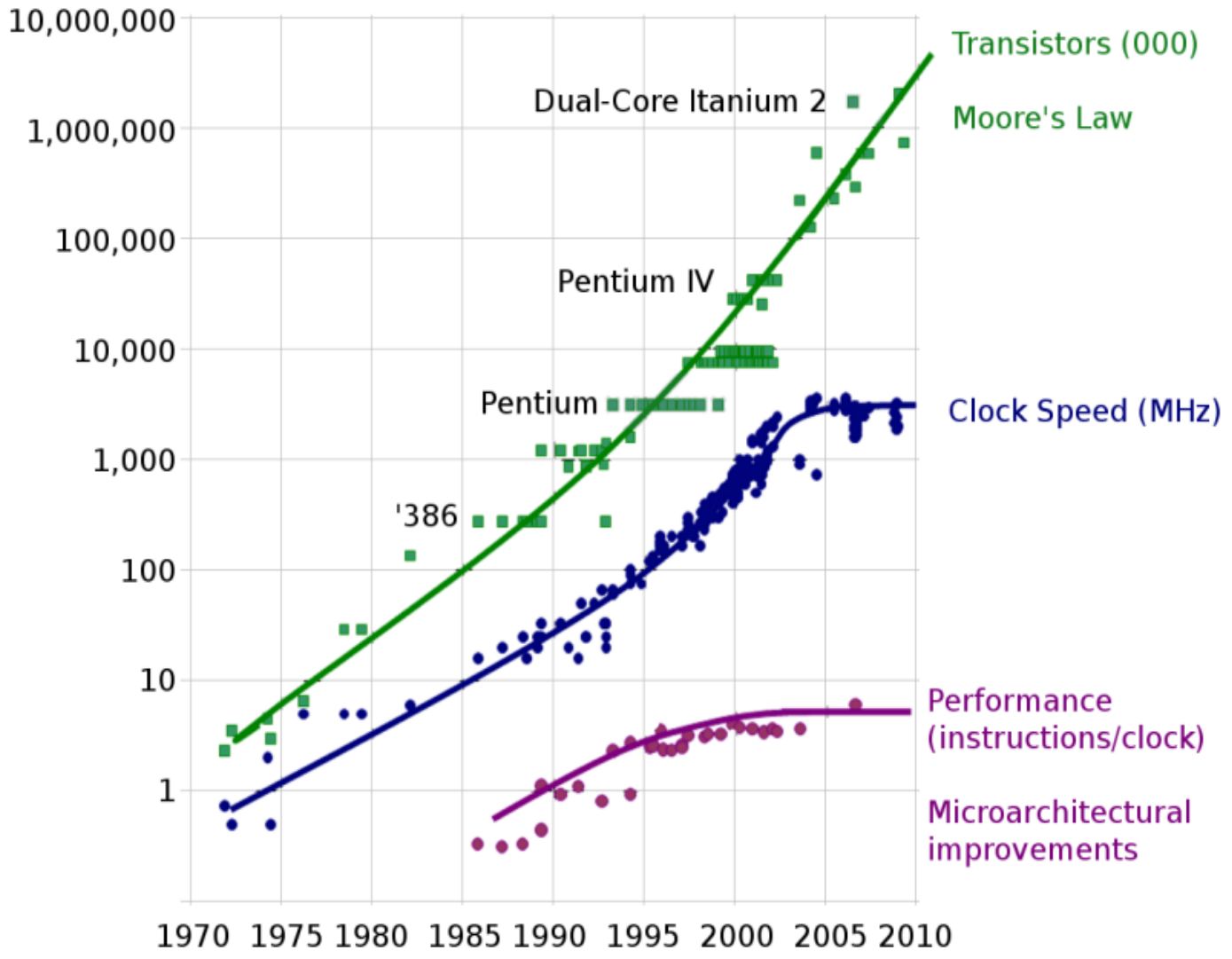
OpenMP

Open Specifications for Multi Processing

OUTLINE

- Introduction
- OpenMP Programming Model
- OpenMP Directives
- Synchronization Constructs
- Runtime Libraries
- Environment Variables

