



The Abdus Salam  
International Centre  
for Theoretical Physics



# Parallel Programming 101

**Ivan Girotto – [igirotto@ictp.it](mailto:igirotto@ictp.it)**

International Centre for Theoretical Physics (ICTP)

# What is High-Performance Computing (HPC)?

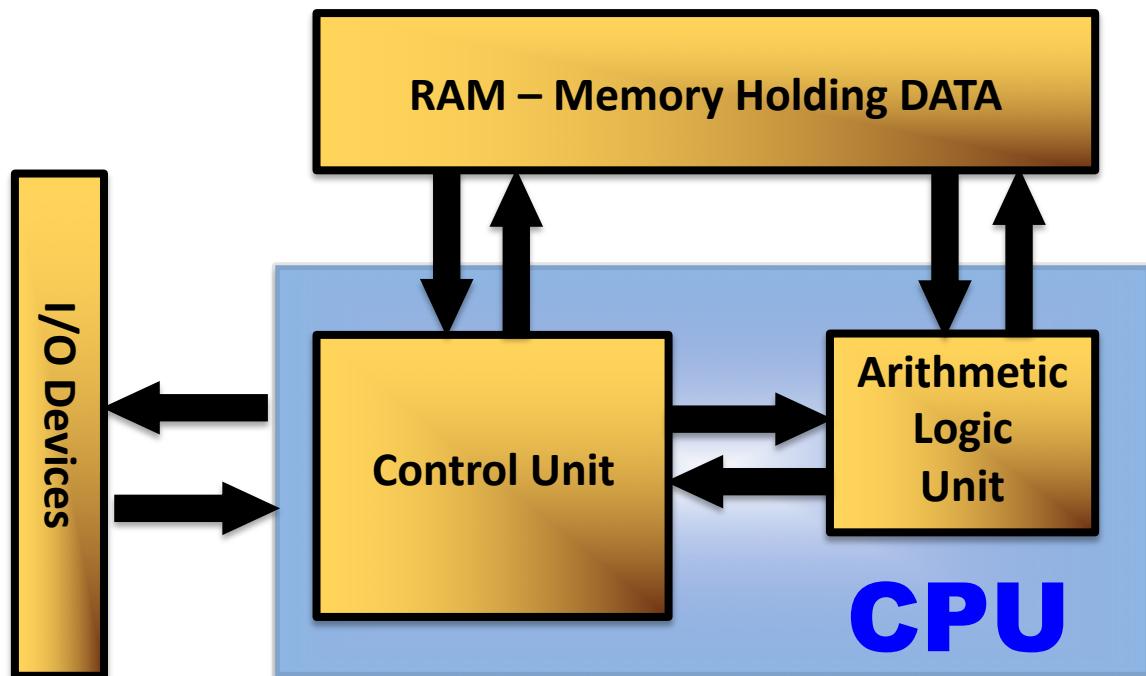


- Not a real definition, depends from the prospective:
  - HPC is when I care how fast I get an answer
  - HPC is when I foresee my problem to get bigger and bigger
- Thus HPC can happen on:
  - A workstation, desktop, laptop, smartphone!
  - A supercomputer
  - A Linux Cluster
  - A grid or a cloud
  - Cyberinfrastructure = any combination of the above
- HPC means also **High-Productivity Computing**

# The Classical Model



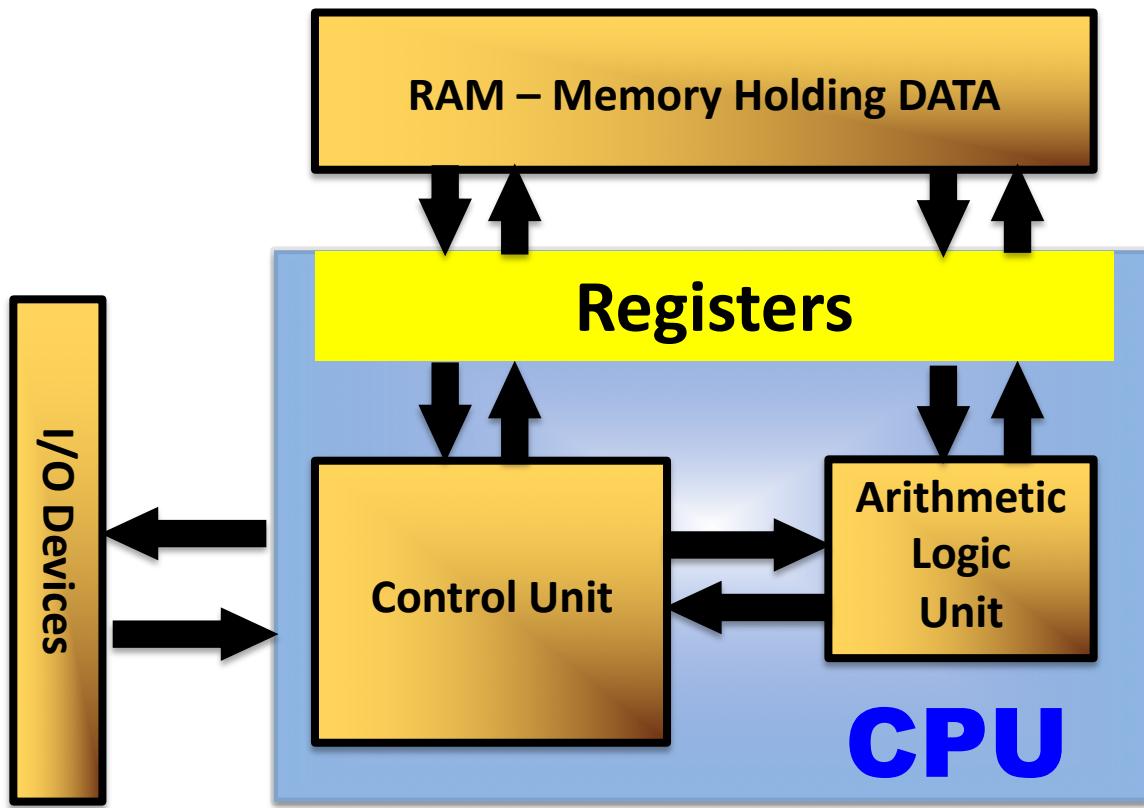
**John Von Neumann**



# The Classical Model

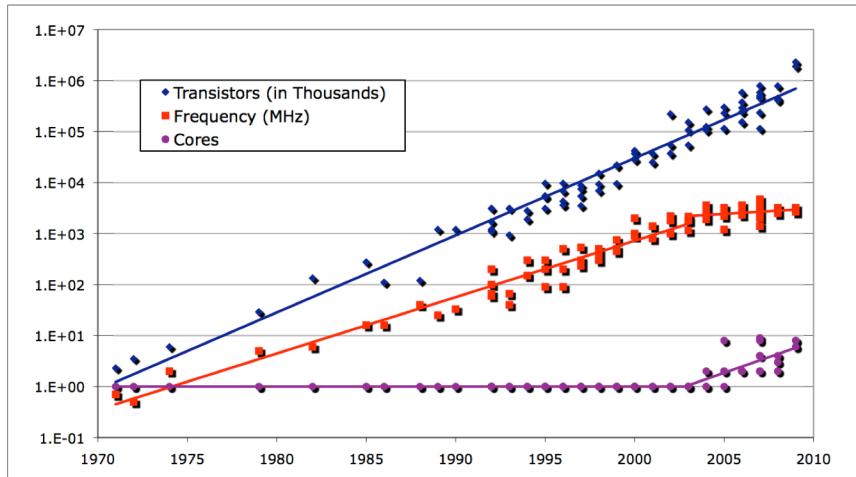


**John Von Neumann**



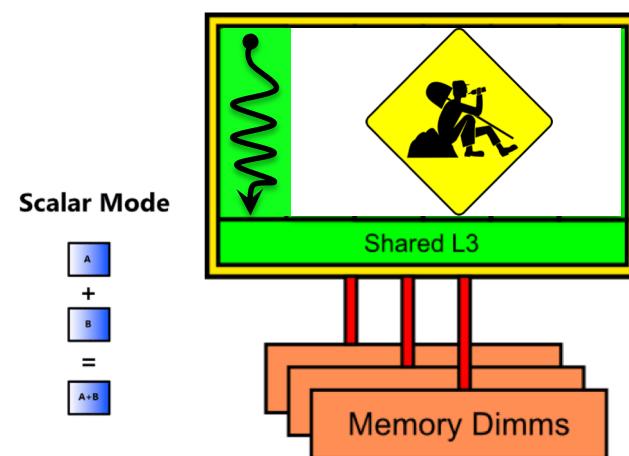
# Evolution of Computers (in a nutshell)

