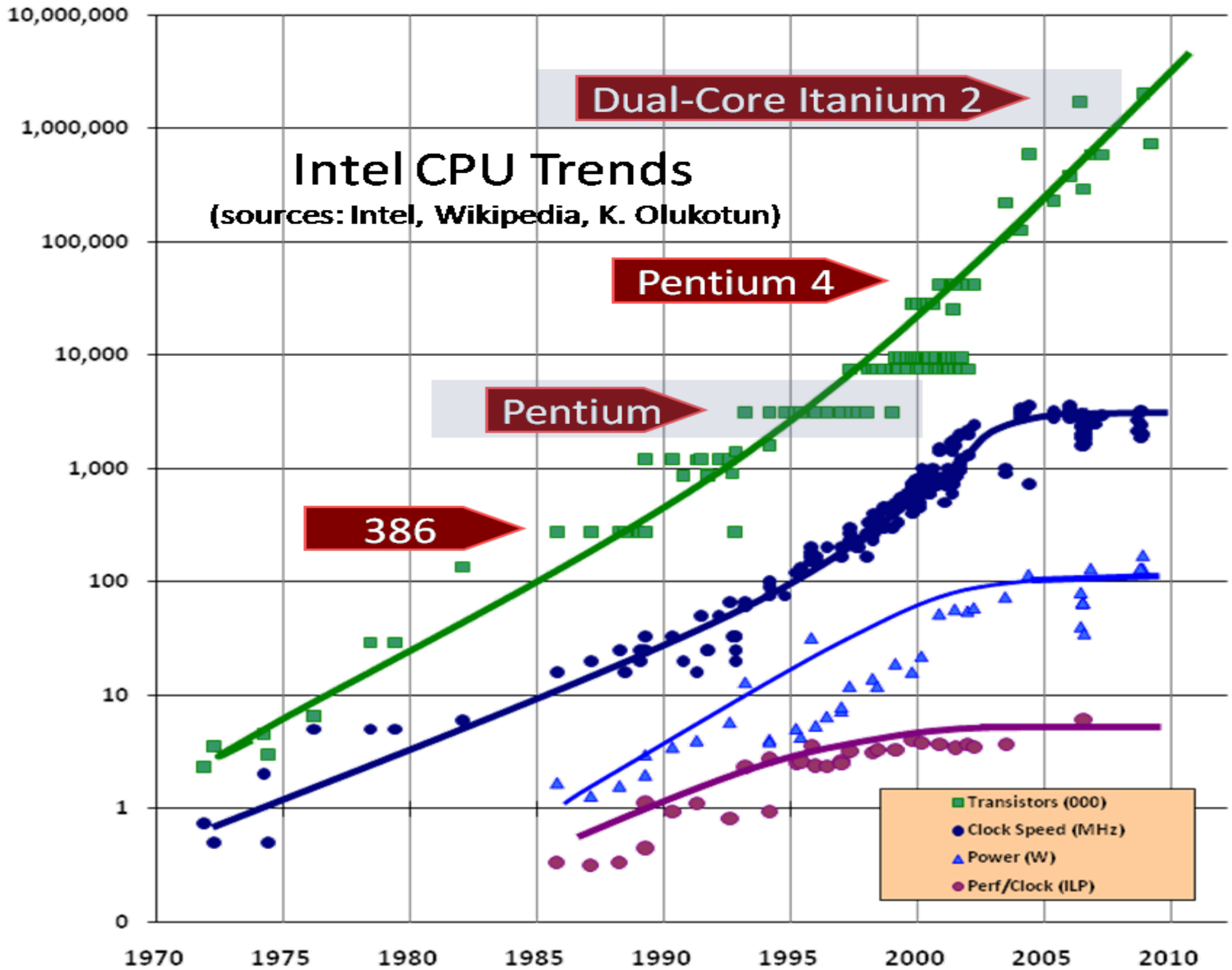
Introduction to OpenMP and Threaded Libraries

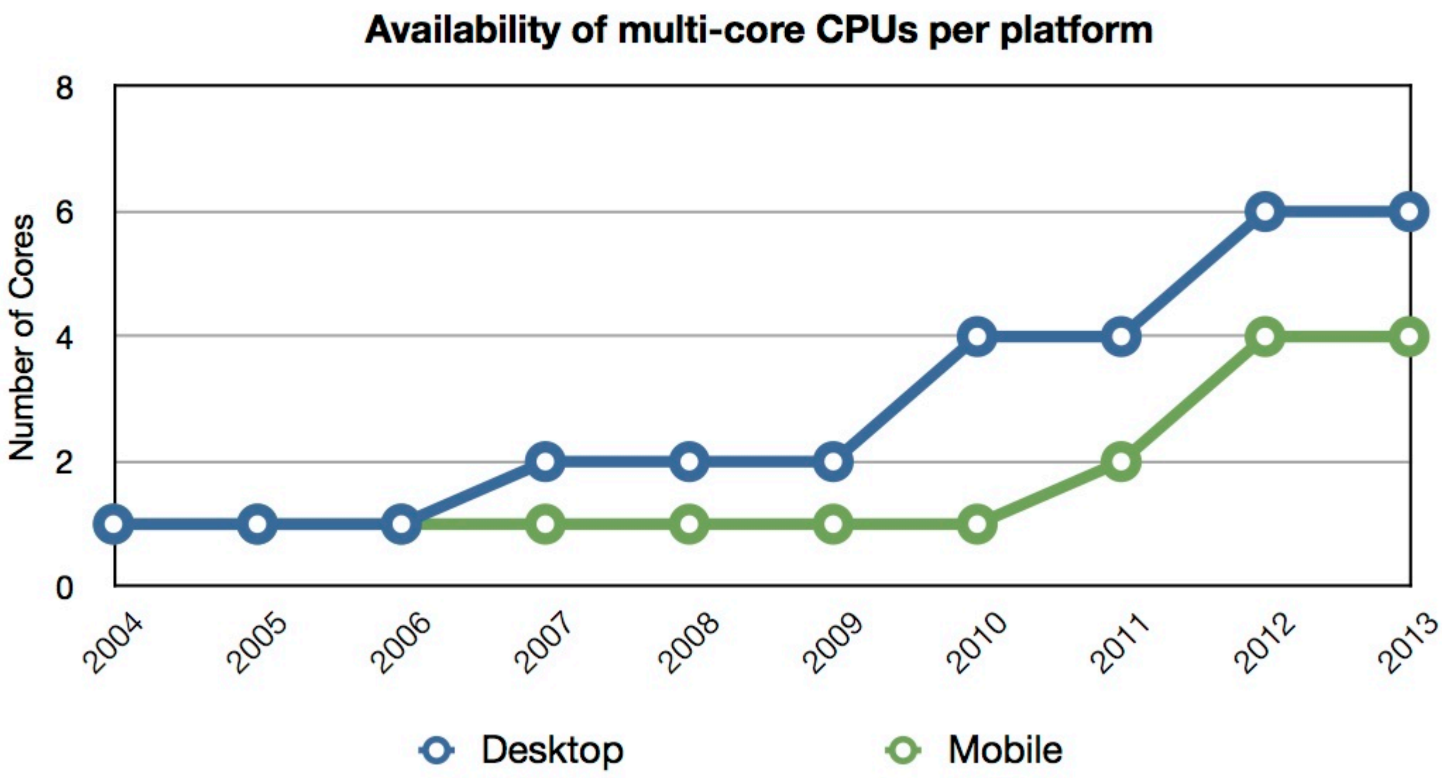
Ivan Girotto – igirotto@ictp.it
Information & Communication Technology Section (ICTS)
International Centre for Theoretical Physics (ICTP)