Intel CPU Trends



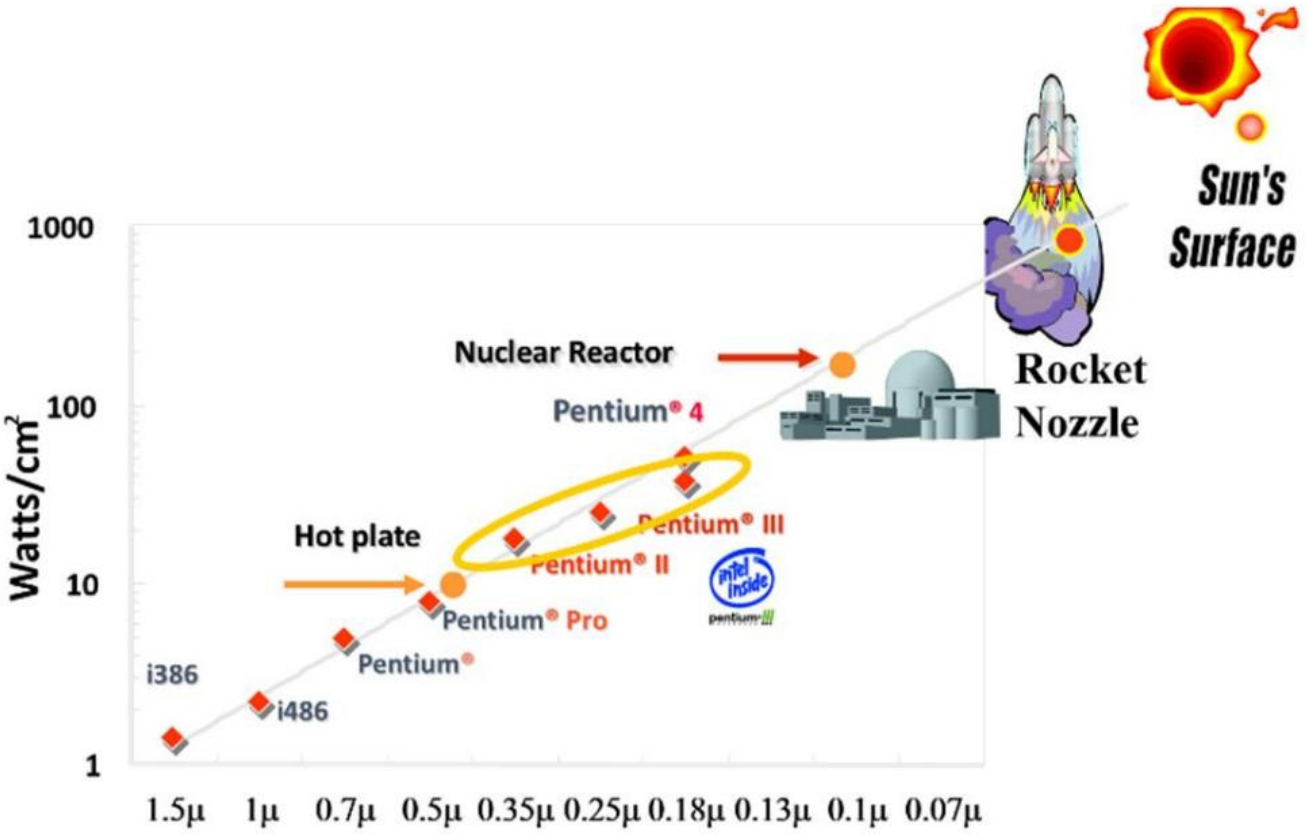
Sutter, *The Free Lunch is Over*, DDJ 2005.

Data: Intel, Wikipedia, K. Olukotun

During 50 yrs of Moore's Law: transistor density X 2 every 2 years

- Transistors grew smaller
- Smaller transistors require less voltage and current
  - Can switch at higher frequency at same power
  - Clock rates increased
  - Capabilities could be added/expanded at constant power
    - Instruction-level parallelism
    - OO executions
    - Prefetching
    - More cache
- Single processor performance X 2 every 18 months

Hitting the Power Wall



Limits to single processor performance

- Moore's Law- CPU doubles its frequency every 2 yrs.
  - Possible by increasing the transistor density
  - Difficult to sustain because of heat produced
- Power increase with transistor size
- Power per transistor cannot be reduced
  - More transistors packed in same area -> higher power density -> more heat
  - Faster processors get too hot(~171F at 3.8GHz), cooling gets bulky and expensive
  - Speed has not really increased (~ 4 GHz max)
  - Adding lower frequency cores keeps computational capacity increasing at comparable power

Multicores

Logical extension is Multicore

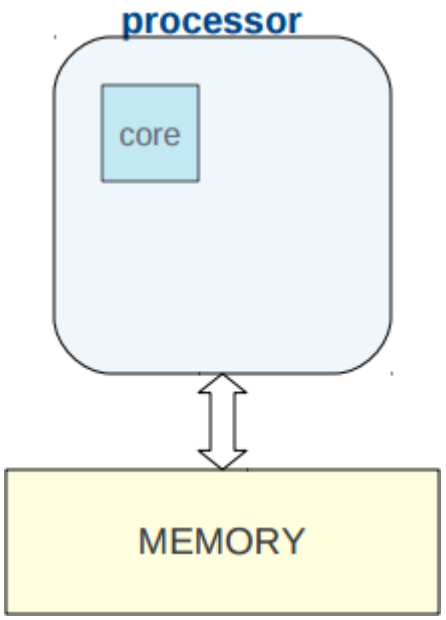
- 100s relatively slow(1.5GHz) processors vs 10s relatively fast(2.5GHz) processors
- Overall more compute available for comparable power
- Multicore is more general purpose
- A multi-core processor is an integrated circuit to which two or more processors have been attached for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks.