- Higher capability is achieved by increasing computer complexity



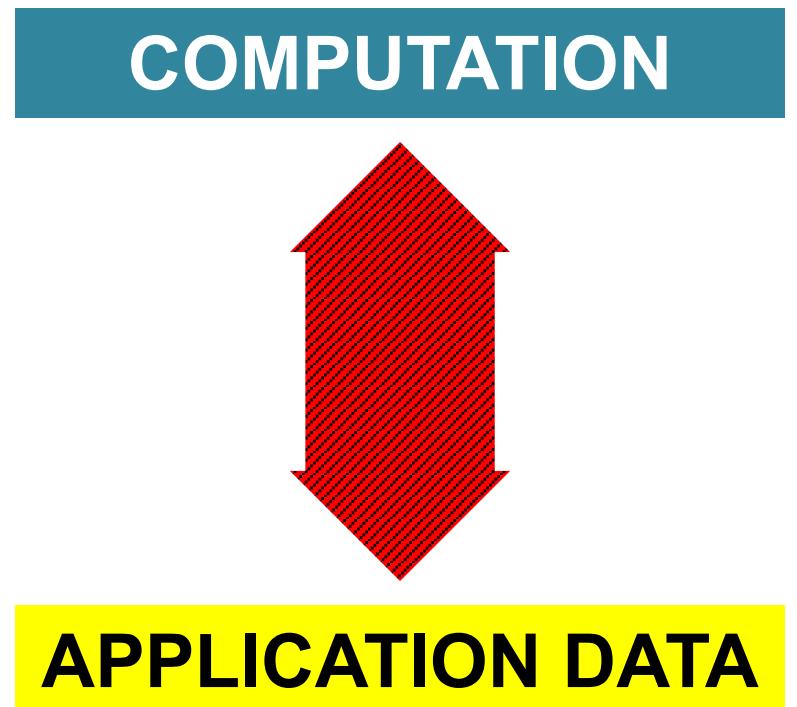
- CPU power is measured in number of floating point operations x second (FLOPs)
  - $\text{FLOPS} = \# \text{cores} \times \text{clock freq.} \times (\text{FLOP/cycle})$

#cores	Vector Length	Freq. (GHz)	GFLOPs
1	1	1.0	1
1	16	1.0	16
10	1	1.0	10
10	16	1.0	160

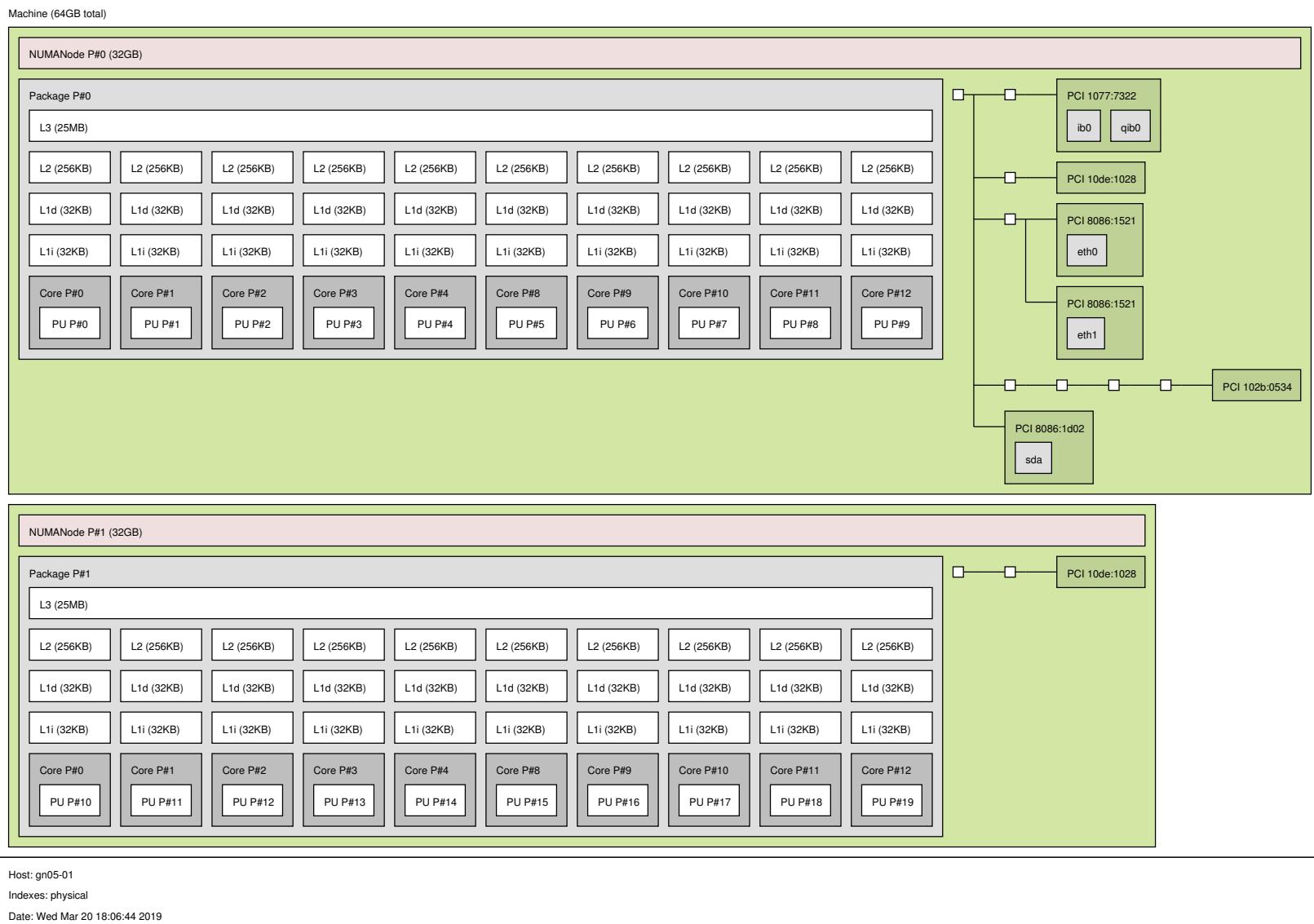


# Key Aspect towards Real Performances

- When all CPU component work at maximum speed that is called *peak of performance*
  - Tech-spec normally describe the theoretical peak
  - Benchmarks measure the real peak
  - Applications show the real performance value
- CPU performance is measured FLOP/s
- But the real performance is in many cases mostly related to the memory bandwidth (Bytes/s)
- The way data are stored in memory is a key-aspect for high performance



