A “Hands-on” Introduction to OpenMP*

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

Outline

- Unit 1: Getting started with OpenMP
  - Mod1: Introduction to parallel programming
  - Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - Disc 1: Hello world and how threads work
- Unit 2: The core features of OpenMP
  - Mod 3: Creating Threads (the Pi program)
  - Disc 2: The simple Pi program and why it sucks
  - Mod 4: Synchronization (Pi program revisited)
  - Disc 3: Synchronization overhead and eliminating false sharing
  - Mod 5: Parallel Loops (making the Pi program simple)
  - Disc 4: Pi program wrap-up
- Unit 3: Working with OpenMP
  - Mod 6: Synchronize single masters and stuff
  - Mod 7: Data environment
  - Disc 5: Debugging OpenMP programs
  - Mod 8: Skills practice … linked lists and OpenMP
  - Disc 6: Different ways to traverse linked lists
- Unit 4: a few advanced OpenMP topics
  - Mod 8: Tasks (linked lists the easy way)
  - Disc 7: Understanding Tasks
  - Mod 8: The scary stuff … Memory model, atomics, and flush (pairwise synch).
  - Disc 8: The pitfalls of pairwise synchronization
  - Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - Disc 9: Random number generators
- Unit 5: Recapitulation

Exercise 1: Solution

A multi-threaded “Hello world” program

- **Write a multithreaded program where each thread prints “hello world”.**

```
#include “omp.h”
int  main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);;
    printf(“ world(%d) \n”, ID);;
  }
}
```

OpenMP include file

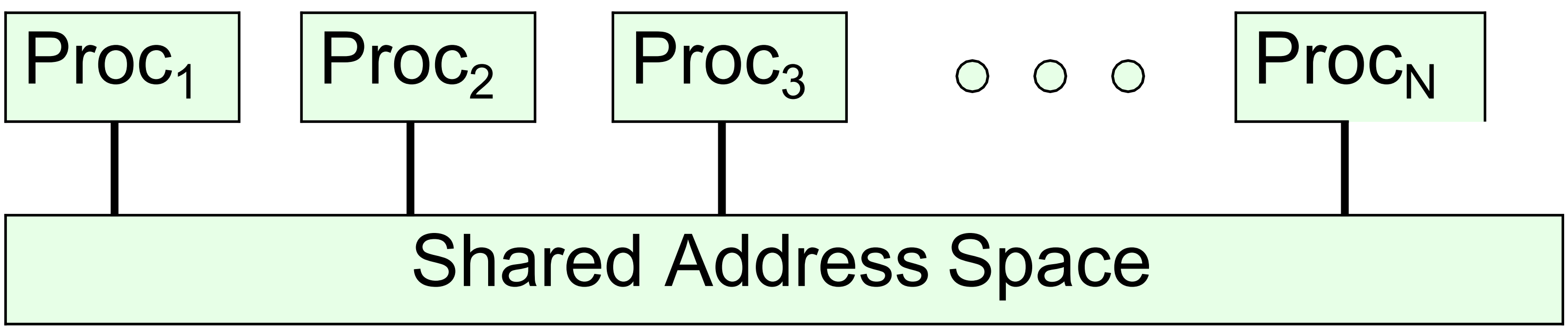
Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