Write Parallel Programs

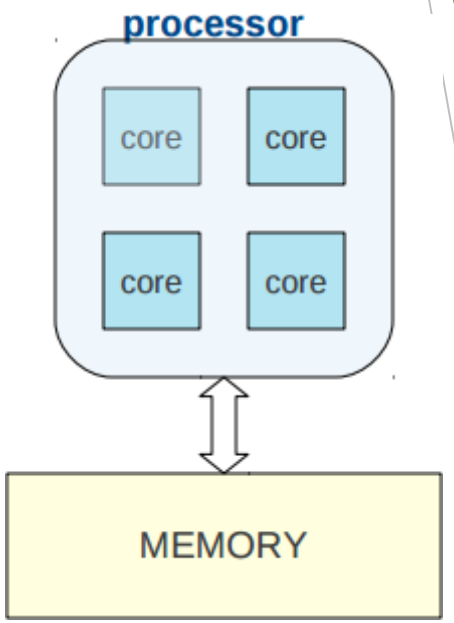
Partition of Work:

- Task Parallelism: Parallelize he different tasks
- Data Parallelism: Split data among cores
- Parallel Programs may contain both data and task parallelism
  - Eg: Sum of N numbers in parallel
- Inter-core coordination: Communication between ores lead to complexity
  - Communication
  - Synchronization
  - Load Balancing

INTRODUCTION



Older processors have only one CPU core to execute instructions



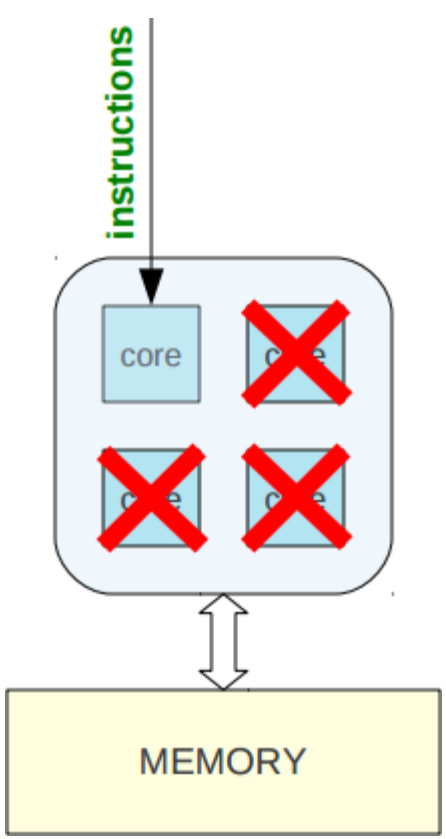
Modern processors have many CPU cores to execute instructions

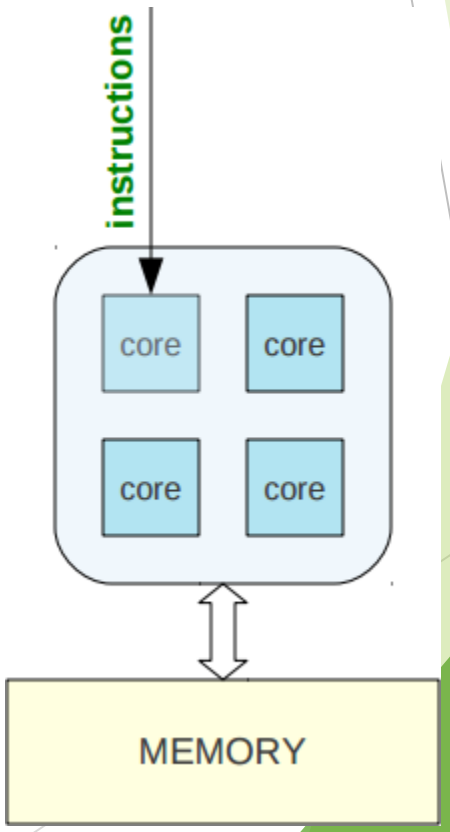
WHY OPENMP?

When you run a Sequential Program:

- Instructions executed on 1 core
- Other cores are idle
- Wastage of available resources – We want all cores to be used – How?

Use "OpenMP"





