A “Hands-on” Introduction to to parallel Programming (with OpenMP*)

Tim Mattson

Intel Corporation

timothy.g.mattson@intel.com

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

Preliminaries: part 1

- Disclosures
  - The views expressed in this tutorial are my own.
    - I am <u>not</u> speaking for my employer.
    - I am <u>not</u> speaking for the OpenMP ARB
- I take my tutorials VERY seriously:
  - Help me improve … let me know if you have ideas on how to improve this content.

Preliminaries: Part 2

- Our plan … Active learning!
  - We will mix short lectures with short exercises.
- Please follow these simple rules
  - Do the exercises I assign and then change things around and experiment.
    - Embrace active learning!
  - **Don’t cheat**: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Our Plan: Day 1

Topic	Exercise	concepts
Intro to parallel programming	No exercise	Basic concepts and the jargon of parallel programming
OMP Intro	Install sw, hello_world	Parallel regions
Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
Synchronization	Pi_spmd_final	False sharing, critical, atomic
Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
The rest of worksharing and synchronization	No exercise	Single, sections, master, runtime libraries, environment variables, synchronization, etc.
Data Environment	Molecular Dyn.	Data environment details, software optimization

Break

Lunch

Our Plan: day 2

Topic	Exercise	concepts
Review material from yesterday	No exercise	Make sure parallel, worksharing and the data environment are understood by all.
OpenMP tasks	Linked list (tasks) Linked list (no tasks)	OpenMP tasks
Memory model	Producer-Consumer	The need for flush
A survey of parallel programming models	No exercise	Cilk, MPI, OpenCL, TBB, etc.

Break

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment

- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

Agenda – parallel theory

- How to sound like a parallel programmer
- An overly brief look at parallel architecture
- Understanding design patterns for parallel programming

The foundation of parallel computing

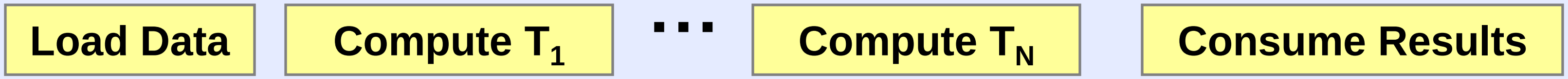
- Concurrency: when multiple tasks are active and able to make progress (in principle) at the same time.
    - Concurrency is a general idea – even on single processor systems inside the OS.

- Two ways to use concurrency
    - Parallel computing – when concurrency is used to make a job run faster.
        - The problem being solved "makes sense" as a serial program … for example, A parallel molecular dynamics program.

    - Concurrent computing – when the concurrency is used to manage availability or reduce latencies for multiple agents.
        - The "job" in question is fundamentally concurrent … there is no reasonable serial analog … for example, a print server.

An Example of Parallel Computing

Compute N independent tasks on one processor



Load Data

Compute T_1

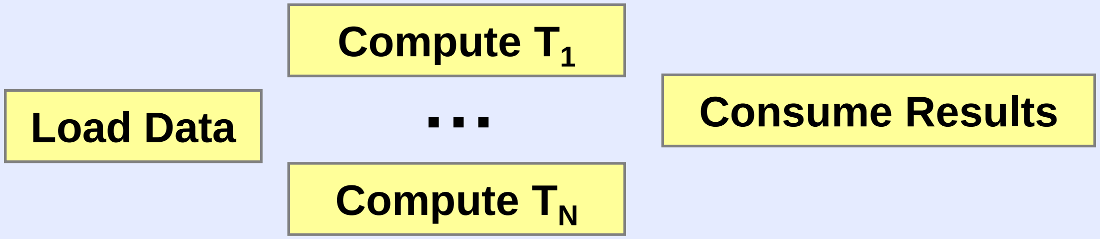
…

Compute T_N

Consume Results

$$\text{Time}_{seq}(1) = T_{load} + N*T_{task} + T_{consume}$$

Compute N independent tasks with P processors



Load Data

Compute T_1

…

Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

(Note: Parallelism only speeds-up the concurrent part)

$$\text{Time}_{par}(P) = T_{load} + (N/P)*T_{task} + T_{consume}$$

Talking about performance

- Speedup: the increased performance from running on P processors.