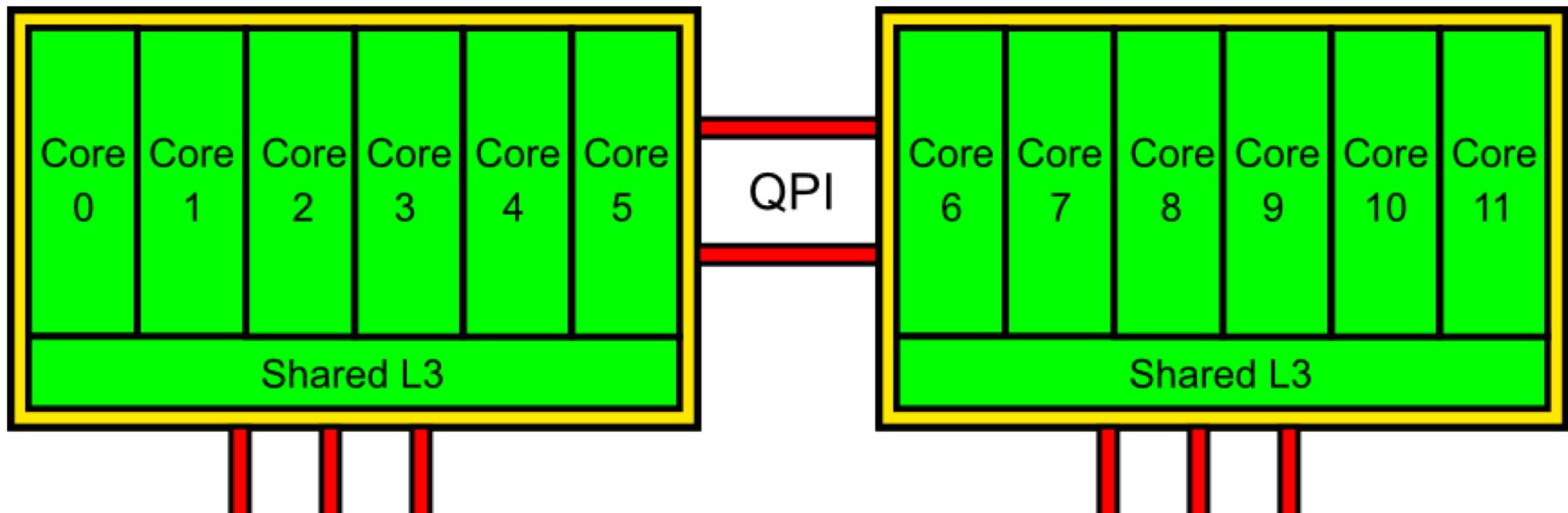
# Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz



Picture obtained on Ulysses compute node using the following commands:

```
$ module load hwloc/1.11.0
$ lstopo --output-format pdf output.pdf
```

# Real World NUMA Nodes



Main Memory  
Dual Socket - 64GB RAM

```
[igirotto@gn05-01 ~]$ free -g -o
```

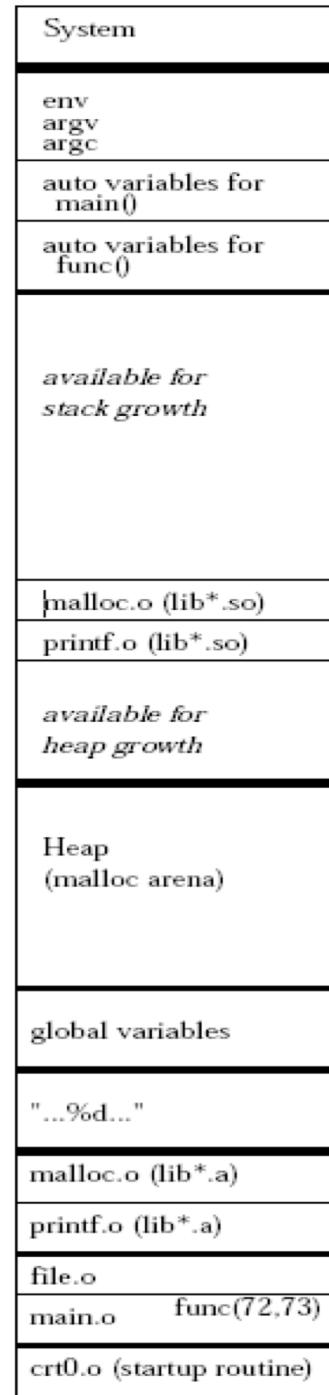
	total	used	free	shared	buffers	cached
Mem:	62	1	61	0	0	0
Swap:	0	0	0			

# TEXT DATA

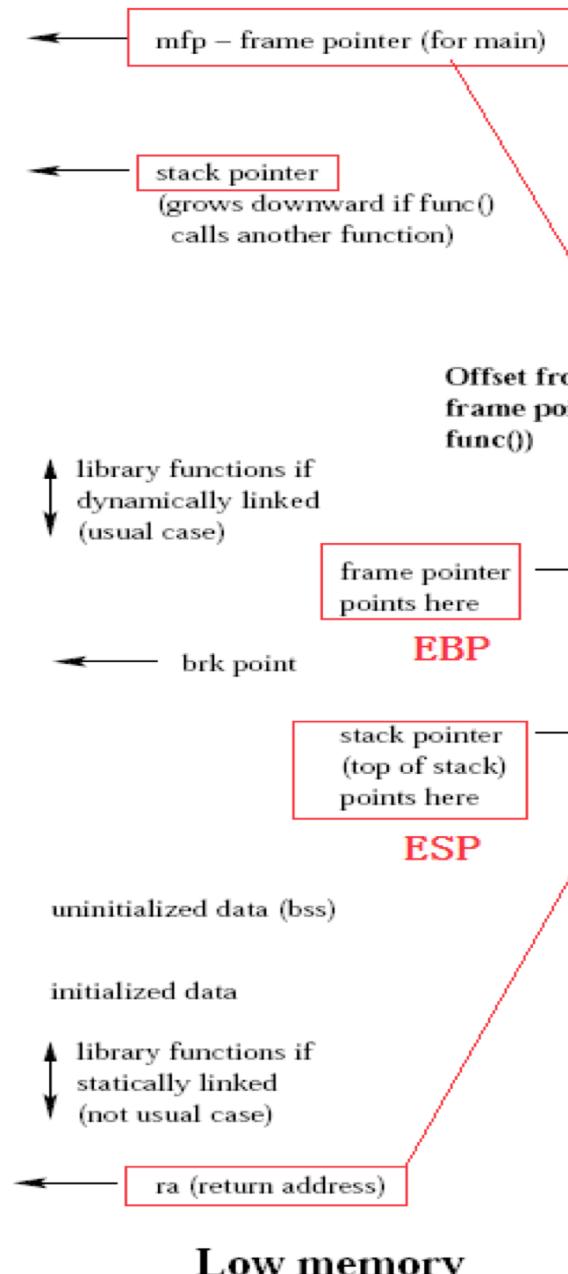
compiled code (a.out)

# SHARED MEMORY

# STACK



# High memory



Stack illustrated after the call func(72,73) called from main().  
assuming func defined by:  
func(int x, int y) {  
 int a;  
 int b[3];  
 /\* no other auto variables \*/

Assumes int = long = char \* of size 4 and assumes stack at high address and descending down.