OUTLINE

- Shared Memory Architectures
- Thinking Parallel
- Threaded Libraries
- The OpenMP Programming Paradigm
- Hands-on

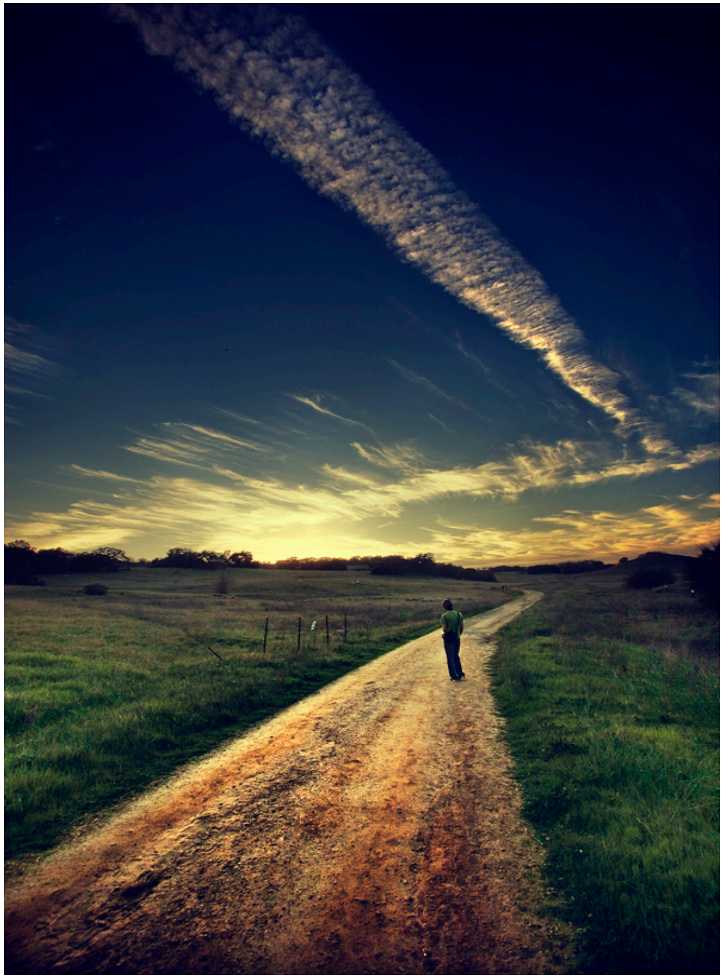
SHARED MEMORY ARCHITECTURES


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

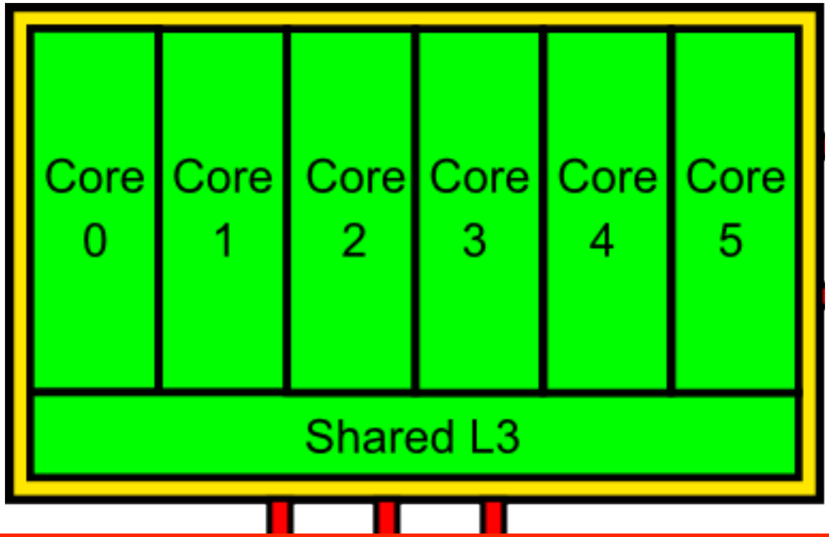

Availability of multi-core CPUs per platform
Number of Cores
8
6
4
2
0
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
Desktop
Mobile

Consequences

- Parallelism is no longer an option for only either larger scale problems or improve the time of response
- It is inescapable to exploit current & next generations of compute processors