$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

- Super-linear Speedup: typically due to cache effects … i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

$$S(P) > P$$

Amdahl’s Law

- What is the maximum speedup you can expect from a parallel program?
- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

- If serial_fraction is α and parallel_fraction is (1- α) then the speedup is:

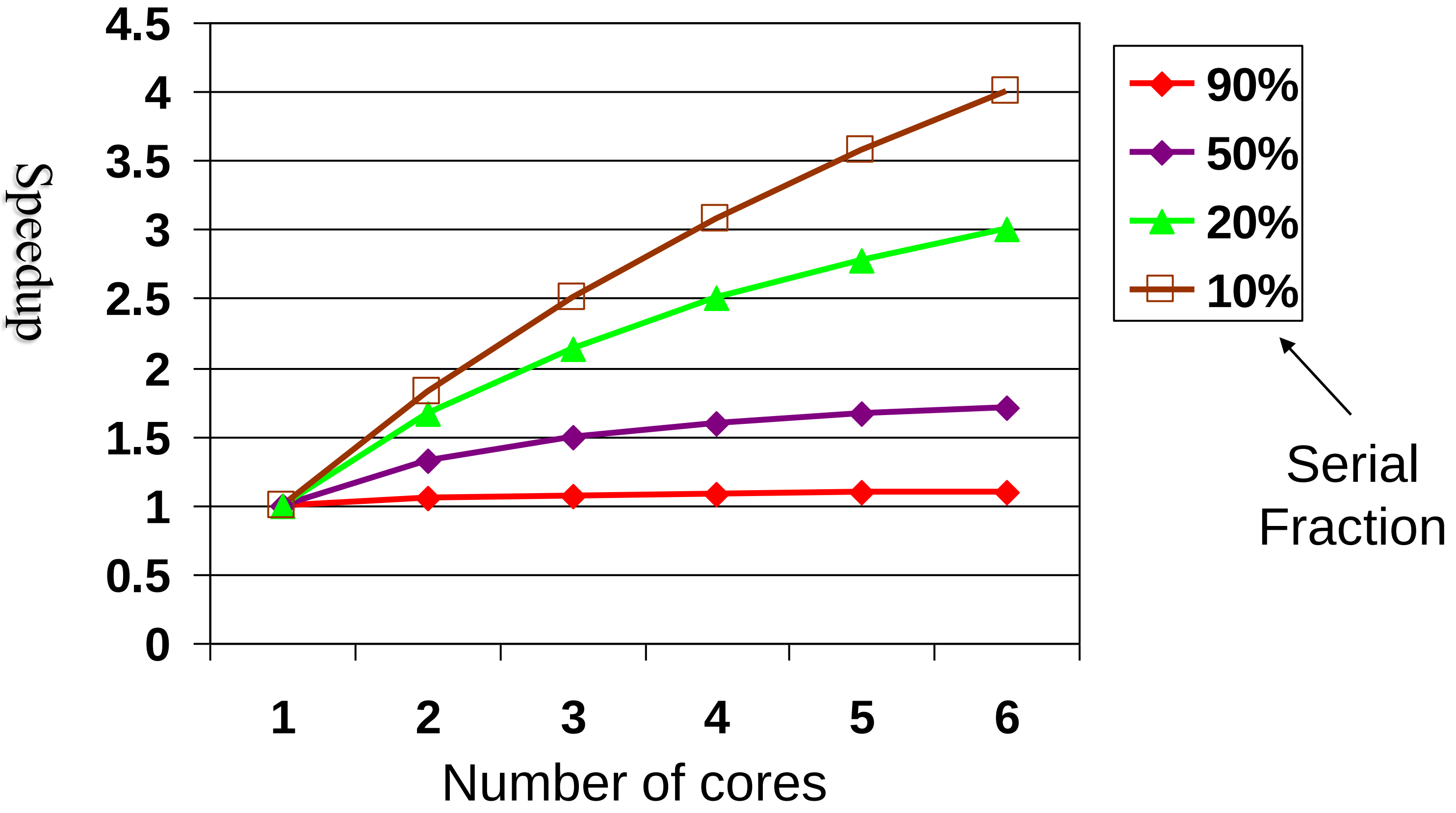
$$S(P) = \frac{Time_{seq}(1)}{(\alpha + \frac{1-\alpha}{P}) * Time_{seq}(1)} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

- If you had an unlimited number of processors: $P \longrightarrow \infty$

- The maximum possible speedup is: $S = \frac{1}{\alpha} \longleftarrow \boxed{\text{Amdahl's Law}}$