# Expanded view of the stack

## Stack

main()	auto variables
73	y
72	x
ra	return address
mfp	caller's frame pointer
garbage	a
garbage	b[2]
garbage	b[1]
garbage	b[0]

## Contents

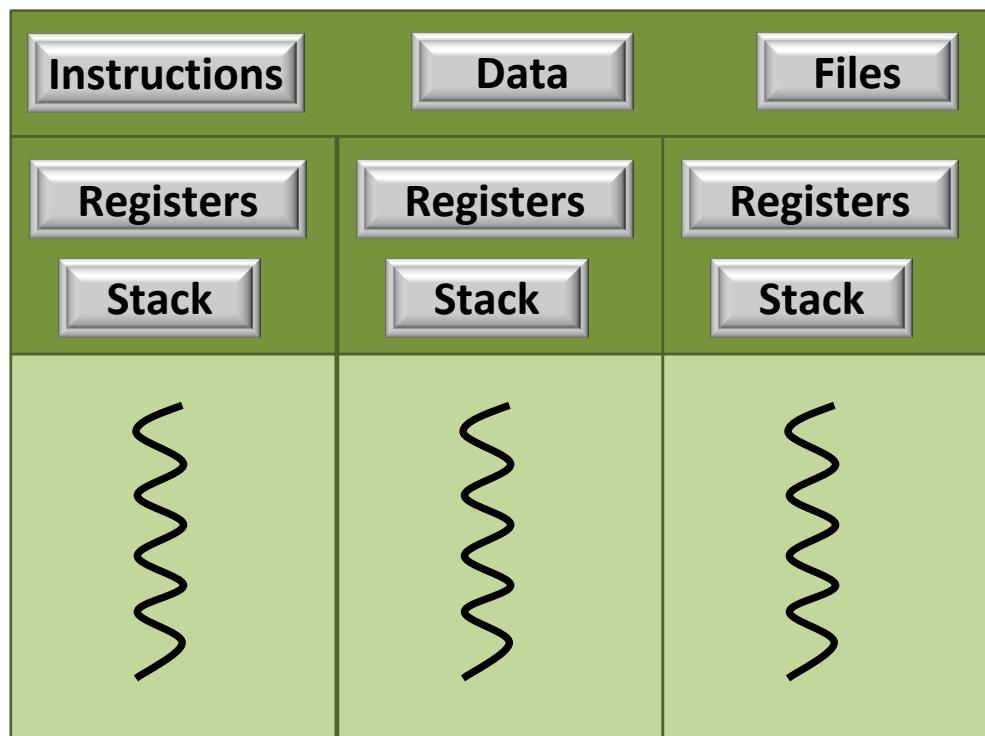
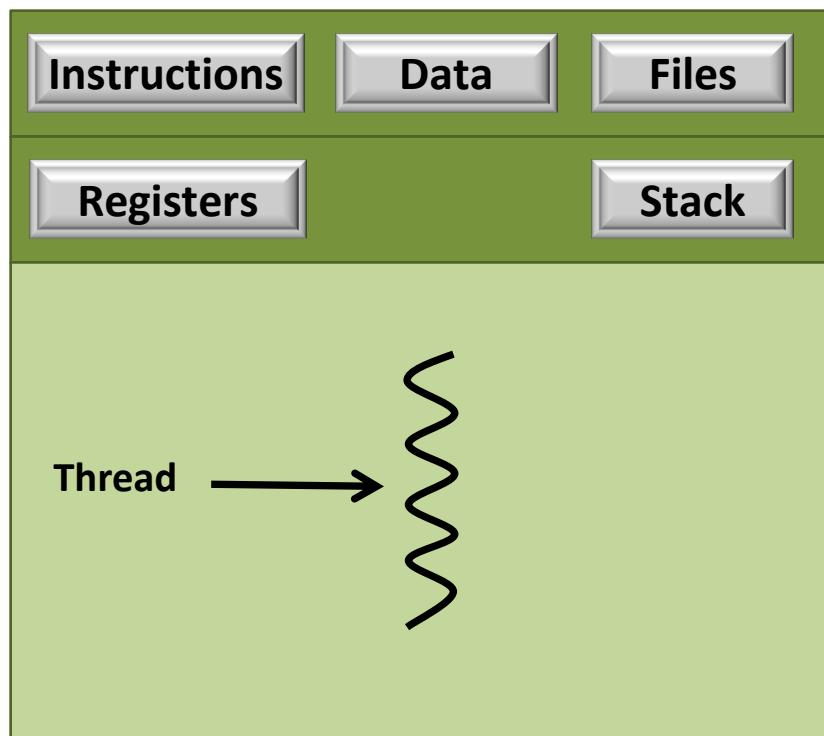
y
x
return address
caller's frame pointer
a
b[2]
b[1]
b[0]

All auto variables and parameters are referenced via offsets from the frame pointer.

The frame pointer and stack pointer are in registers (for fast access).

When funct returns, the return value is stored in a register. The stack pointer is moved to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

# Processes and Threads



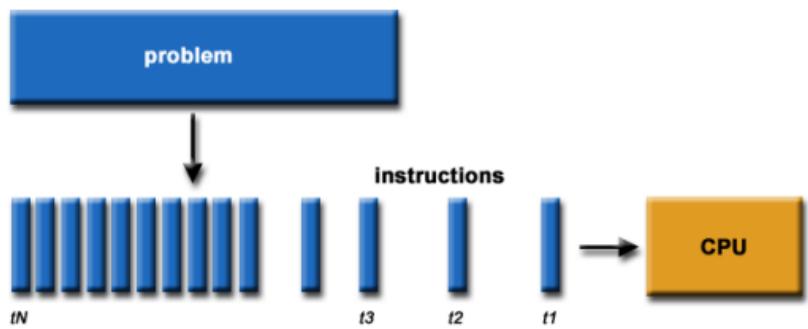
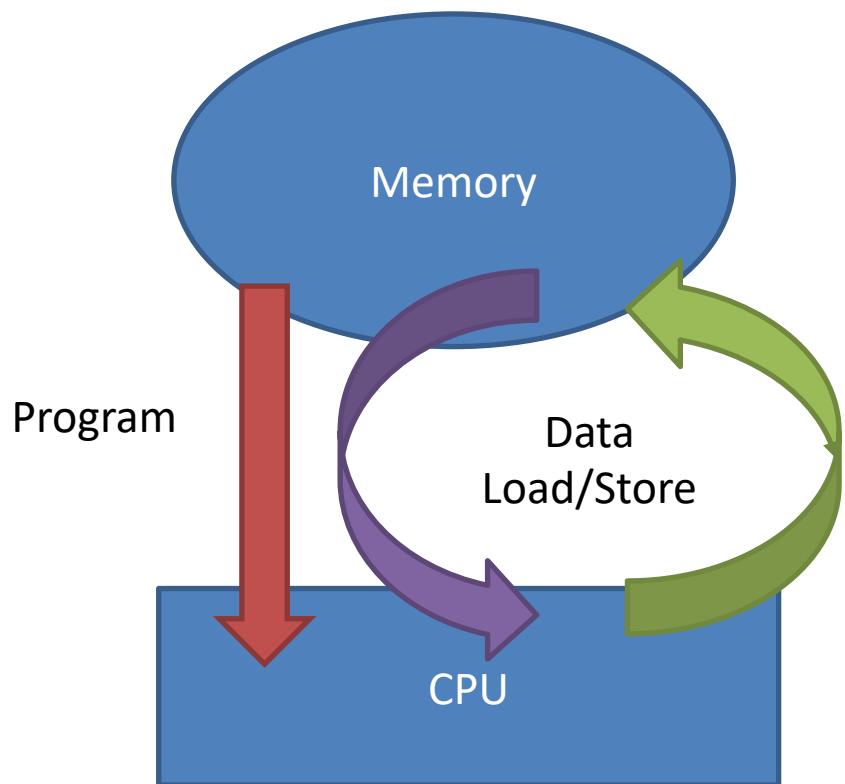
# Multi-threading - Recap