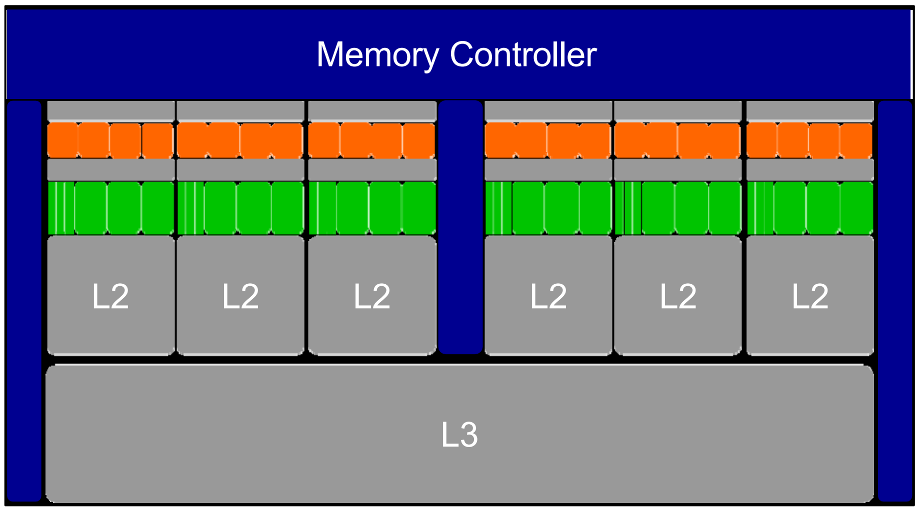
Shared memory Computers

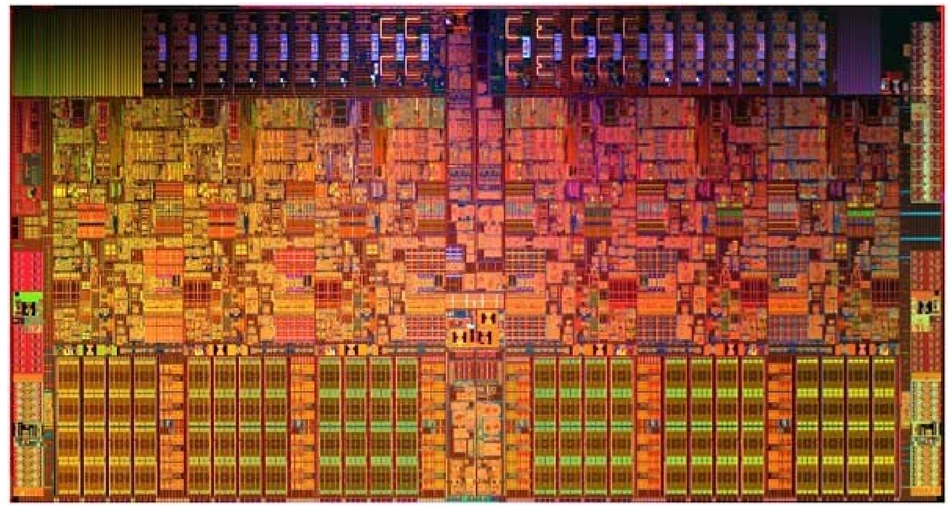
- **Shared memory computer** : any computer composed of multiple processing elements that share an address space. Two Classes:
  - **Symmetric multiprocessor** (**SMP**): a shared address space with "equal-time" access for each processor, and the OS treats every processor the same way.
  - **Non Uniform address space multiprocessor** (**NUMA**): different memory regions have different access costs … think of memory segmented into "Near" and "Far" memory.



Shared memory machines: SMP

Intel® Core™ i7-970 processor: Often called an SMP, but is it?





- 6 cores, 2-way multithreaded, 6-wide superscalar, quad-issue, 4-wide SIMD (on 3 of 6 pipelines)
- 4.5 KB (6 x 768 B) "Architectural" Registers, 192 KB (6 x 32 KB) L1 Cache, 1.5 MB (6 x 256 KB) L2 cache, 12 MB L3 Cache
- MESIF Cache Coherence, Processor Consistency Model
- 1.17 Billion Transistors on 32 nm process @ 2.6 GHz

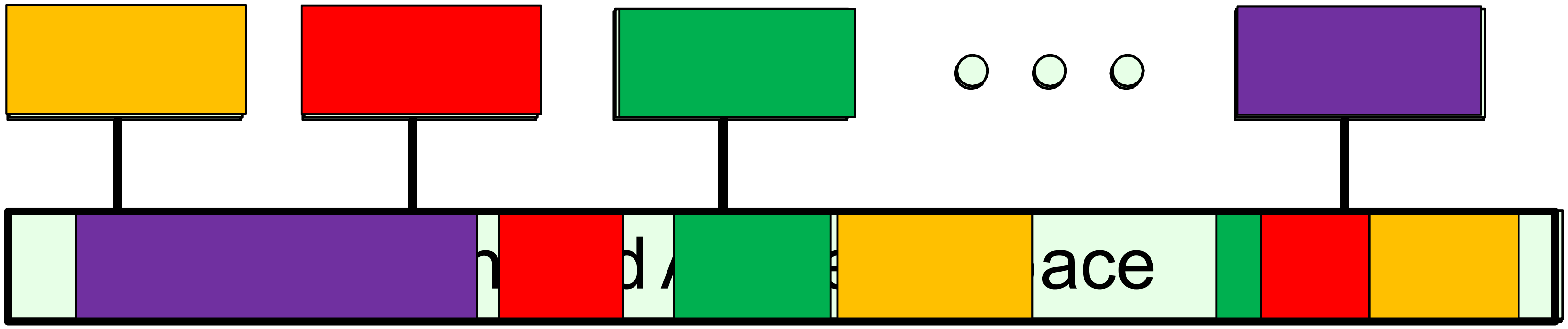
Cache hierarchy means different processors have different costs to access different address ranges …. It's NUMA