Implications of Amdahl’s Law

- Consider benefits of adding processors to your parallel program for different serial fractions.
- Note: getting a serial fraction under 10% is challenging for the typical application



Acknowledgement: , 2009 DAC Tutorial – Tom Spyrou

Granularity

- Granularity is the ratio of compute time to communication time
  - □ Hardware: raw compute rate vs. communication rate or memory latency
  - □ Software: Consider time spent in local computations vs. time spent updating state between computing agents.
    - Single channel Seismic codes and rendering programs are coarse grained.
    - Unstructured mesh codes tend to be fine grained

Key rule: Granularity demanded by software must be met or bettered by hardware. Fine grained applications do not run well on coarse grained systems.

Load Balancing

- Load Balancing: The distribution of work among the processors of a parallel computer:
  - static load balancing: distribution deterministic and setup at program startup.
  - dynamic load balancing: distribution changes as the calculation proceeds.

Overall performance depends on the processor that takes the longest time.

Top Performance requires that all processors are equally loaded.

Fooling the masses with performance results on parallel computers

- Compare 32 bit results on the machine you like (e.g. a GPU) to 64 bit results on the machine you "don't like" (e.g. a CPU).
- Present results for a highly tuned inner kernel and then suggest the results reflect performance for the full application.
- Use aggressive tuning (assembly code) on the system you like.
- Report speedups comparing a great parallel algorithm to a poor serial algorithm (or call the parallel algorithm running on one core your serial algorithm).
- Exclude memory movement costs … warm your caches and load local memory before starting the clock (a common trick by people pushing GPGPU programming and accelerators).

Inspired by David Bailey's classic paper "Twelve Ways to Fool the Masses when giving performance results on parallel computers", Supercomputing Review, Aug 1991, pp. 54-55. http://crd.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf

recap

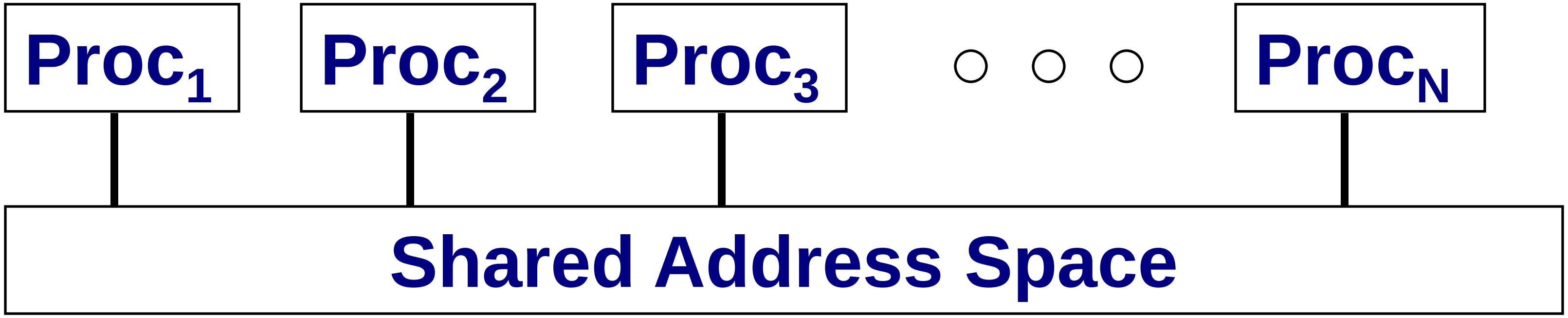
- So now you know how to sound like a parallel programmer.
- Essential issues are:
  - □ Finding enough concurrency to meet desired scalability targets.
  - □ Balance the load carefully since the slowest core determines the overall runtime.
  - □ Minimize serial fraction in your problem and keep parallel overhad low … or Amdahl’s law will get you.
  - □ Learn how to use performance results to mislead people (a useful skill when annual review time comes around).
  - □ Parallel software is the key challenge
    - Find concurrency
    - Structure your algorithm to exploit concurrency
    - Express concurrency in source code
    - Run on a parallel computer.