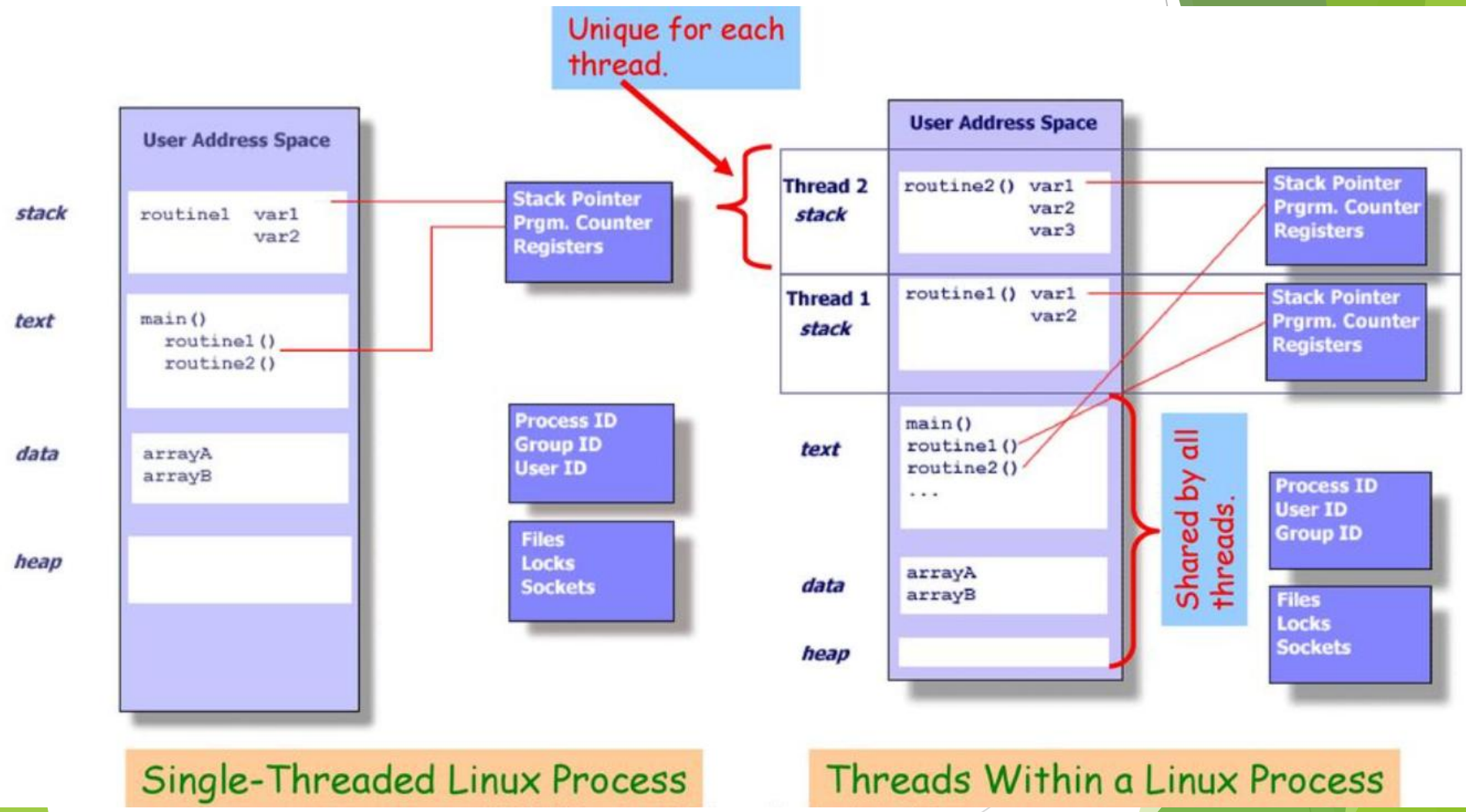
INTRODUCTION TO OPENMP

- OpenMP is an Application Program Interface(API)
- It provides a portable scalable model for developers of shared memory parallel applications
- The API supports C, C++ and FORTRAN
- OpenMP API Consists of :
  - Compiler Directives(eg: #pragma)
  - Runtime Library Routines (eg : omp_())
  - Environment variables (eg: OMP_)
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
- Scenarios
  - Creating new program
  - Parallelizing existing one

THREADS V/S PROCESS

- A process is a program in execution. A thread is a light weight process.
- Threads share the address space of the process that created it, process have their own address space.
- Threads can directly communicate with other threads of the same process. Processes must use IPC to communicate with other process.
- Changes to main thread may affect behaviour of other threads of the process. Changes to the parent process do not affect child process.

Threads

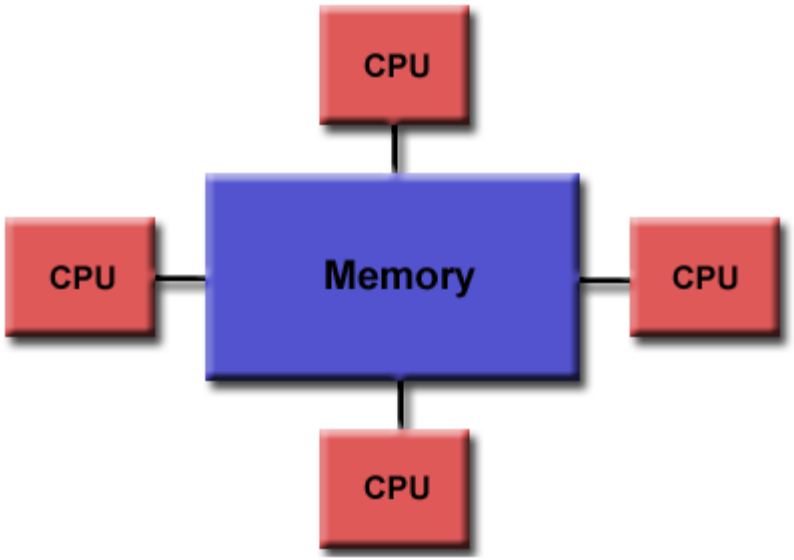

Unique for each thread.
User Address Space
stack
routine1 var1 var2
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets
Single-Threaded Linux Process
User Address Space
Thread 2
stack
routine2() var1 var2 var3
Thread 1
stack
routine1() var1 var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Shared by all threads.
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets
Threads Within a Linux Process