Representation of Multi-cores system

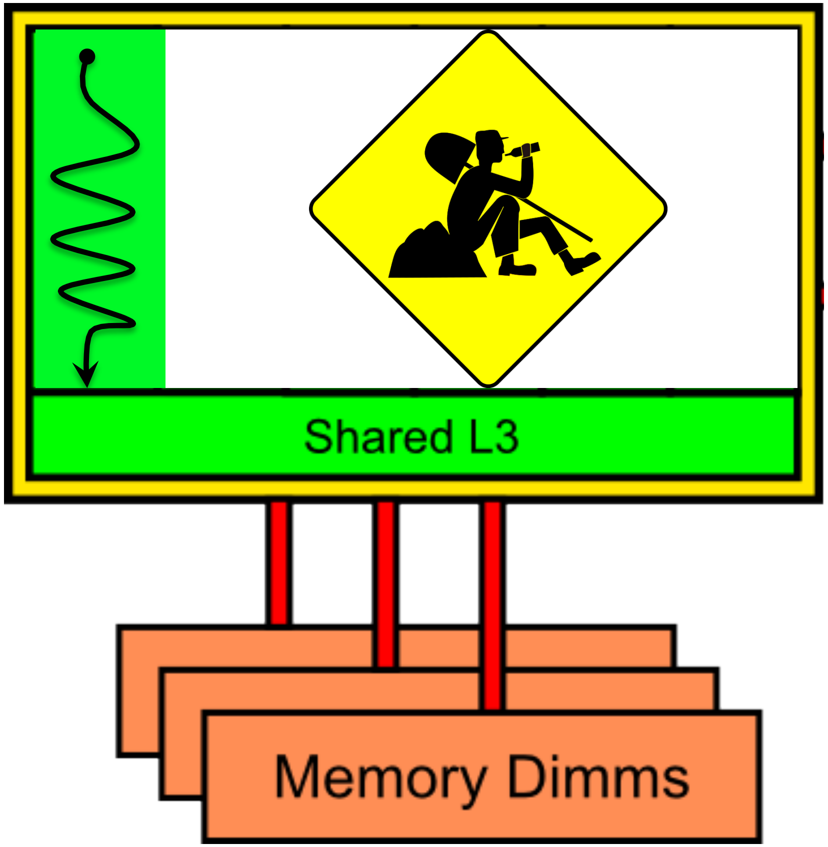


Xeon E5650 hex-core processors

Main Memory

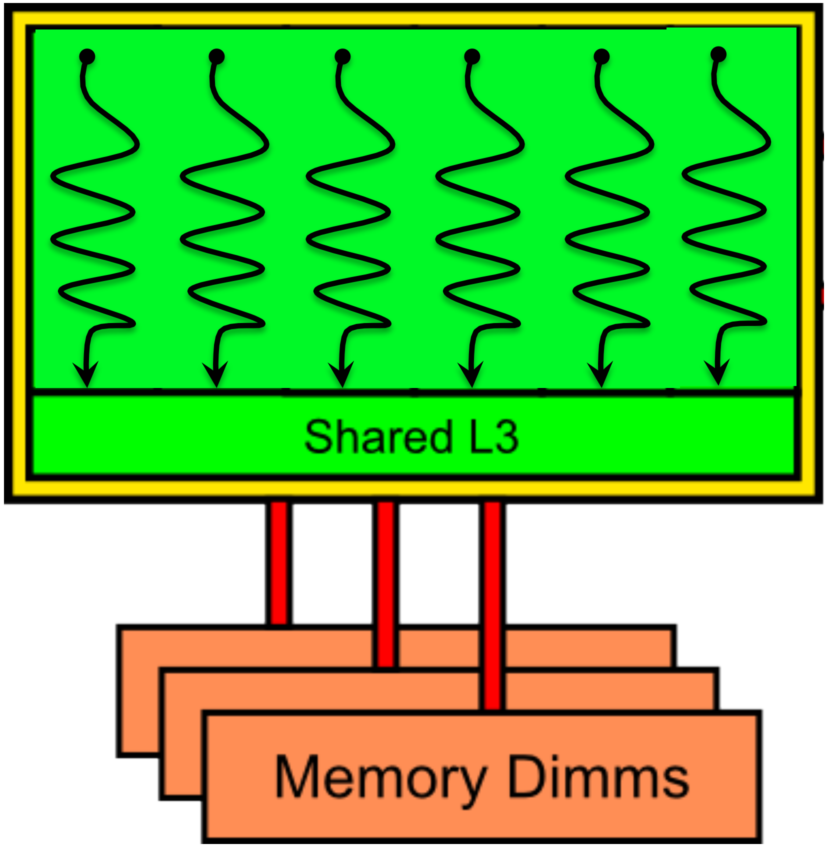
Dual Socket (Westmere) - 24GB RAM

Multi-core system Vs Serial Programming



Xeon E5650 hex-core processors (12GB - RAM)

Multi-core system Vs // Programming



Xeon E5650 hex-core processors (12GB - RAM)

THINKING IN PARALLEL

Design of Parallel Algorithm /1

- A serial algorithm is a sequence of basic steps for solving a given problem using a single serial computer
- Similarly, a parallel algorithm is a set of instruction that describe how to solve a given problem using multiple (>=1) parallel processors
- The parallelism add the dimension of concurrency. Designer must define a set of steps that can be executed simultaneously!!!

Design of Parallel Algorithm /2

- Identify portions of the work that can be performed concurrently
- Mapping the concurrent pieces of work onto multiple processes running in parallel
- Distributing the input, output and intermediate data associated within the program
- Managing accesses to data shared by multiple processors
- Synchronizing the processors at various stages of the parallel program execution

Type of Parallelism

- **Functional (or task) parallelism**: different people are performing different task at the same time

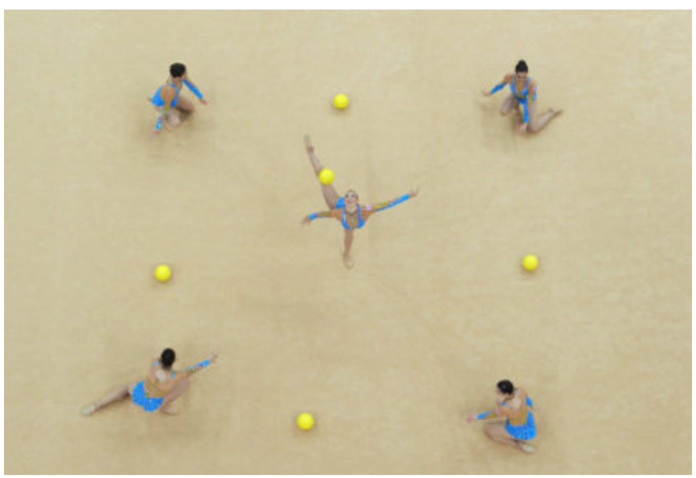
- **Data Parallelism**: different people are performing the same task, but on different equivalent and independent objects

Task/Process Mapping

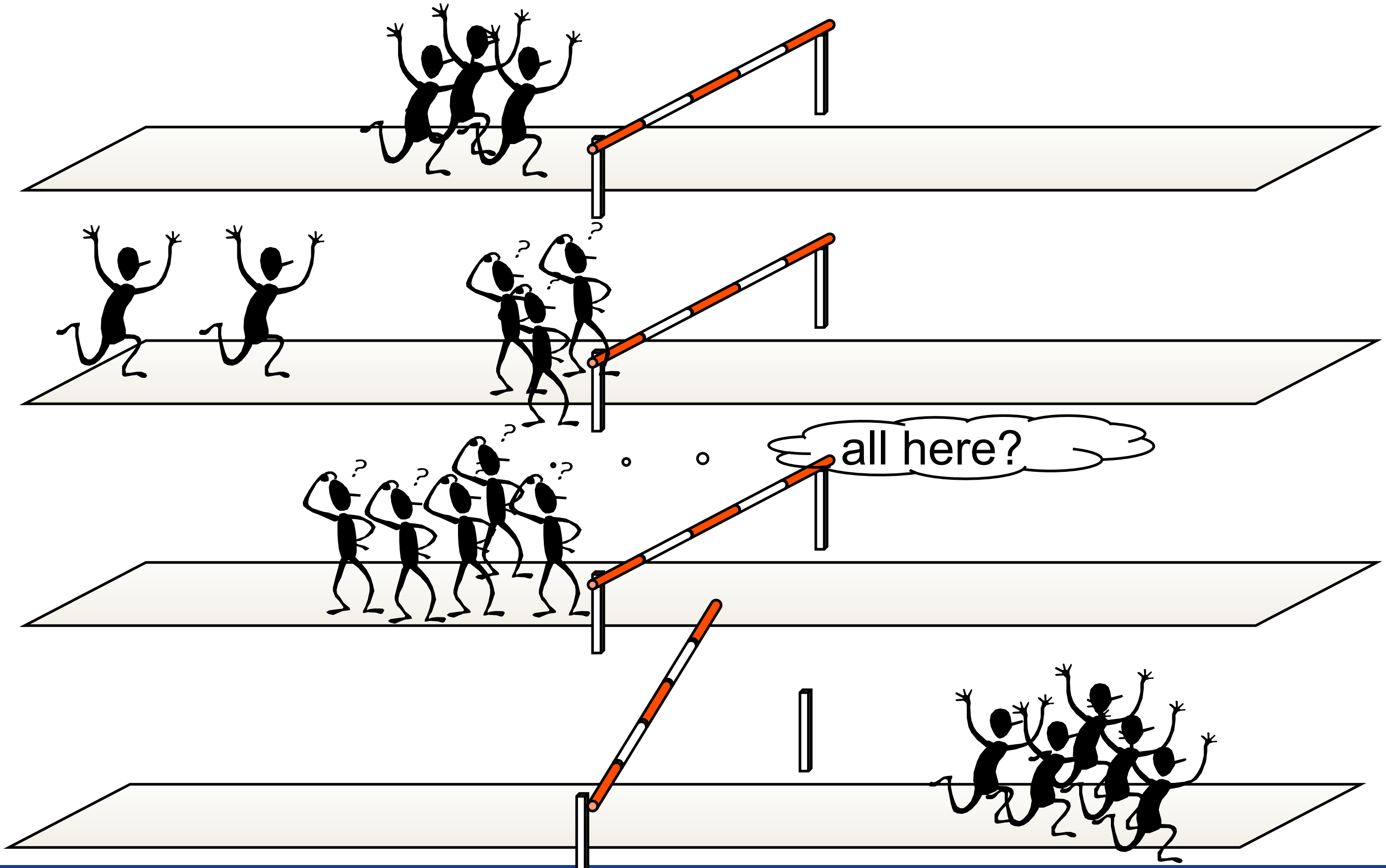
- The tasks, into which a problem is decomposed, are performed on physical processors
- The process is the computing agent that performs given tasks within a finite amount of time
- The mapping is the relation (N:N) by which tasks are assigned to processes for execution
- Mapping is one of the needed keys to find a decent load balancing. Mapping techniques are mainly:
  - Static Mapping: is defined prior to the execution (i.e., task distribution based on domain decomposition).
  - Dynamic Mapping: the work is distributed during the execution (i.e., master-slave model)