Agenda – parallel theory

- How to sound like a parallel programmer
- ➡ An overly brief look at parallel architecture
- Understanding design patterns for parallel programming

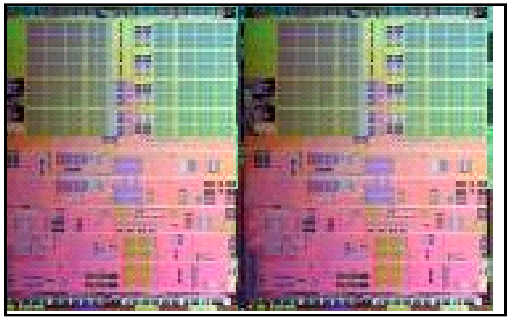
How do we connect cores together?

- A symmetric multiprocessor (SMP) consists of a collection of processors that share a single address space:
  - Multiple processing elements.
  - A shared address space with “equal-time” access for each processor.
  - The OS treats every processor the same



How realistic is this model?

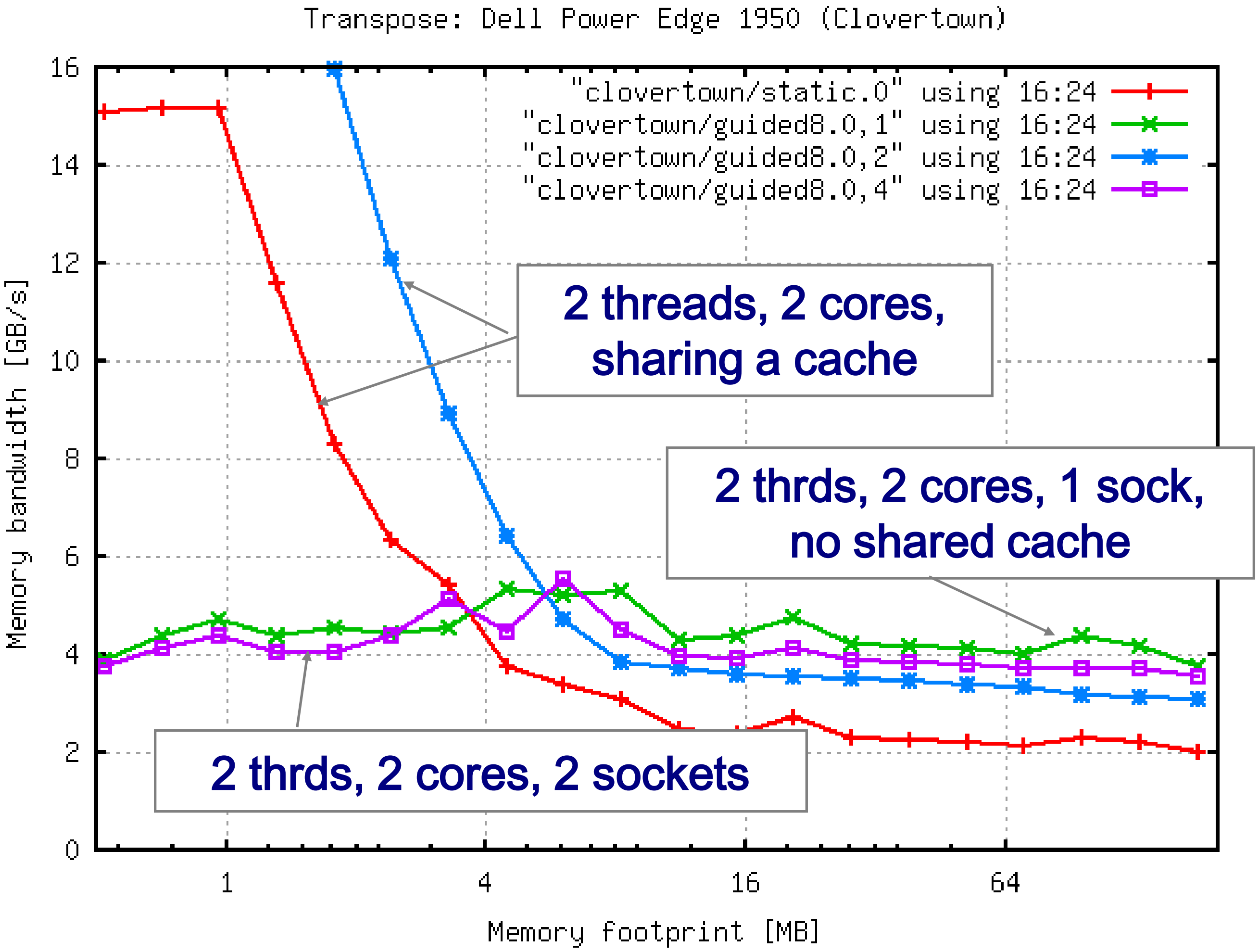
- Some of the old supercomputer mainframes followed this model,



A CPU with lots of cache …

- But as soon as we added caches to CPUs, the SMP model fell apart.
  - Caches … all memory is equal, but some memory is more equal than others.

