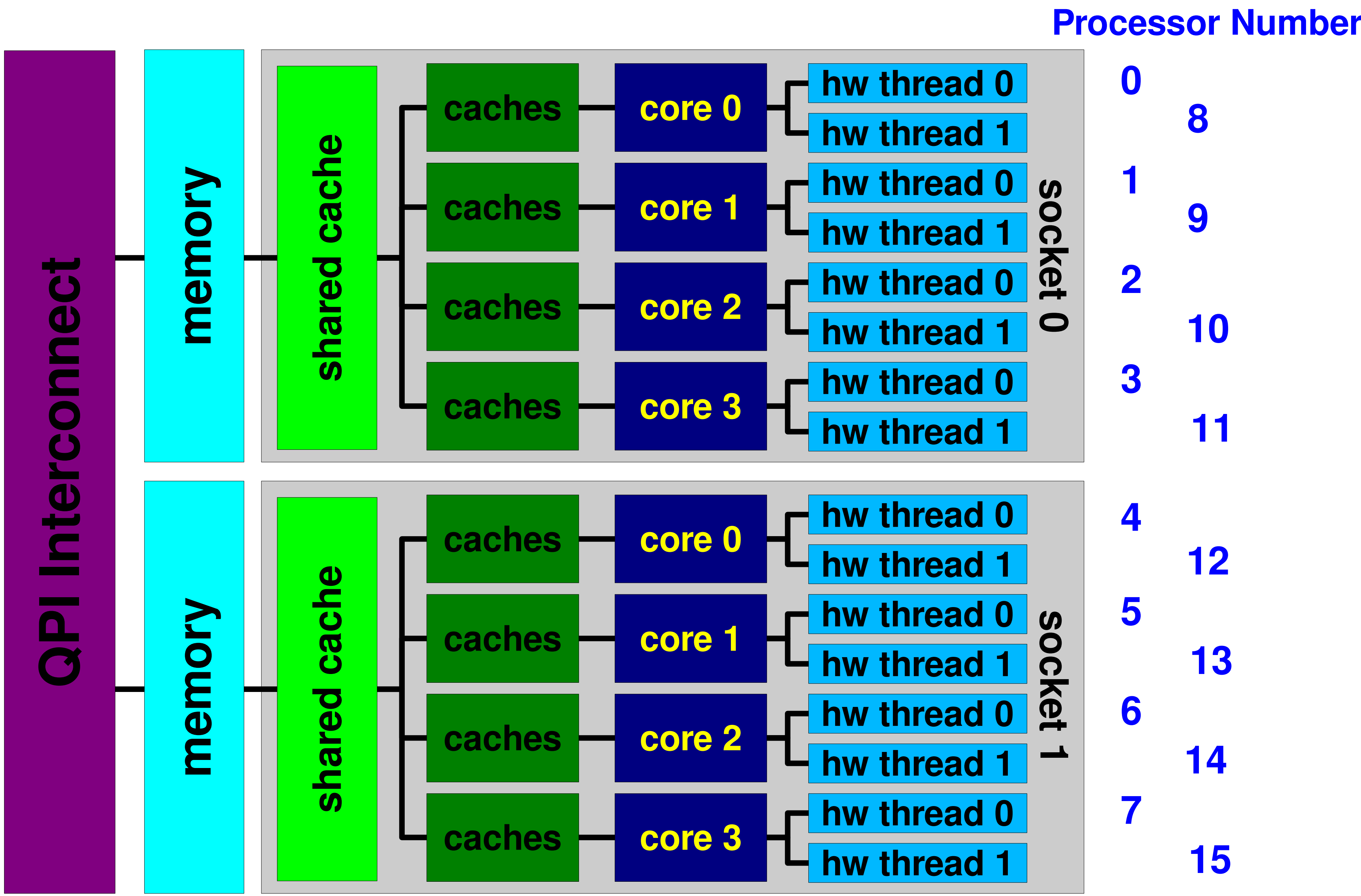
```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
 }
```