Process Interactions /1

- The effective speed-up obtained by the parallelization depend by the amount of overhead we introduce making the algorithm parallel
- There are mainly two key sources of overhead:
  1. Time spent in inter-process interactions (**communication**)
  2. Time some process may spent being idle (**synchronization**)



What happens if the girls are not well trained?



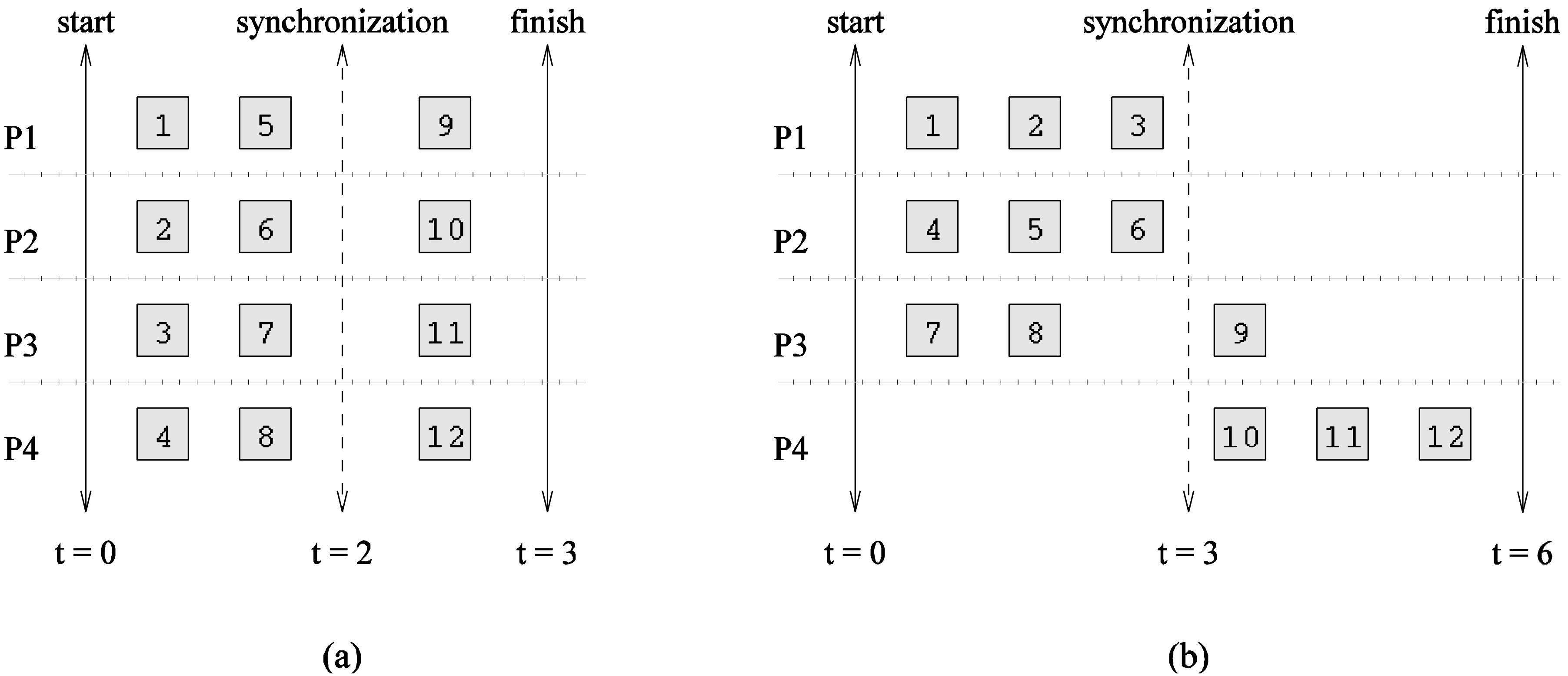
Process Interactions /2

Synchronization

+

Communication

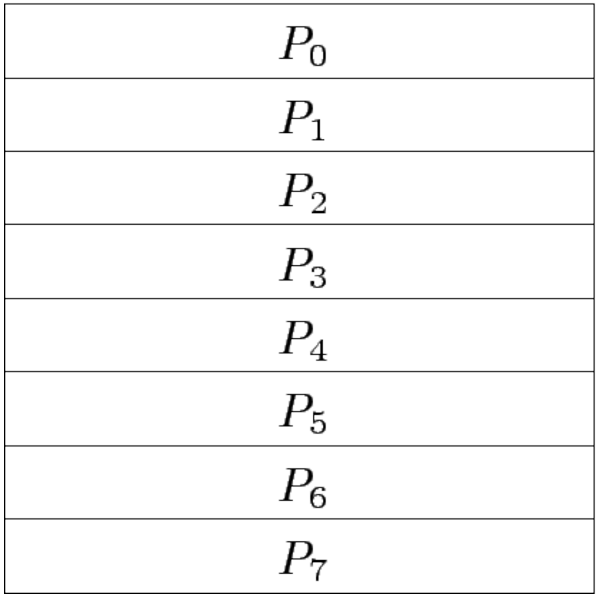
Mapping and Synchronization



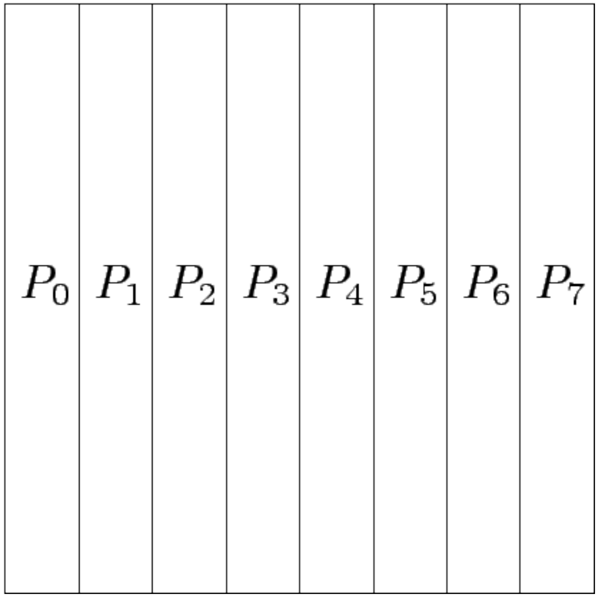
Static Data Partitioning

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution



column-wise distribution



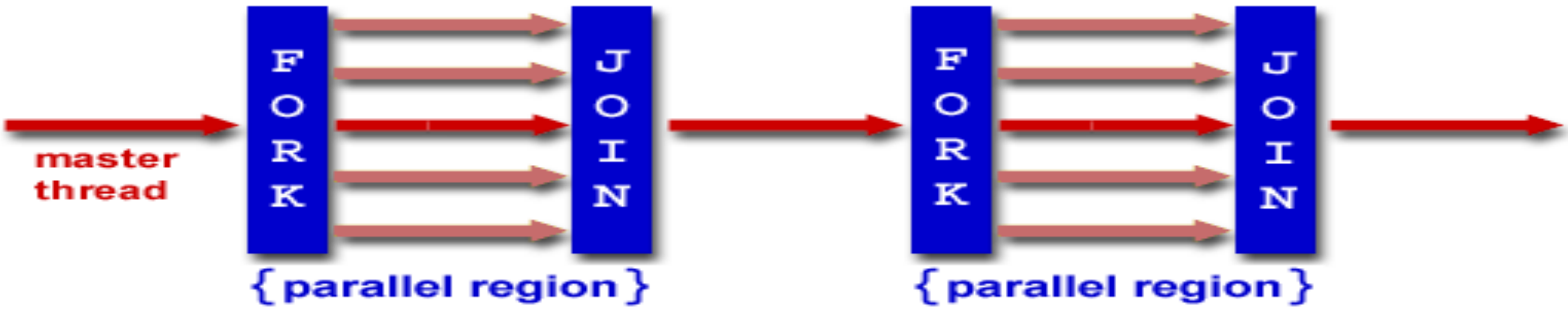
Granularity

- Granularity is determined by the decomposition level (number of task) on which we want divide the problem
- The degree to which task/data can be subdivided is limit to concurrency and parallel execution
- Parallelization has to become "topology aware"
  - coarse grain and fine grained parallelization has to be mapped to the topology to reduce memory and I/O contention

OpenMP™

OpenMP (*Open spec. for Multi Processing*)

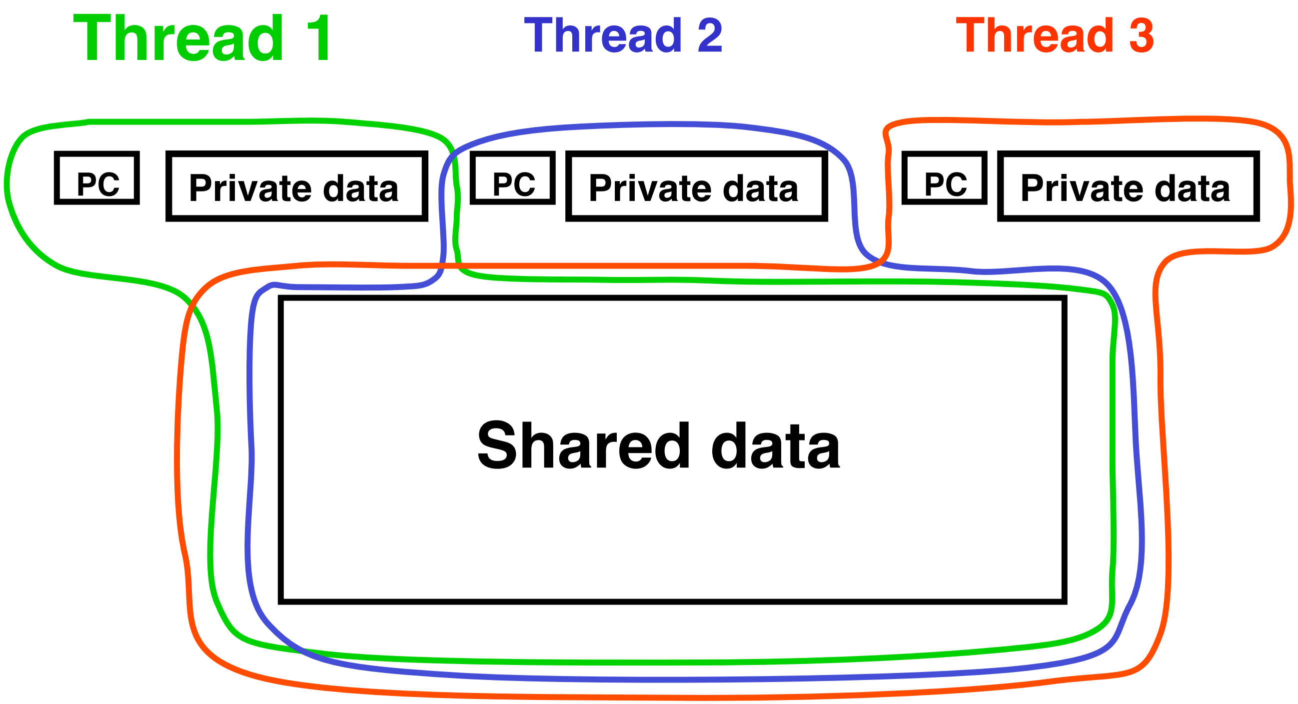
- OpenMP is not a computer language
  - Rather it works in conjunction with existing languages such as standard Fortran or C/C++
- Application Programming Interface (API)
  - that provides a portable model for shared memory // applications.
  - Three main components:
    - Compiler directives
    - Runtime library routines
    - Environment variables
- Three main advantages:
  - Incremental parallelization, Ease of use, Standardised



- Thread-based Parallelism
- Explicit Parallelism
- Fork-Join Model
- Compiler Directive Based
- Dynamic Threads

*Source: http://www.llnl.gov/computing/tutorials/openMP/#ProgrammingModel

Memory footprint



Multi-threading - Recap

- A thread is a (lightweight) process - an instance of a program + its data (private memory)
- Each thread can follow its own flow of control through a program.
- Threads can share data with other threads, but also have private data.
- Threads communicate with each other via the shared data.
- The *master thread* is responsible for co-ordinating the threads group

Getting Started with OpenMP

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Work sharing
  - Data Environment (scope)
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is esentially the same for both Fortran and C/C++

Directives Format

- A directive is a special line of source code with meaning only to certain compilers.
- A directive is distinguished by a sentinel at the start of the line.
- OpenMP sentinels are:
  - Fortran: **!$OMP** (or **C$OMP** or ***$OMP**)
  - C/C++: **#pragma omp**