- A thread is a (**lightweight**) process - an instance of a program plus its own 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.
- A *master thread* is responsible for co-ordinating the threads group

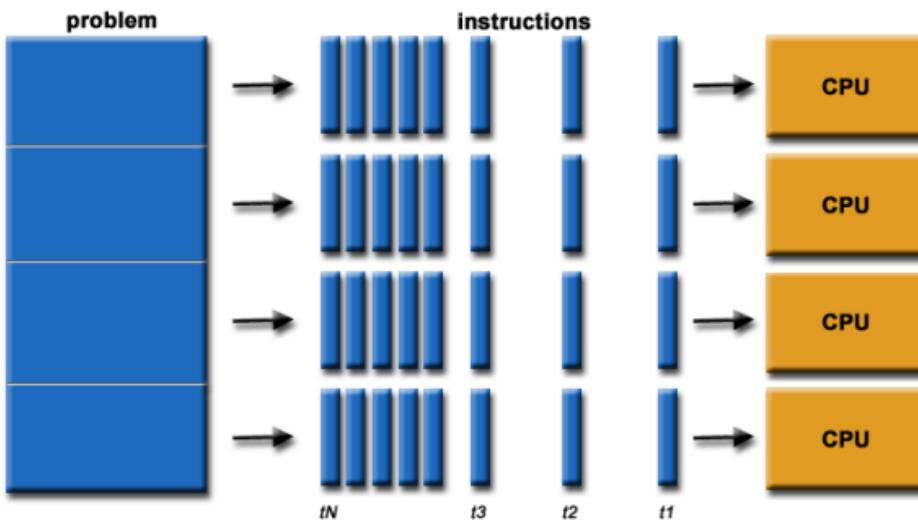
# Serial Programming

- A problem is broken into a discrete series of instructions
- Instructions are executed one after another
- Only one instruction may execute at any moment in time



# Parallel Programming

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently



- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control / coordination mechanism is employed

# Fundamental Steps of Parallel Design



- 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

- Granularity is determined by the decomposition level (number of tasks) on which we want to divide the problem
- The degree to which task/data can be subdivided is limited 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
  - make your code modularized to enhance different levels of granularity and consequently to become more “platform adaptable”

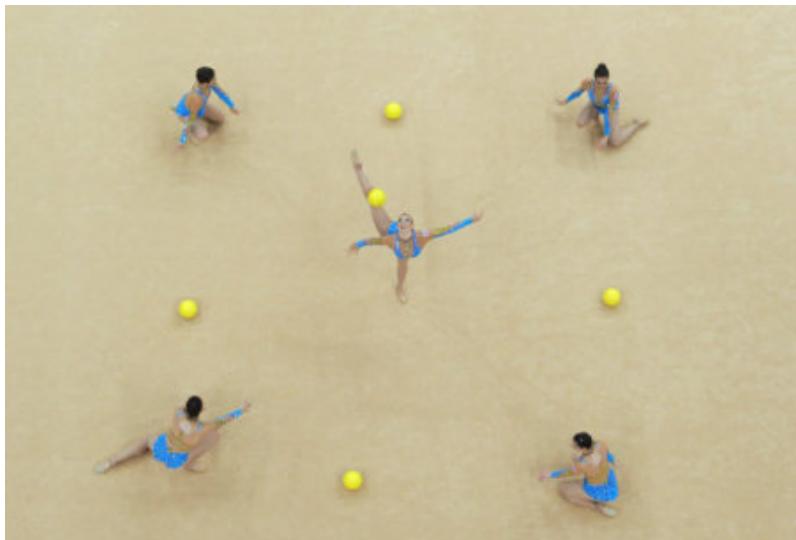
# Type of Parallelism

- **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



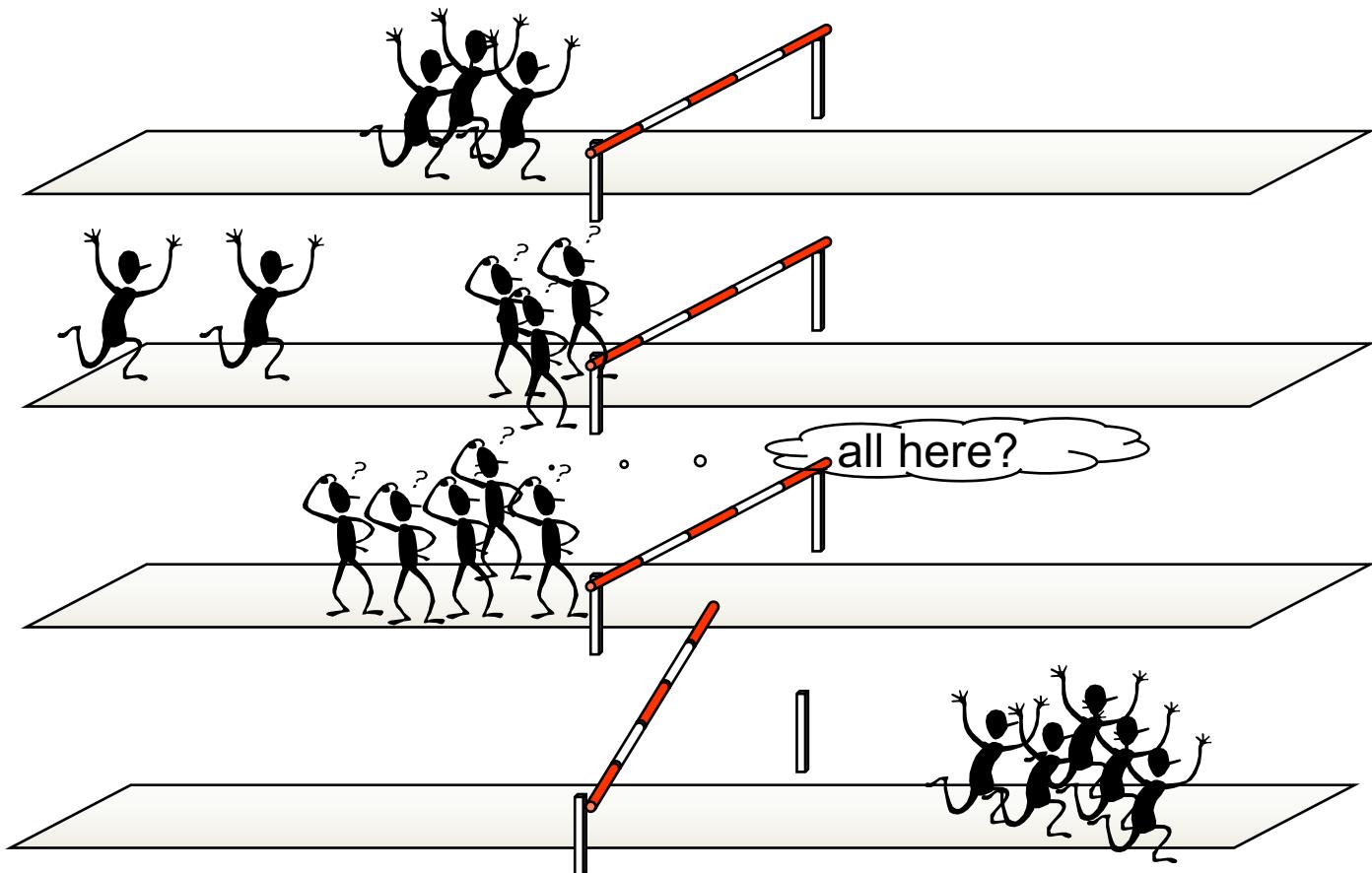
# Tasks Interactions

- 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**)