Shared memory computers

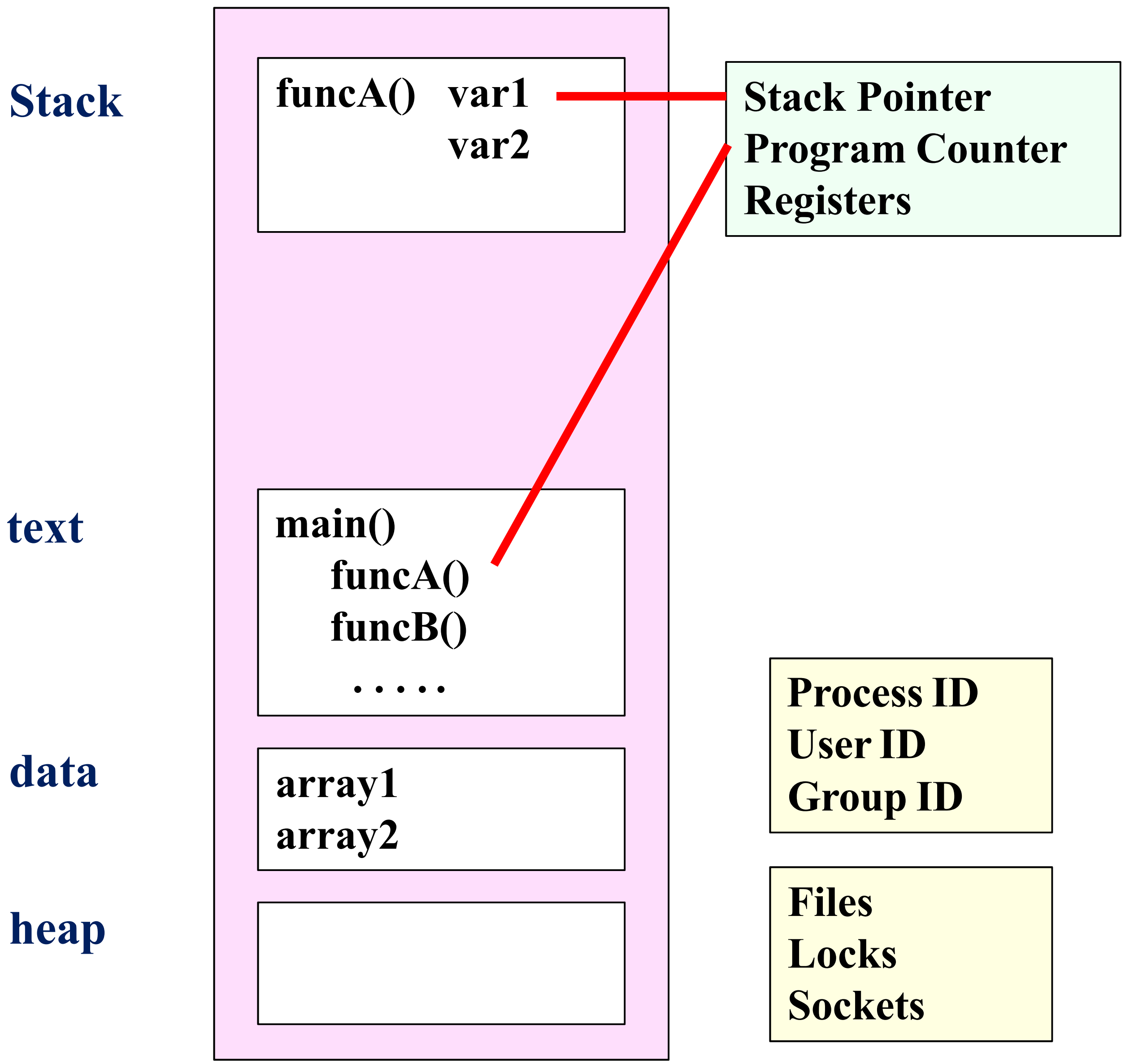
- Shared memory computers are everywhere … most laptops and servers have multicore multiprocessor CPUs

- The shared address space and (as we will see) programming models encourage us to think of them at SMP systems.