NUMA issues on a Multicore Machine
2-socket Clovertown Dell PE1950

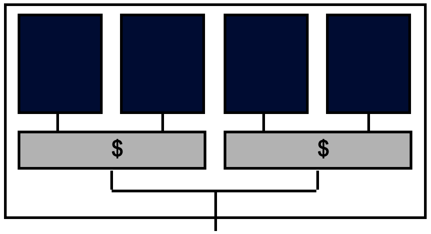


2 threads, 2 cores, sharing a cache

2 thrds, 2 cores, 1 sock, no shared cache

2 thrds, 2 cores, 2 sockets

A single quad-core chip is a NUMA machine!



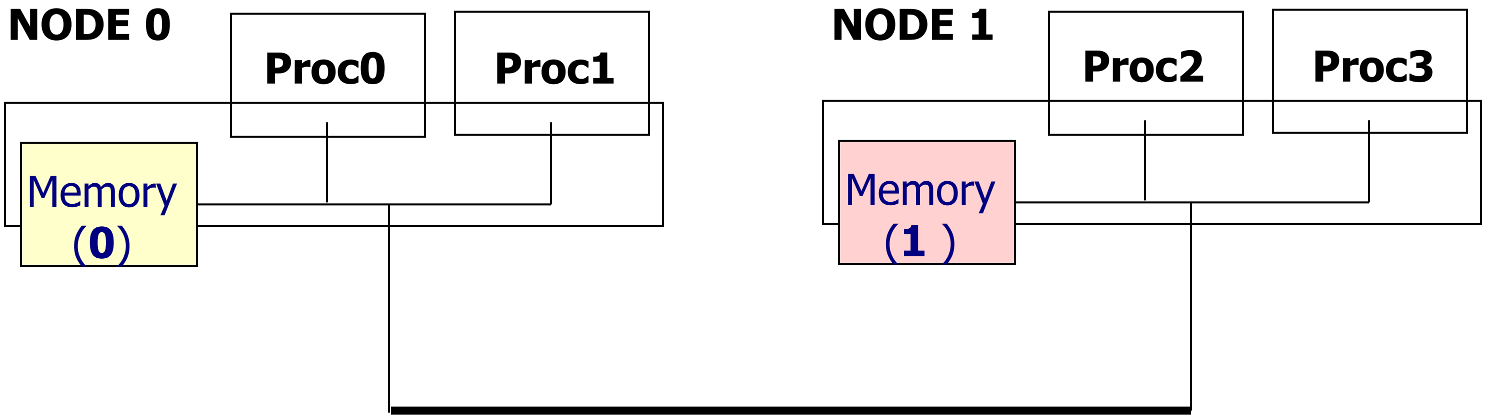
Xeon® 5300 Processor block diagram

Third party names are the property of their owners.

Source Dieter an Mey, IWOMP'07 face to face meeting

Put these into a larger system and it only get's worse

- Consider a typical NUMA computer:



- Memory access takes longer if memory is remote.
- For example, on an SGI Altix:

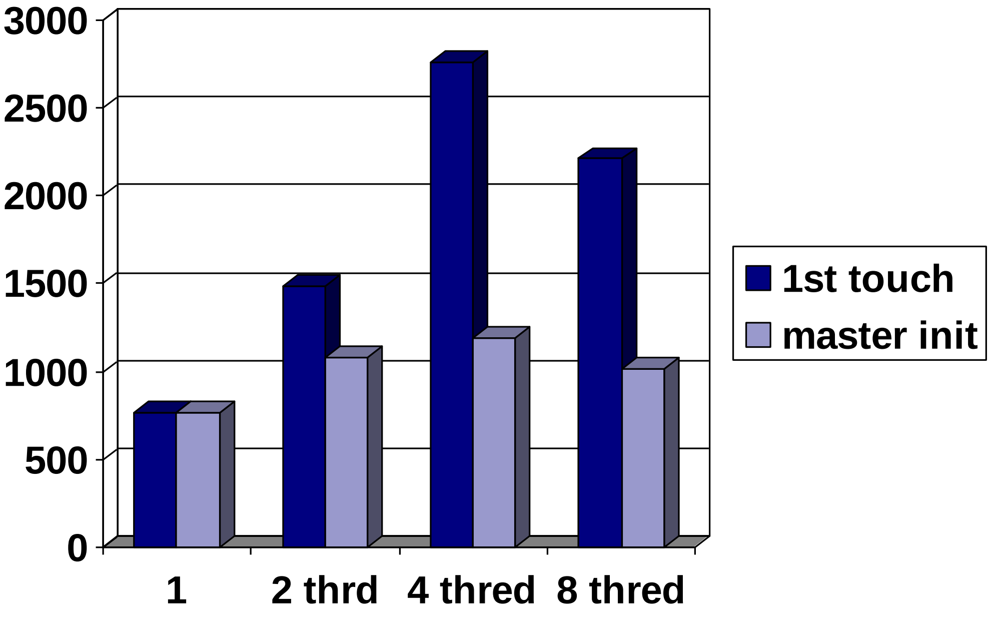
•Proc0 to local memory (0)	207 cycles
•Proc0 to remote memory (1)	409 cycles

Source: J. Marathe & F. Mueller, Gelato ICE, April 2007.

Surviving NUMA: initializing data

- Keep data close to where it is needed:
  - Bind threads to cores.
  - Iniitialize the data so its near the core that will use it.
- Test problem: Jacobi from www.openmp.org, with 2000x2000 matrix.
- Hardware: a 4-socket machine with dualcore Opteron processors with processor binding enabled.

MFLOPS vs. number of threads

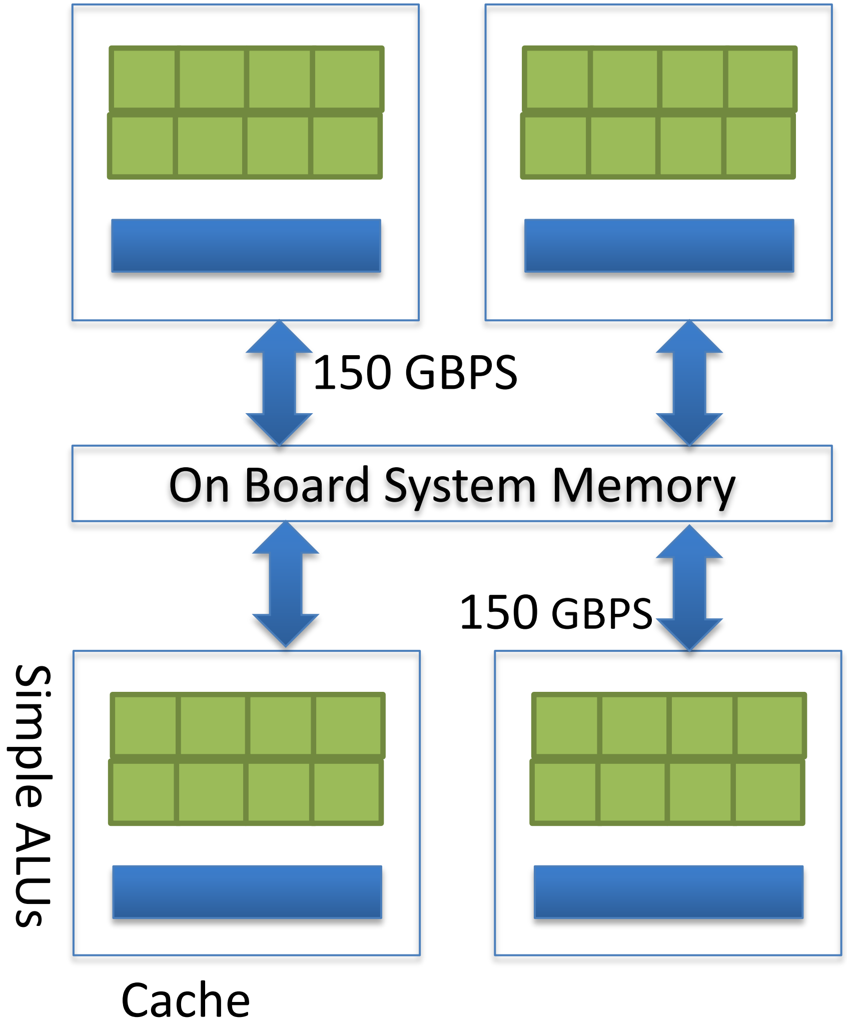


Source Dieter an Mey, IWOMP'07 face to face meeting

Third party names are the property of their owners.