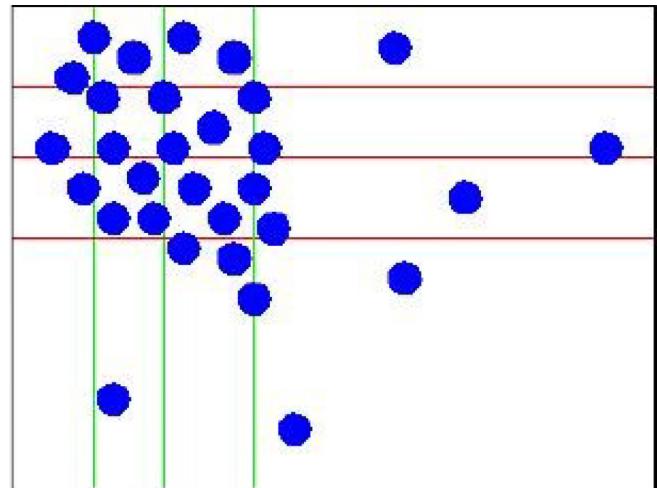
- Equally divide the work among the available resource: processors, memory, network bandwidth, I/O, ...
- This is usually a simple task for the problem decomposition model
- It is a difficult task for the functional decomposition model

# Effect of Load Unbalancing



# Limitations of Parallel Computing

- Fraction of serial code limits parallel speedup
- Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution
- Load imbalance:
  - parallel tasks have a different amount of work
  - CPUs are partially idle
  - redistributing work helps but has limitations
  - communication and synchronization overhead



- In parallel programming, developers must manage exclusive access to shared resources
- Resources are in different forms:
  - concurrent read/write (including parallel write) to shared memory locations
  - concurrent read/write (including parallel write) to shared devices
  - a message that must be send and received

# Shared Resources

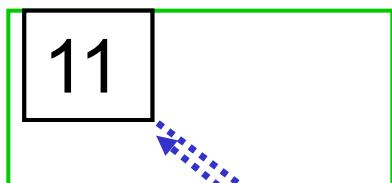
Program

Private  
data

Shared  
data

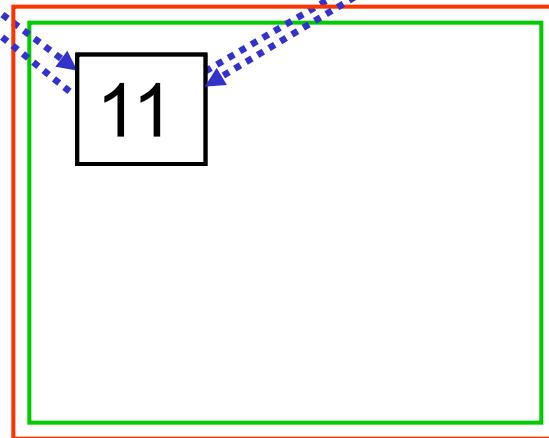
Thread 1

load a  
add a 1  
store a



Thread 2

load a  
add a 1  
store a



# Fundamental Tools of Parallel Programming



- Time measurement becomes a relevant issue:
  - MPI\_WTIME()
  - Clock
  - Seconds()

# How do we evaluate the improvement?



- We want estimate the amount of the introduced overhead =>  $T_o = n_{pes} T_p - T_s$
- But to quantify the improvement we use the term **Speedup**:

$$S_p = \frac{T_s}{T_p}$$

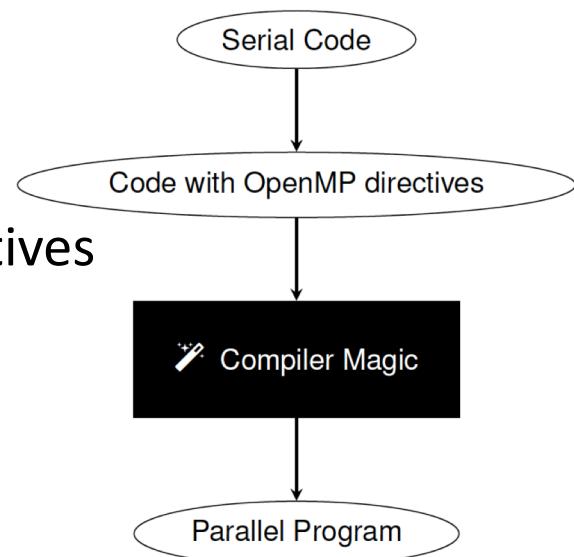
- Amdahl's law (there is a serial part which is not effected)
- Parallelism introduces overhead due to communication, synchronization, additional operations (I/O, memory copies, reordering, etc...)
- Sometimes the limit of scalability can be somehow predicted:
  - is there enough work for any process?
  - When considering complex problem not all part of the problem scale equally

**OpenMP**<sup>TM</sup>

- OpenMP **is not a programming language**
  - Rather it works in conjunction with existing languages such as standard Fortran or C/C++
- A high-level application programming interface (API) used to write multi-threaded, portable shared-memory applications
- Application Programming Interface (API)
  - that provides a **portable** model for parallel applications
  - Three main components:
    - Compiler **directives**
    - **Runtime library** routines
    - **Environment variables**

# OpenMP Parallelization

- OpenMP is directive based
  - code (can) work without them
  - you annotate your code with OpenMP directives
- OpenMP can be added incrementally
- OpenMP only works in shared memory
  - multi-socket nodes, multi-core processors
- OpenMP hides the calls to a threads library
  - less flexible, but much less programming
- **Caution:** write access to shared data can easily lead to race conditions and incorrect data



- 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**) *directive [clause [clause]...]*
  - C/C++:  
**#include <omp.h>**  
**#pragma omp directive [clause [clause]...]**

# Hello World!



```
#include <stdio.h>

int main() {
    printf("Hello World!\n");

    return 0;
}
```

Output:

```
Hello World!
```

# Hello World!



```
#include <stdio.h>

int main() {
    #pragma omp parallel
    printf("Hello OpenMP! \n");

    return 0;
}
```

# Hello World!



```
#include <stdio.h>

int main() {
    printf("Starting!\n");

    #pragma omp parallel
    printf("Hello OpenMP!\n");

    printf("Done!\n");

    return 0;
}
```