Performance - 2 Socket Nehalem

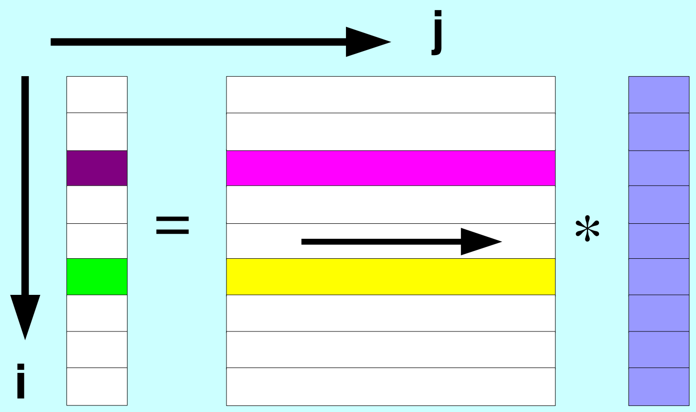


A Two Socket Nehalem System

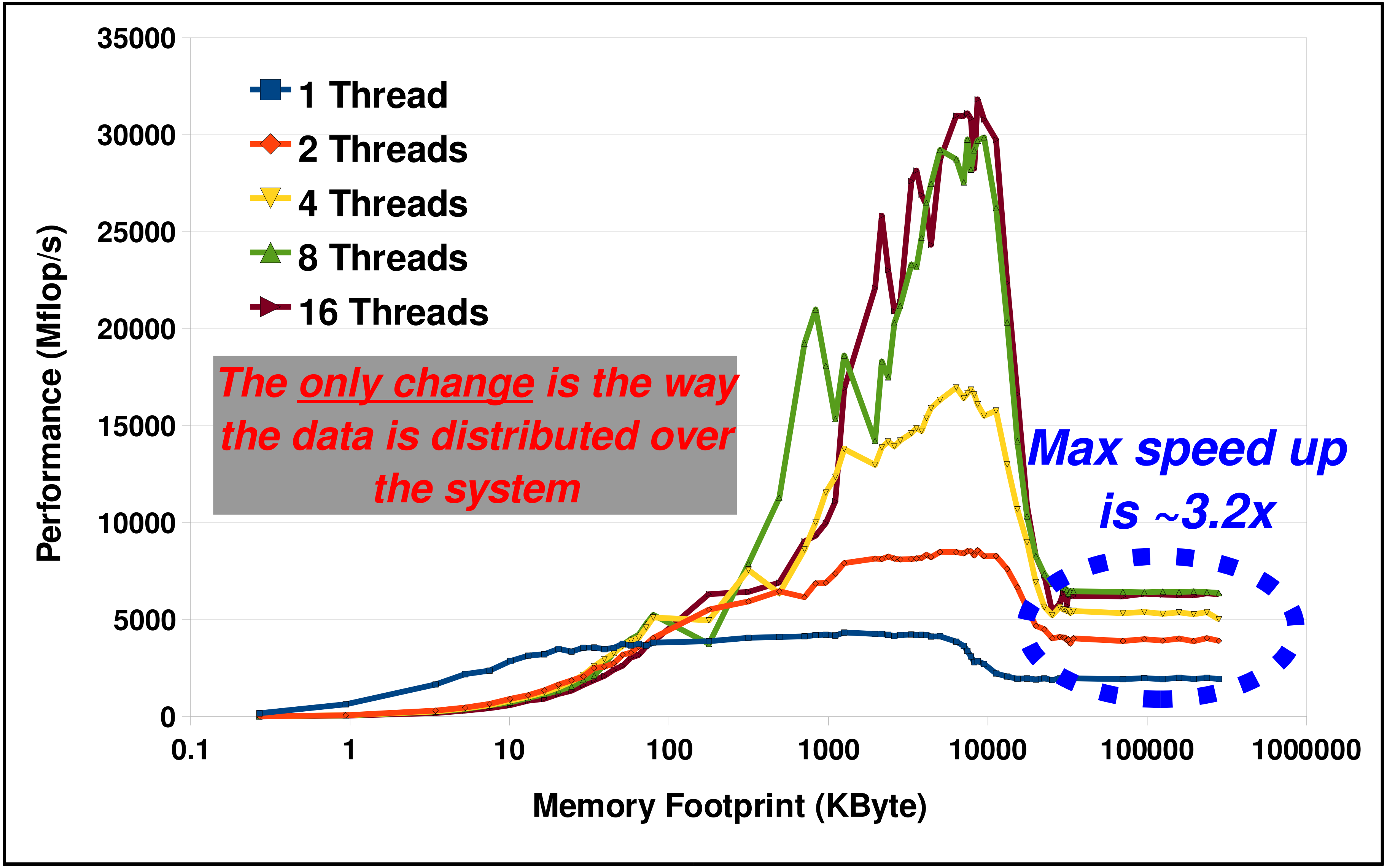


Data Initialization

```
#pragma omp parallel default(none) \
        shared(m,n,a,b,c) private(i,j)
{
#pragma omp for
   for (j=0; j<n; j++)
      c[j] = 1.0;

#pragma omp for
   for (i=0; i<m; i++)
   {
      a[i] = -1957.0;
      for (j=0; j<n; j++)
         b[i[]j] = i;
   } /*-- End of omp for --*/

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



Exploit First Touch



Summary Case Studies

- *There are several important basic aspects to consider when it comes to writing an eff cient OpenMP program*
- *Moreover, there are also obscure additional aspects:*
  - *cc-NUMA*
  - *False Sharing*
- *Key problem is that most developers are not aware of these rules and .... blaming OpenMP is all that easy*
  - *In some cases it is a trade-off between ease of use and performance*
  - *OpenMP typically goes for the former, but .....*
    - ✔ *With some extra effort can be made to scale well in many cases*

The Wrapping

Wrapping Things Up

"While we're still waiting for your MPI debug run to finish, I want to ask you whether you found my information useful."

"Yes, it is overwhelming. I know."

"And OpenMP is somewhat obscure in certain areas. I know that as well."

"I understand. You're not a Computer Scientist and just need to get your scientific research done."

"I agree this is not a good situation, but it is all about Darwin, you know. I'm sorry, it is a tough world out there."

It Never Ends

"Oh, your MPI job just finished! Great."

"Your program does not write a file called 'core' and it wasn't there when you started the program?"

"You wonder where such a file comes from? Let's get a big and strong coffee first."

That's It

Thank You and ..... Stay Tuned !

Ruud van der Pas
ruud.vanderpas@sun.com