- Reality is more complex … any multiprocessor CPU with a cache is a NUMA system. Start out by treating the system as an SMP and just accept that much of your optimization work will address cases where that case breaks down.



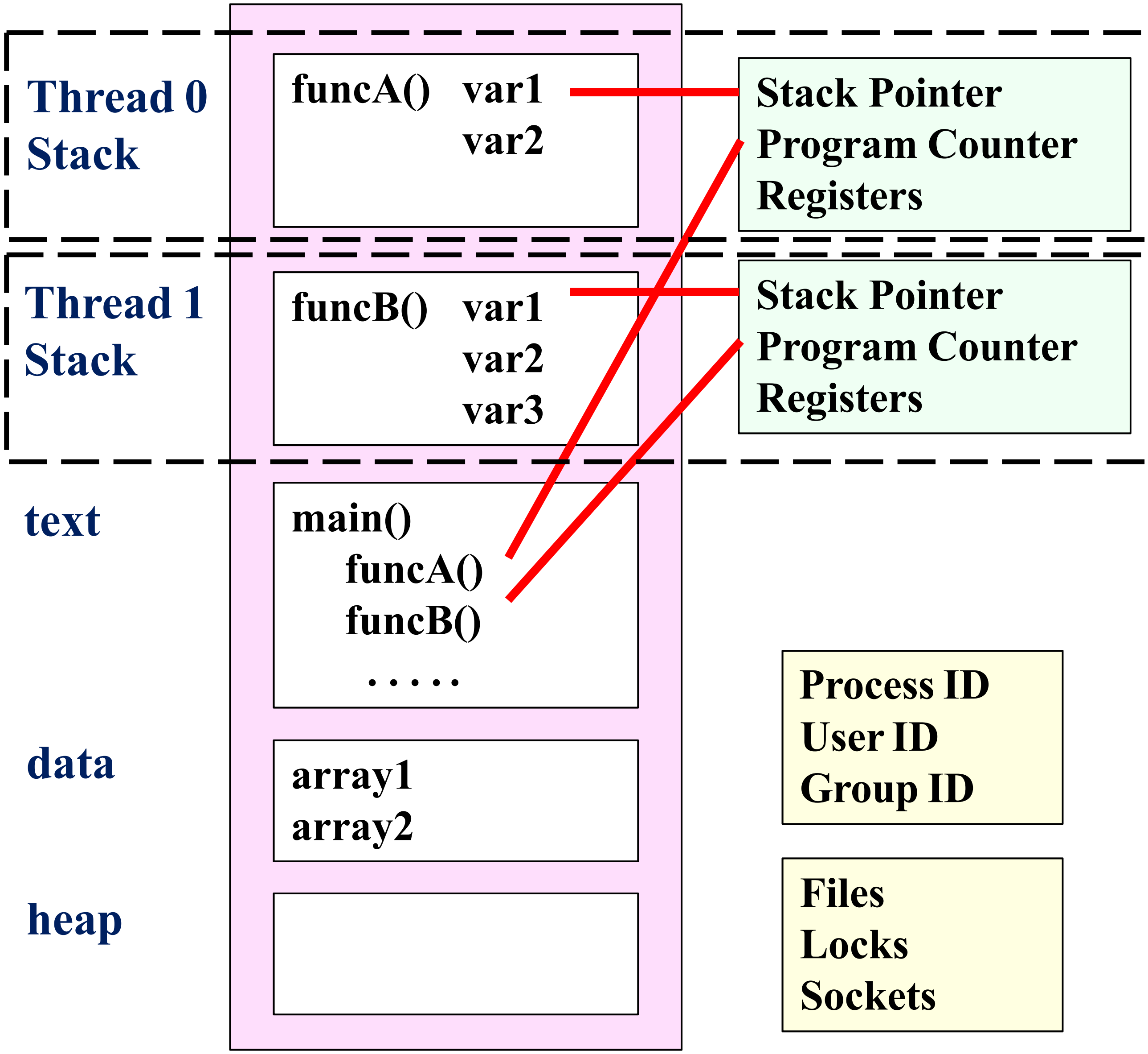
Programming shared memory computers



Process

- An instance of a program execution.
- The execution context of a running program … i.e. the resources associated with a program's execution.