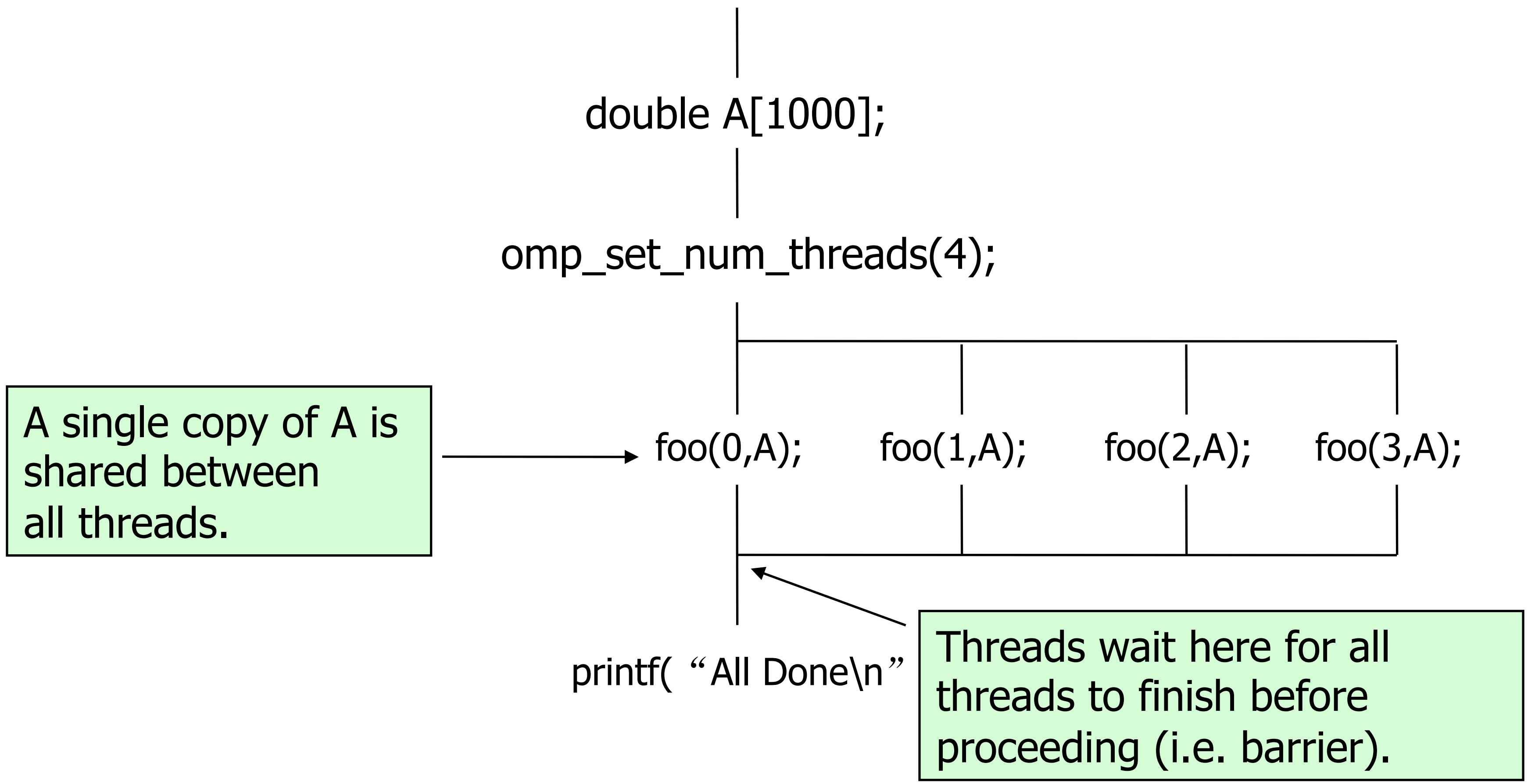
OpenMP: Parallel Regions

- For example, to create a 4-thread parallel region:
  - each thread calls foo(ID,A) for **ID** = **0** to **3**

Each thread redundantly executes the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID =omp_get_thread_num();
   foo(ID,A);
}
printf( "All Done\n" );
```

thread-safe routine: A routine that performs the intended function even when executed concurrently (by more than one thread)


double A[1000];
omp_set_num_threads(4);
A single copy of A is shared between all threads.
foo(0,A);
foo(1,A);
foo(2,A);
foo(3,A);
printf("All Done\n");
Threads wait here for all threads to finish before proceeding (i.e. barrier).

How many threads?

The number of threads in a parallel region is determined by the following factors:

- Use of the omp_set_num_threads() library function
- Setting of the OMP_NUM_THREADS environment variable
- The implementation default

Threads are numbered from 0 (master thread) to N-1.

OpenMP runtime library

OMP_GET_NUM_THREADS() – returns the current # of threads.

OMP_GET_THREAD_NUM() - returns the id of this thread.

OMP_SET_NUM_THREADS(n) – set the desired # of threads.

OMP_IN_PARALLEL() – returns .true. if inside parallel region.

OMP_GET_MAX_THREADS() - returns the # of possible threads.

Simple C OpenMP Program

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

int main ( ) {
	printf("Starting off in the sequential world.\n");
	#pragma omp parallel
	{

		printf("Hello from thread number %d\n", omp_get_thread_num() );
	}
	printf("Back to the sequential world.\n");
}
```

hands-on

HELLO WORLD AND COMPILING

Exploiting Loop Level Parallelism

Loop level Parallelism: parallelize only loops

- Easy to implement
- Highly readable code
- Less than optimal performance (sometimes)
- Most often used

Parallel Loop Directives

- Fortran do loop directive
  - !$omp do
- C\C++ for loop directive
  - #pragma omp for
- These directives do not create a team of threads but assume there has already been a team forked.
- If not inside a parallel region shortcuts can be used.
  - !$omp parallel do
  - #pragma omp parallel for

Parallel Loop Directives continued

- These are equivalent to a parallel construct followed immediately by a worksharing construct.

```
!$omp parallel do

      Same as

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