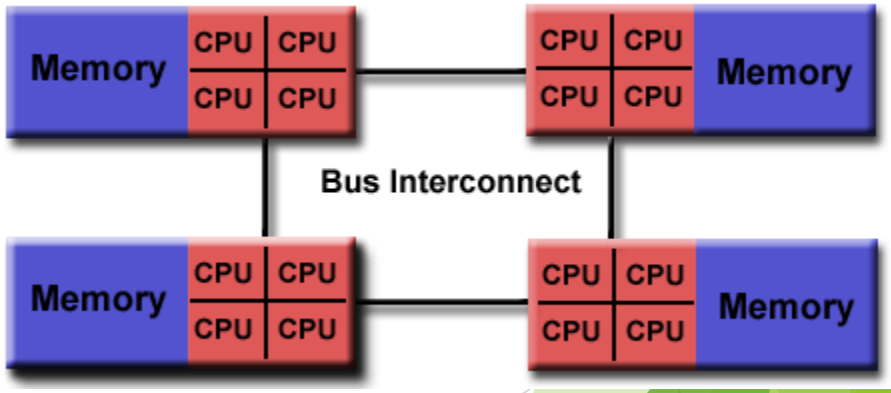
Threads

- Each Process consists of multiple independent instruction streams(or threads) that are assigned compute resource by some scheduling procedure
- Threads of a process share the address space of this process
  - Global variables and all dynamically allocated data objects are accessible by all the threads of a process
- Each thread has its own stack, register set, program counter
- Threads can communicate by reading/writing variables in the common address space

Memory Model

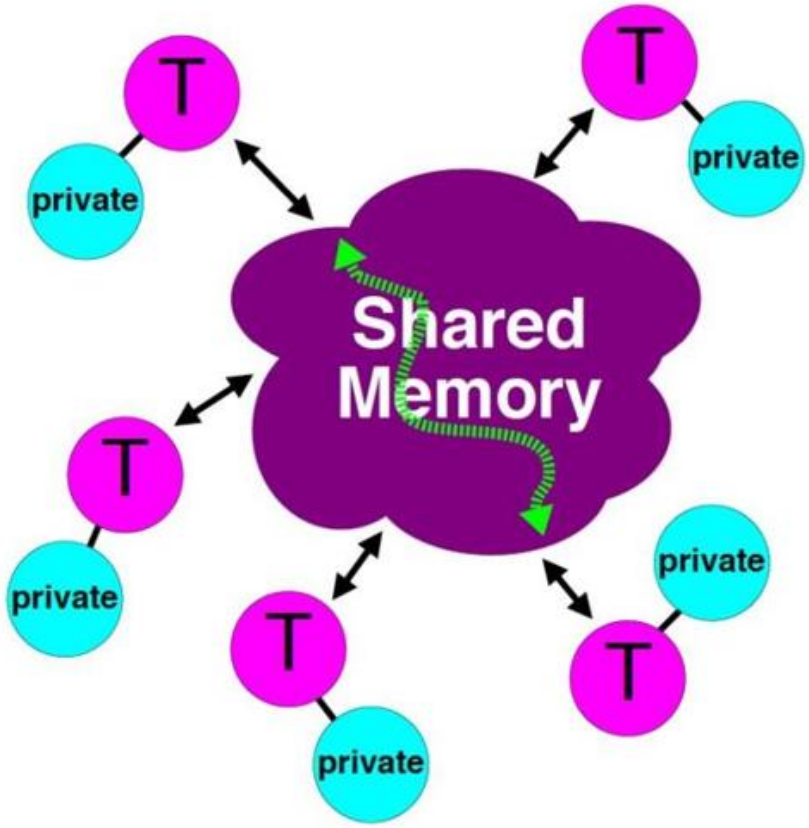
- OpenMP is designed for multi-process/shared memory machines. The architecture can be shared memory UMA or NUMA.