Programming shared memory computers

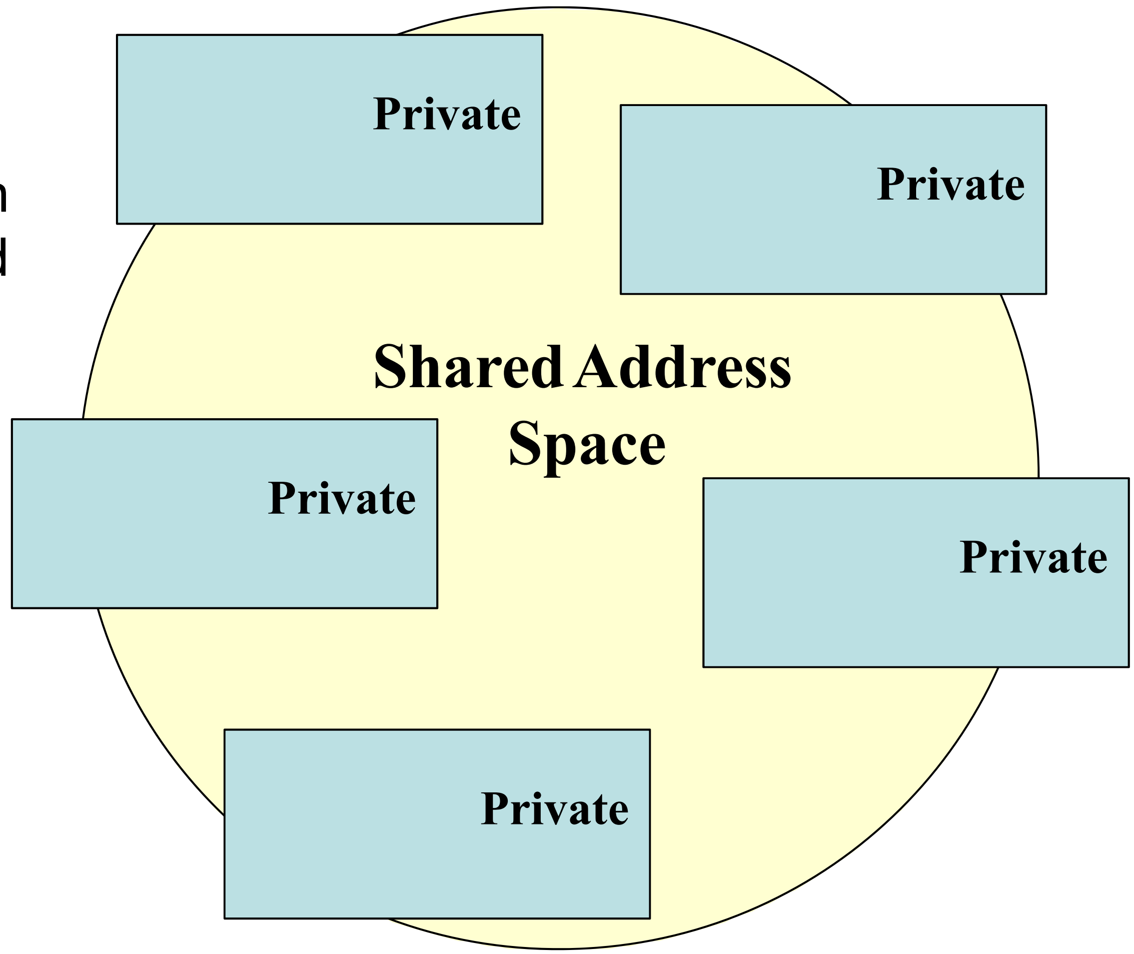


Threads:

- Threads are "light weight processes”
- Threads share Process state among multiple threads … this greatly reduces the cost of switching context.

A shared memory program

- **An instance of a program:**
  - One process and lots of threads.
  - Threads interact through reads/writes to a shared address space.
  - OS scheduler decides when to run which threads … interleaved for fairness.
  - Synchronization to assure every legal order results in correct results.



Exercise 1: Solution
A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints "hello world".**

```
#include "omp.h"
int  main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);;
    printf(" world(%d) \n", ID);;
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

OpenMP Overview:
How do threads interact?

- OpenMP is a multi-threading, shared address 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.