## Output:

```
Starting!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Done!
```

# Compiling



## GCC

```
gcc -fopenmp -o omp_hello omp_hello.c
```

```
g++ -fopenmp -o omp_hello omp_hello.cpp
```

## Intel

```
icc -qopenmp -o omp_hello omp_hello.c
```

```
icpc -qopenmp -o omp_hello omp_hello.cpp
```

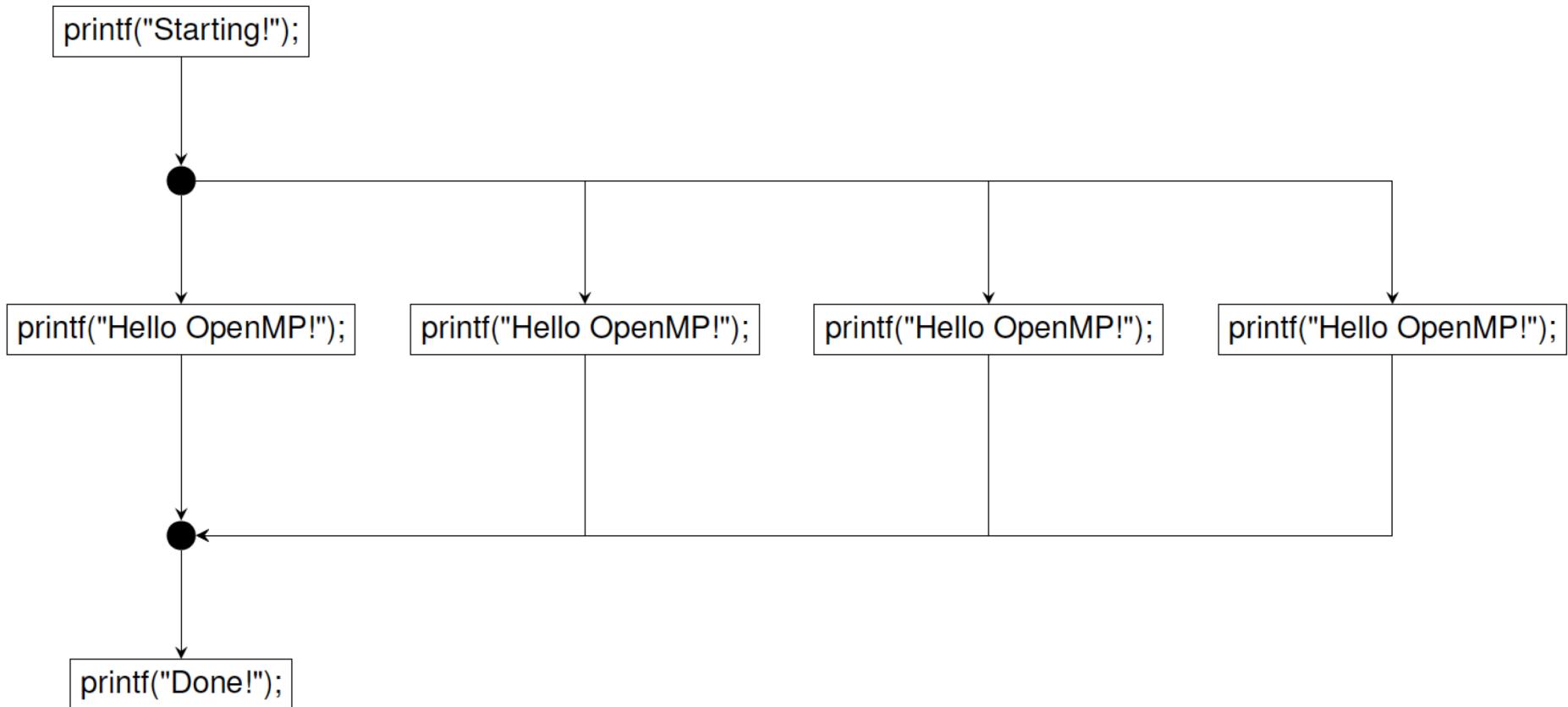
# Running



```
# default: number of threads equals number of cores  
./omp_hello
```

```
# set environment variable OMP_NUM_THREADS to limit default  
$ OMP_NUM_THREADS=4 ./omp_hello  
  
# or  
  
$ export OMP_NUM_THREADS=4  
$ ./omp_hello
```

# The Fork/Join Model



- ▶ Each thread executes the structured block independently
- ▶ The end of a parallel region acts as a **barrier**
- ▶ All threads must reach this barrier, before the main thread can continue.

# How many threads?



## 1. At the parallel directive

```
#pragma omp parallel num_threads(4)
{
    ...
}
```

## 2. Setting a default via the `omp_set_num_threads(n)` library function

Set the number of threads that should be used by the **next** parallel region

## 3. Setting a default with the `OMP_NUM_THREADS` environment variable

number of threads that should be spawned in a parallel region if there is no other specification. By default, OpenMP will use all available cores.

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

- ▶ Requires the inclusion of the `omp.h` header!

`omp_get_num_threads()`

Returns the number of threads in current team

`omp_set_num_threads(n)`

Set the number of threads that should be used by the **next** parallel region

`omp_get_thread_num()`

Returns the current thread's ID number

`omp_get_wtime()`

Return walltime in seconds

# Parallel Region



Launches a team of threads to execute a block of structured code in parallel.

```
#pragma omp parallel
statement; // this is executed by a team of threads

// implicit barrier: execution only continues when all
// threads are complete
```

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
// implicit barrier: execution only continues when all
// threads are complete
```

# SPMD (Single Process Multiple Data)

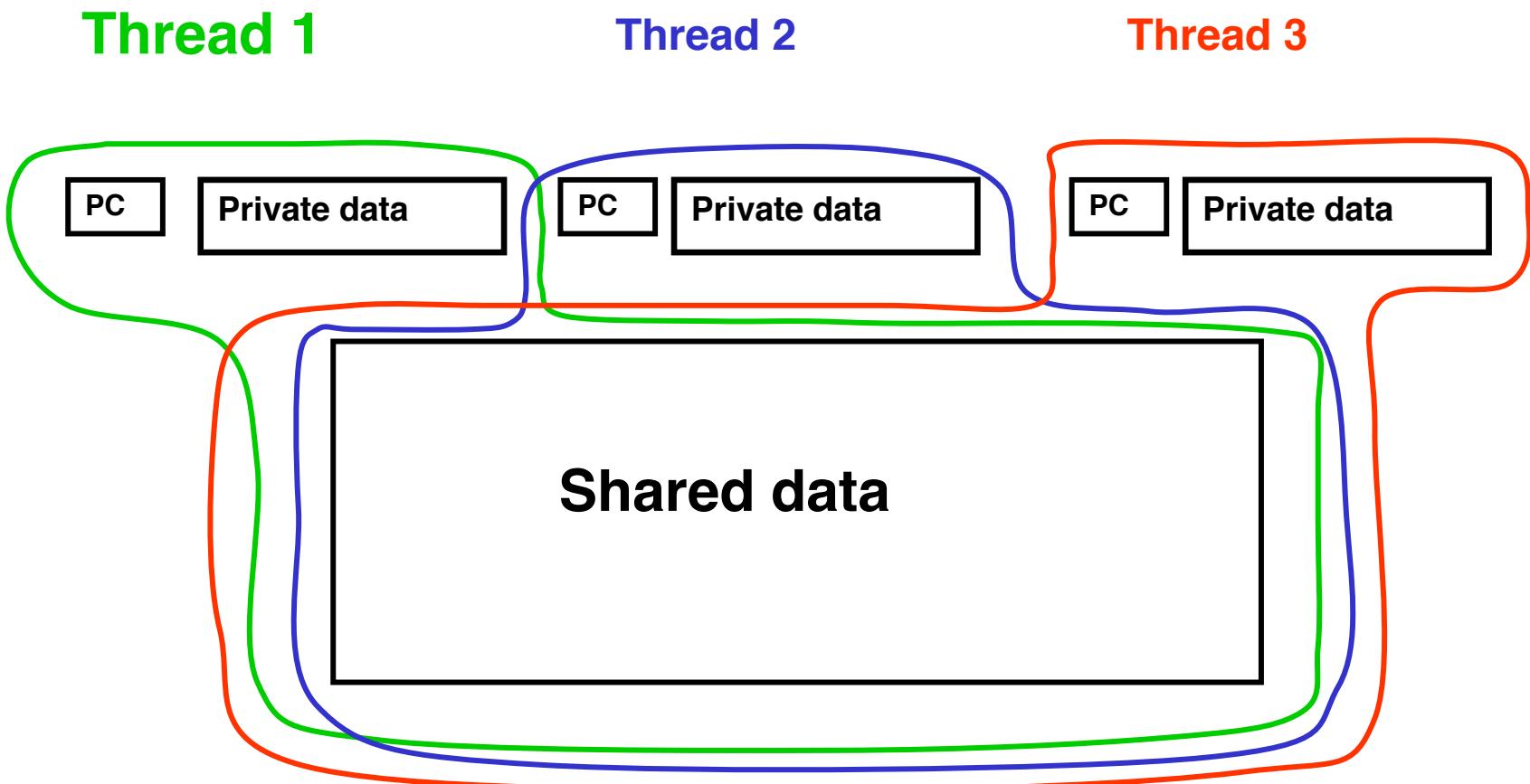