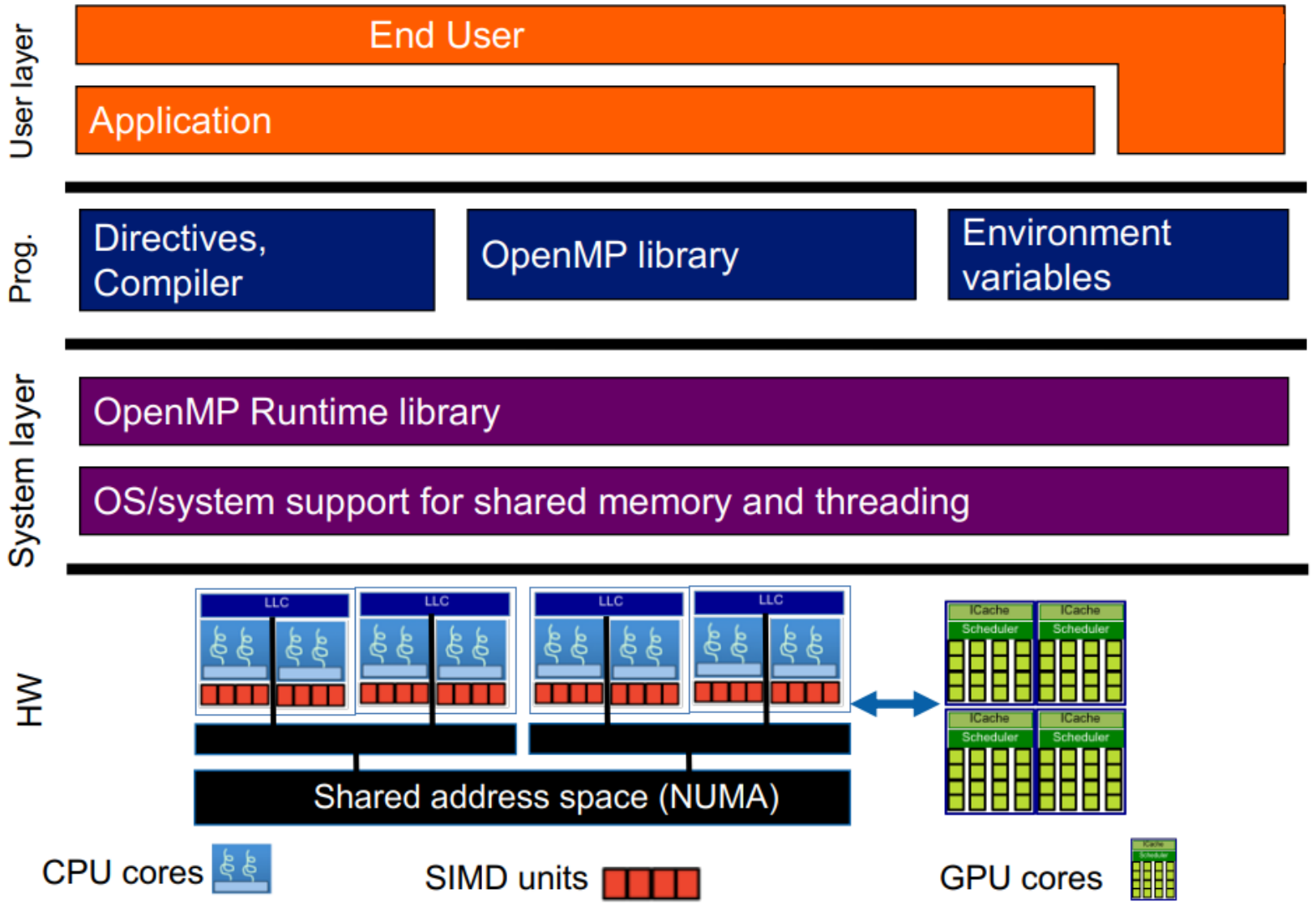


Memory Model

- Data is private or shared.
- Private accessed only by owned threads.
- Shared data accessible by all threads
- Data transfer is transparent to programmer
- Synchronization takes place.



OpenMP stack



Sequential Code

```
#include <stdio.h>
void main()
{
	int ID;
	printf(“Hello my ID is :%d\n”,ID);
}
```

Parallel Code

```
#include <stdio.h>
#include <….>
void main()
{
        <……>
        {
        int ID= <……>;
        printf(“Hello my ID is :%d\n”,ID);
        }
}
```

Parallel Code

```
#include <stdio.h>
#include <omp.h>
void main()
{
    #pragma omp parallel
    {
        int ID= omp_get_thread_num();
        printf("Hello my ID is :%d\n",ID);
    }
}
```