```
#pragma omp parallel for

      Same as

#pragma omp parallel
...
#pragma omp for
```

How is OpenMP Typically Used?

- OpenMP is usually used to parallelize loops:

Split-up this loop between multiple threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

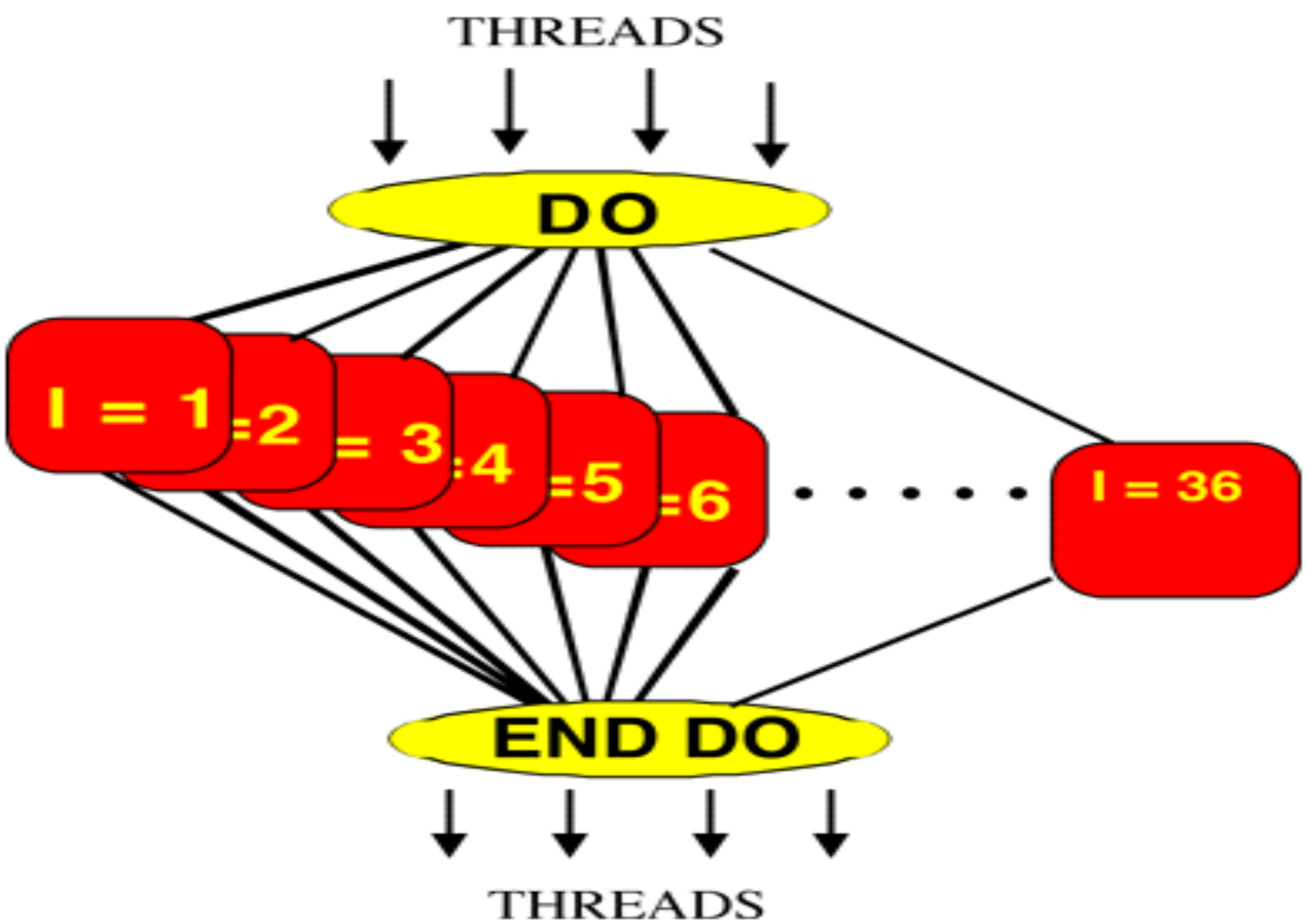
Sequential program

```
void main()
{
   double Res[1000];
   #pragma omp parallel for
   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Parallel program

Work-Sharing Constructs

- Divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs **do not** launch new threads.
- No implied barrier upon entry to a work sharing construct.
- However, there is an implied barrier at the end of the work sharing construct (unless nowait is used).


THREADS
DO
I = 1
=2
= 3
=4
=5
=6
I = 36
END DO
THREADS

C\C++ syntax for the *parallel for* directive

```
#pragma omp parallel for [clause [,] [clause…]]

  for ( index = first; index <= last last ; index++ ){
       body of the loop
  }
```

Work Sharing Constructs - example

Sequential code

```
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP // Region

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

OpenMP Parallel Region and a work-sharing for construct

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

The Schedule Clause SCHEDULE (type [,chunk])

- The schedule clause effects how loop iterations are mapped onto threads

schedule(static [,chunk])

- Deal-out blocks of iterations of size "chunk" to each thread

schedule(dynamic [,chunk])

- Each thread grabs "chunk" iterations off a queue until all iterations have been handled

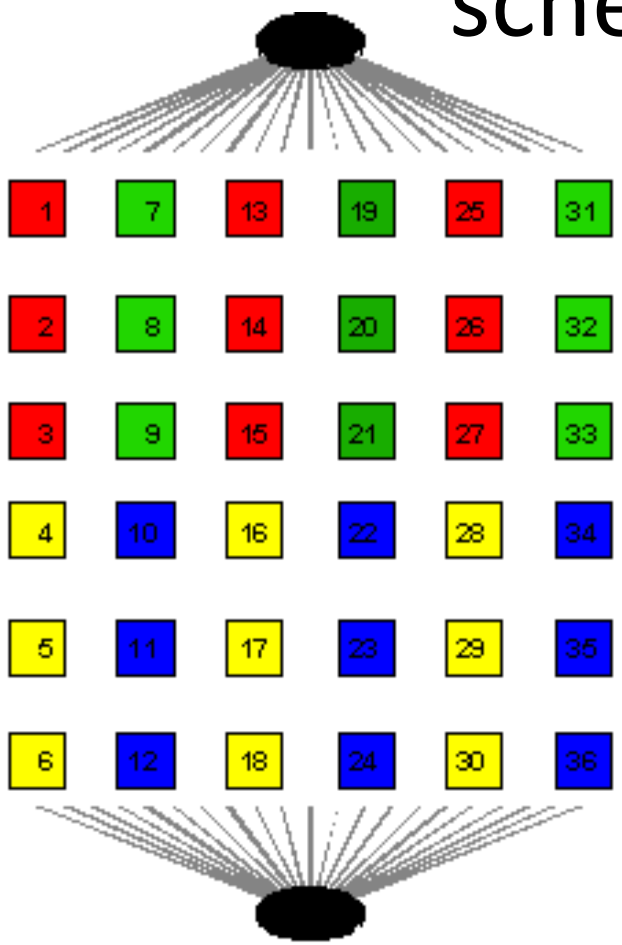
schedule(guided [,chunk])

- Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds

schedule(runtime)

- Schedule and chunk size taken from the OMP_SCHEDULE environment variable

schedule(static [,chunk])



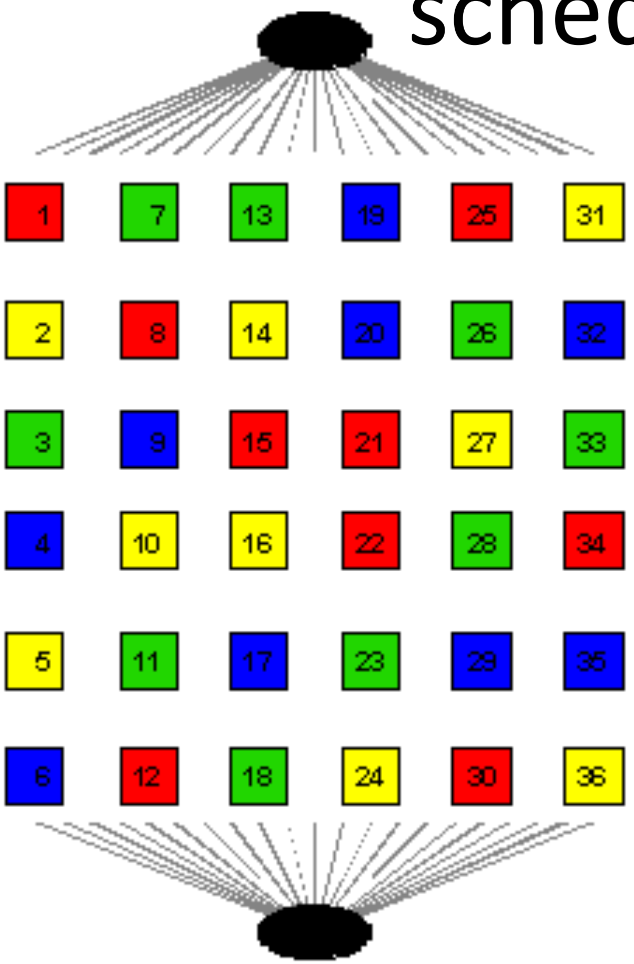
- Iterations are divided evenly among threads
- If chunk is specified, divides the work into chunk sized parcels
- If there are N threads, each thread does every N^{th} chunk of work.