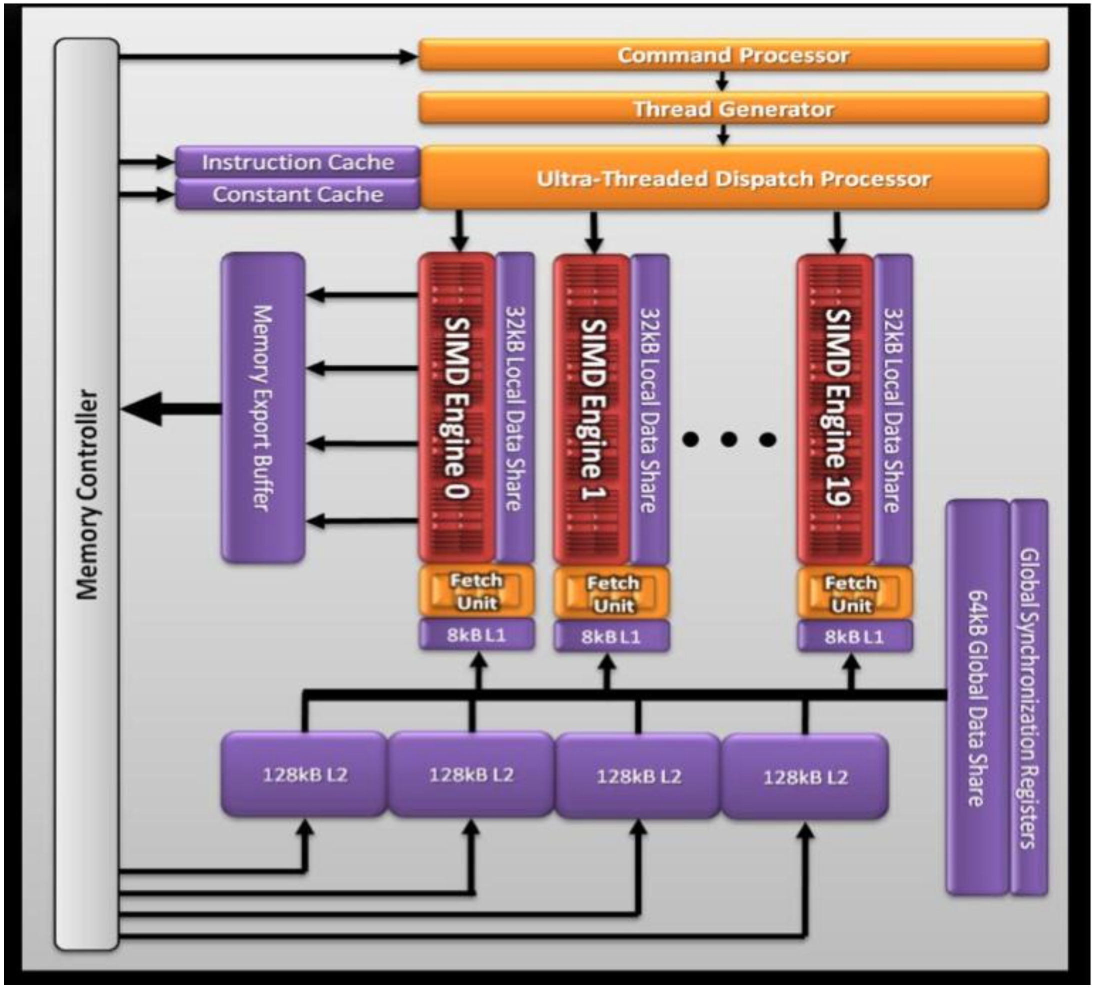
Modern GPGPU Architecture

- Generic many core GPU
- Less space devoted to control logic and caches
- Large register files to support multiple thread contexts
- Low latency hardware managed thread switching
- Large number of ALU per "core" with small user managed cache per core
- Memory bus optimized for bandwidth