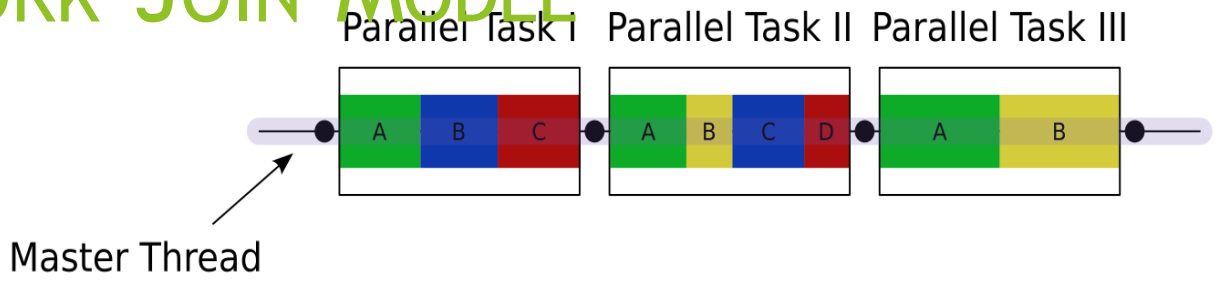
Compiler switches

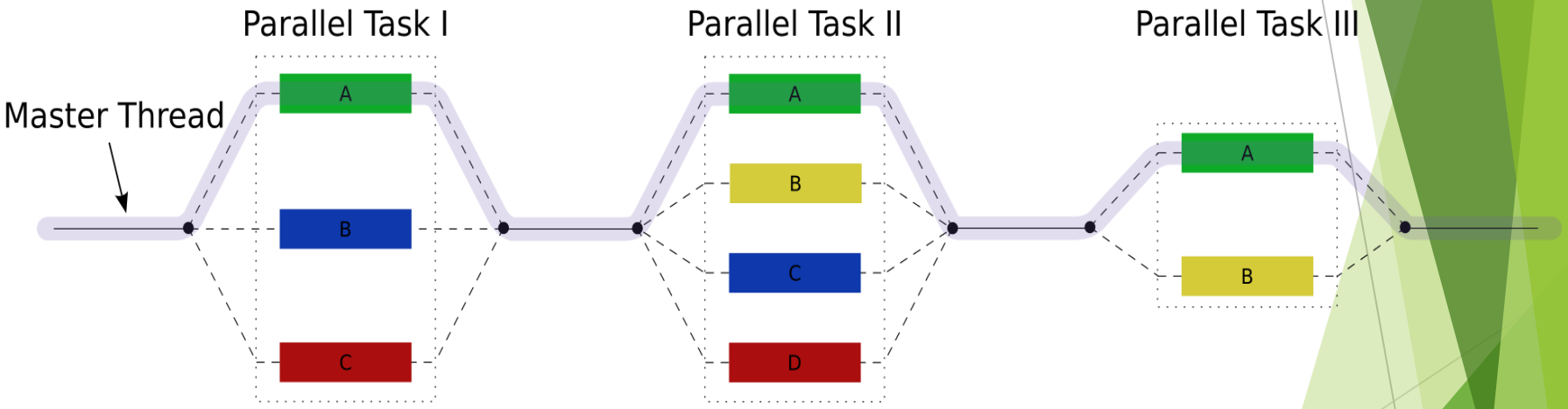
- GNU Compiler Example
  - gcc –fopenmp program.c –o <output_name>
  - g++ -fopenmp program.cpp –o <output_name>
  - ./<output_name>
- Intel Compiler Example
  - icc – o omp_helloc –openmp omp_hello.c

OpenMP Execution Model

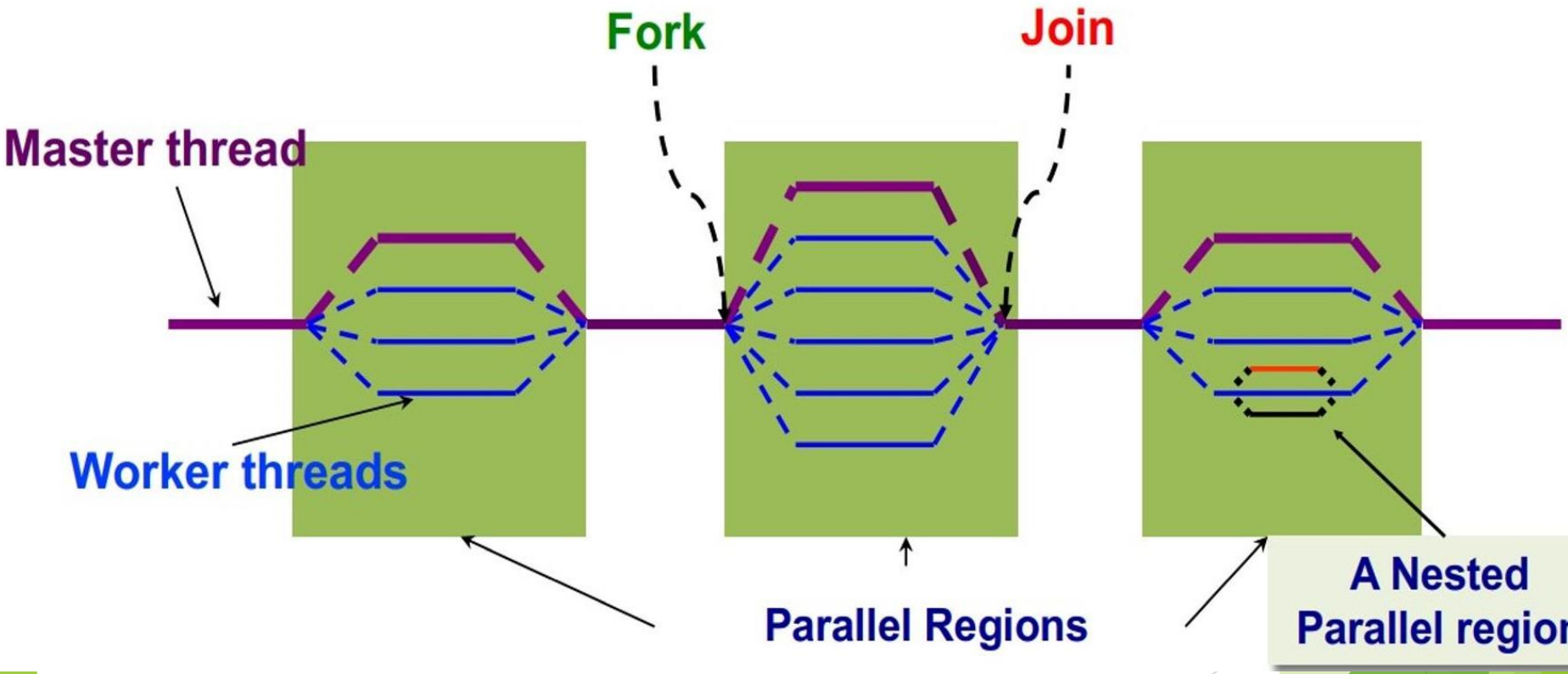
- Shared Memory, Thread Based Parallelism- use of threads
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm
- Explicit Parallelism- explicit programming model
  - full control to developers
- Compiler Directive Based
  - Most OpenMP parallelism is specified through the use of compiler directives embedded in code
- Fork - Join Model- of parallel execution