```
!$OMP PARALLEL DO &
!$OMP SCHEDULE(STATIC,3)

DO J = 1, 36
        Work (j)
END DO

!$OMP END DO
```

schedule(dynamic [,chunk])



- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is one.
- More overhead, but potentially better load balancing.

```
!$OMP PARALLEL DO & !
$OMPSCHEDULE(DYNAMIC,1)

DO J = 1, 36
          Work (j)
END DO

!$OMP END DO
```

No Wait Clauses

- No wait: if specified then threads do not synchronise at the end of the parallel loop.
- For Fortran, the END DO directive is optional with NO WAIT being the default.
- Note that the nowait clause is incompatible with a simple parallel region meaning that using the composite directives will not allow you to use the nowait clause.

OpenMP: Reduction(op : list)

- The variables in "list" must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
  - pair wise "op" is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

OpenMP: A Reduction Example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i;
    double ZZ, func(), sum=0.0;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(ZZ)

    for (i=0; i< 1000; i++){
            ZZ = func(i);
            sum = sum + ZZ;
    }
}
```

if CLAUSE

We can make the parallel region directive itself conditional.

Fortran: IF (*scalar logical expression*)

C/C++: if (*scalar expression*)

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

SYNCHRONIZATION

OpenMP: How do Threads Interact?

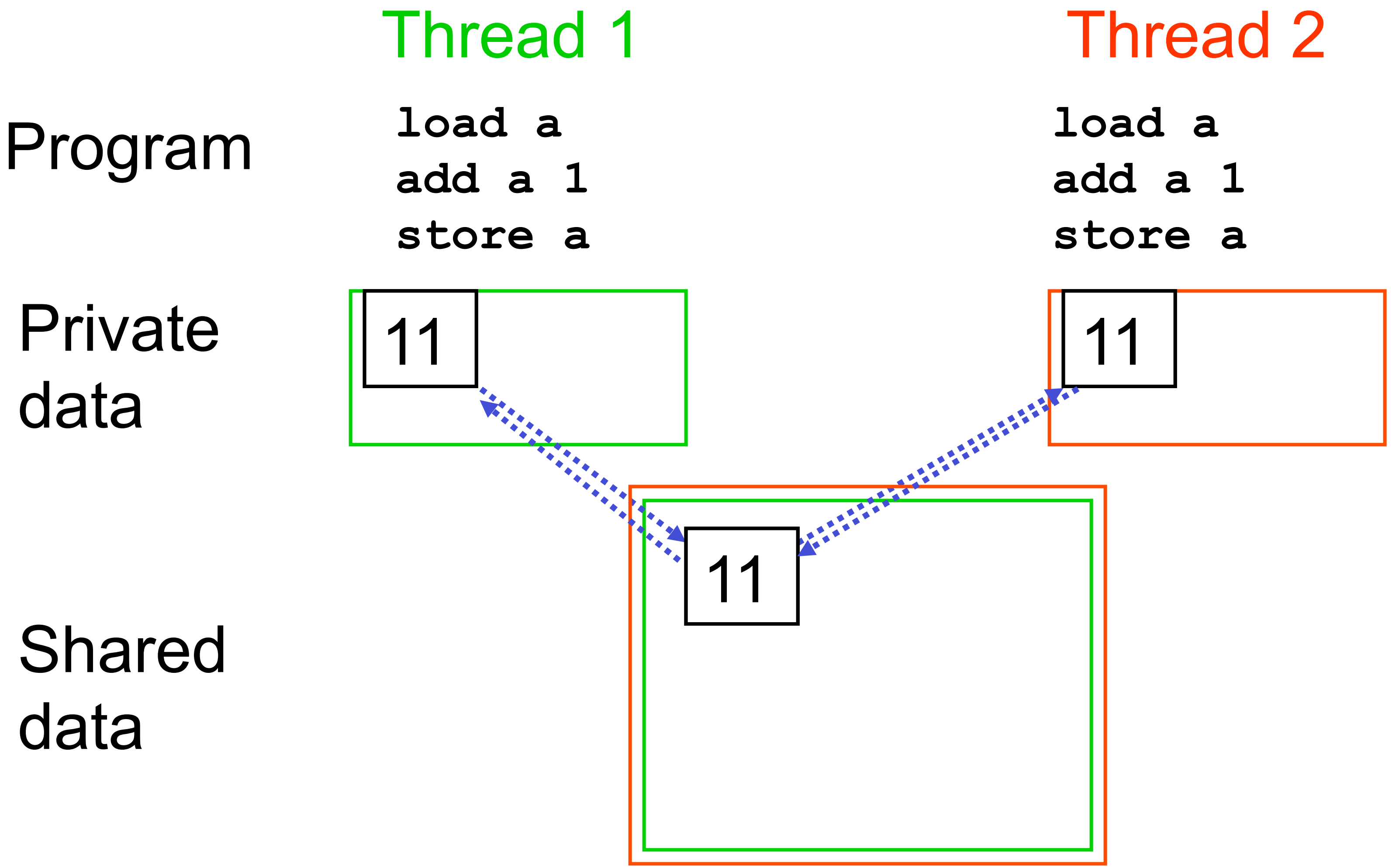
- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is stored to minimize the need for synchronization.

Note that updates to shared variables:

(e.g. a = a + 1)

are *not* atomic!

If two threads try to do this at the same time, one of the updates may get overwritten.


Thread 1
Thread 2
Program
load a
add a 1
store a
load a
add a 1
store a
Private data
11
11
Shared data
11

Barrier

Fortran - !$OMP BARRIER

C\C++ - #pragma omp barrier

- This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.
- Implicit barriers exist after work sharing constructs. The nowait clause can be used to prevent this behaviour.

Critical

- Only one thread at a time can enter a critical section.

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
    ...
!$OMP CRITICAL (STACKPROT)
    inext = getnext(stack)
!$OMP END CRITICAL (STACKPROT)
    call work(inext,inew)
!$OMP CRITICAL (STACKPROT)
    if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL (STACKPROT)
    ...
!$OMP END PARALLEL
```

Atomic

- Atomic is a special case of a critical section that can be used for certain simple statements

Fortran: **!$OMP ATOMIC**

statement

where *statement* must have one of these forms:

x = *x op expr*, *x* = *expr op x*, *x* = *intr* (*x*, *expr*) or

x = *intr* (*expr*, *x*)

op is one of **+**, *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**

intr is one of **MAX**, **MIN**, **IAND**, **IOR** or **IEOR**

C/C++: `#pragma omp atomic`

statement

where *statement* must have one of the forms:

x binop = *expr*, *x*`++`, `++`*x*, *x*`--`, or `--`*x*

and *binop* is one of `+`, `*`, `-`, `/`, `&`, `^`, `<<`, or `>>`

HANDS-ON ON THREADED LIBRARIES

OpenMP Practical

Compute pi by integrating f(x) = 4/(1 + x**2)

- Set the number of rectangles used in the approximation (n)
- Each thread:
  1. calculates the areas of the assigned rectangles
  2. Synchronizes for a global summation
- print the result

Main variables description:

- pi the calculated result
- n number of points of integration
- x midpoint of each rectangle's interval
- f function to integrate
- sum,pi area of rectangles