```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        printf("Hello from thread %d/%d!\n", tid, nthreads);
    }
    return 0;
}
```

```
Hello from thread 2/4!
Hello from thread 1/4!
Hello from thread 0/4!
Hello from thread 3/4!
```

# OpenMP Data Environment



# Variable Scope /1



- ▶ by default all variables which are visible in the parent scope of a parallel region are **shared**
- ▶ variables declared inside of the parallel region are by definition of the scoping rules of C/C++ only visible in that scope. Each thread has a **private** copy of these variables

```
int a; // shared

#pragma omp parallel
{
    int b; // private
    ...
    // both a and b are visible
    // a is shared among all threads
    // each thread has a private copy of b
    ...
} // end of scope, b is no longer visible
```

# Variable Scope /2

- ▶ a variable's scope can be modified at the beginning of a parallel region using clauses
- ▶ useful for legacy code where all variables are declared at the beginning of a function.  
E.g. in Legacy Fortran an C code you need to declare all variables at the beginning of a function.

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel shared(a) private(b,c)
{
    // a is shared among all threads (a = 1.0)
    // each thread has a private copy of b and c
    // b = uninitialized
    // c = uninitialized
    // outside b and C are not visible
    ...
}
```

# Variable Scope /3

- ▶ a variable's scope can be modified at the beginning of a parallel region using clauses
- ▶ useful for legacy code where all variables are declared at the beginning of a function.  
E.g. in Legacy Fortran an C code you need to declare all variables at the beginning of a function.

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel shared(a)
{
    // a is shared among all threads (a = 1.0)
    // each thread has a private copy of b and c
    int b, c;
    // outside b and c are not visible
    ...
}
```

# Variable Scope /3



- ▶ the `firstprivate` clause does the same as `private`, but initializes each copy with the value of the parent thread.

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel firstprivate(b) private(c)
{
    // a is shared
    // the value of the private b is 3.0
    // the value of the private c is uninitialized
    ...
}
```

Let's make some examples ...

- 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).

# Work-Sharing: loop

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

# Work-Sharing: loop

Sequential code

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

OpenMP // Region

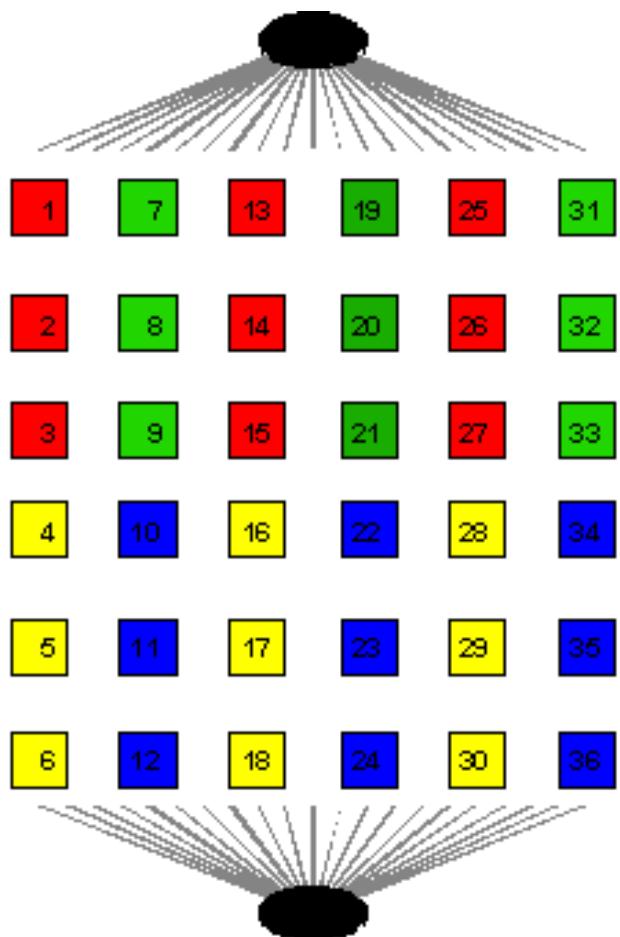
```
#pragma omp parallel  
{  
    int id, i, Nthrds, istart, iend;  
    id = omp_get_thread_num();  
    Nthrds = omp_get_num_threads();  
    istart = id * N / Nthrds;  
    iend = ( id + 1 ) * N / Nthrds;  
    for(i = istart; i < iend; i++ ) {  
        a[ i ] = a[ i ]+ b[ i ];  
    }  
}
```

OpenMP Parallel  
Region and a work-  
sharing for construct

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

# schedule(static [,chunk])

- Deal-out blocks of iterations of size “chunk” to each thread
- 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<sup>th</sup> chunk of work.



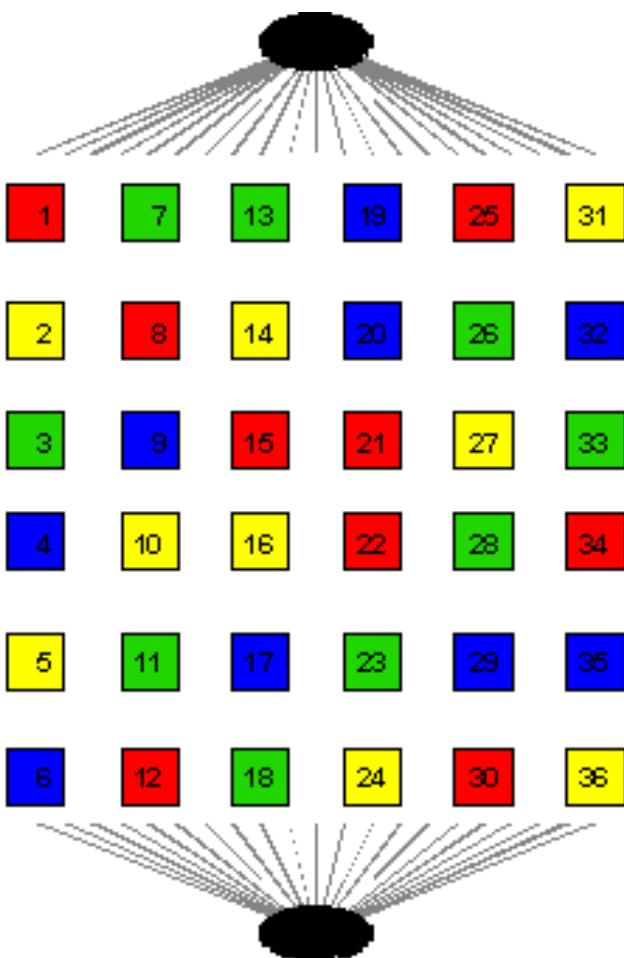
!\$OMP PARALLEL DO &  
!\$OMP SCHEDULE(STATIC,3)

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

!\$OMP END DO

# schedule(dynamic [,chunk])

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

<https://computing.llnl.gov/tutorials/openMP/>