Fork-Join Model





FORK-JOIN MODEL



How do threads interact?

- OpenMP is an multi-threading, shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes 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 accessed to minimize the need for synchronization.

General Syntax

- Header file
  - #include <omp.h>
- Parallel Region

```
#pragma    omp   parallel [clauses…]
{
   // … do some work here
} // end of parallel region/block
```

- **Functions and Environment variables**

An Example – Hello World

Sequential

```
void main( )
{
        int ID = 0;
        printf(“ Thread: %d - Hello World ”, ID);
}
```

OpenMP

OpenMP include file

```
#include <omp.h>
void main( )
{
        #pragma omp parallel
        {
                int ID = 0;
                printf(“ Thread: %d - Hello World ”, ID);
        }
}
```

Parallel region with default Number of threads

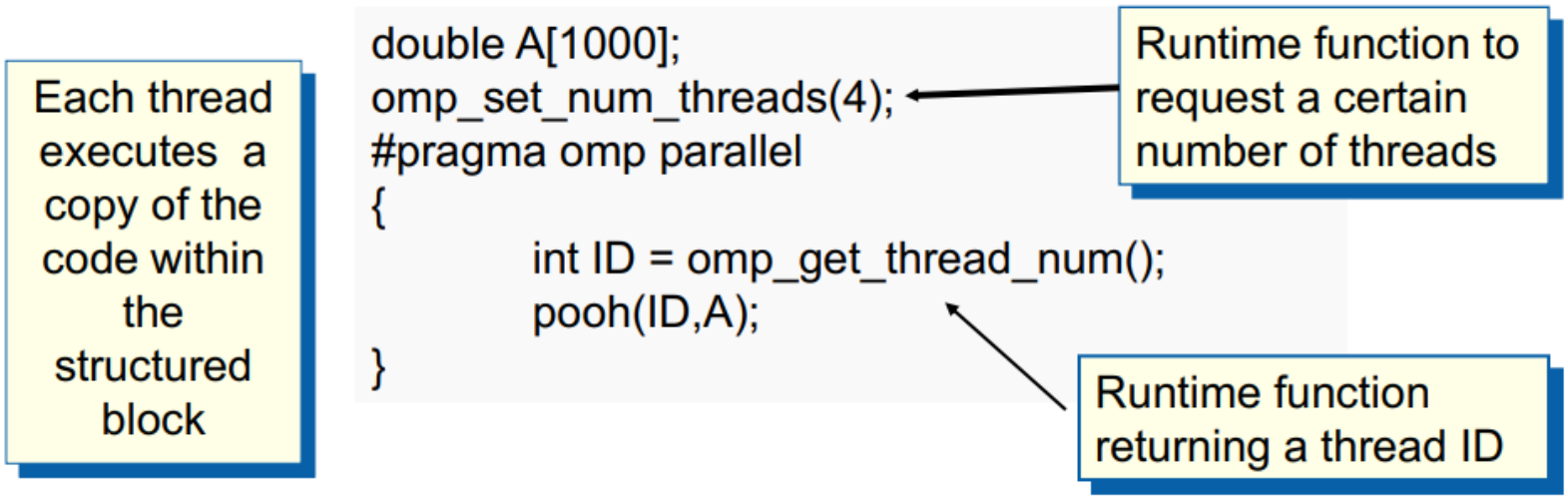
End of parallel region

COMPILATION

- GNU Compiler Example :
  - gcc -o helloc.x -fopenmp hello.c
- IBM AIX compiler :
  - xlc -o helloc.x -qsmp=omp hello.c
- Portland group compiler:
  - pgcc -o helloc.x -mp hello.c
- Intel Compiler Example:
  - icc -o helloc.x -openmp hello.c

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.
- For example, to create a 4 thread Parallel region:



- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread creation: How many threads did you actually get?

- Request a number of threads with omp_set_num_threads()
- The number requested may not be the number you actually get.
  - An implementation may silently give you fewer threads than you requested.
  - Once a team of threads has launched, it will not be reduced.

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID      = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

- Each thread calls pooh(ID,A) for ID = 0 to nthrds-1

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.

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