Source: Advanced topics in heterogeneous computing with OpenCL, *SC10* Benedict R. Gaster

AMD GPU Hardware Architecture

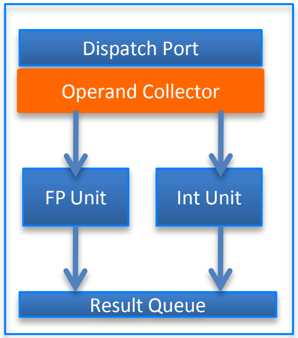


- AMD 5870 – Cypress
- 20 SIMD engines
- 16 SIMD units per core
- 5 multiply-adds per functional unit (VLIW processing)
- 2.72 Teraflops Single Precision
- 544 Gigaflops Double Precision

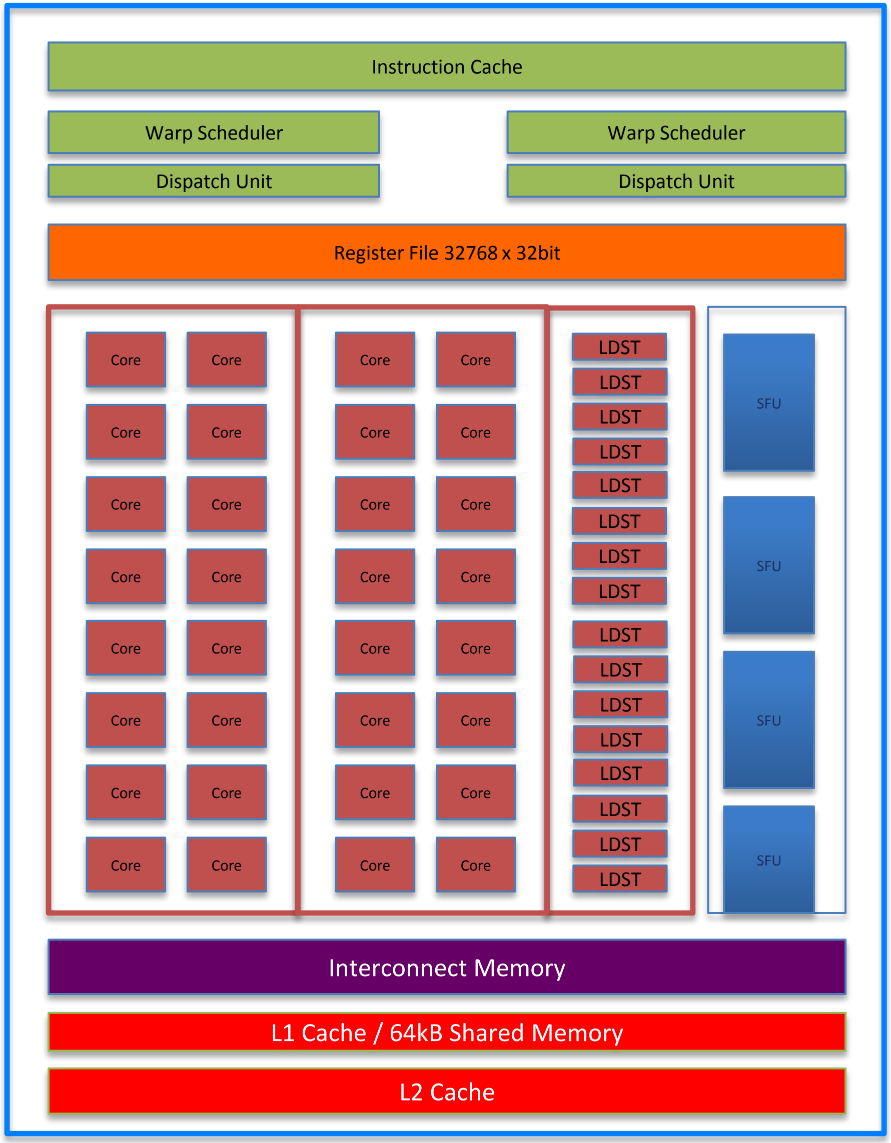
Source: Introductory OpenCL *SAAHPC2010,* Benedict R. Gaster

Nvidia GPUs - Fermi Architecture

- GTX 480 - Compute 2.0 capability
  - 15 cores or Streaming Multiprocessors (SMs)
  - Each SM features 32 CUDA processors
  - 480 CUDA processors
- Global memory with ECC



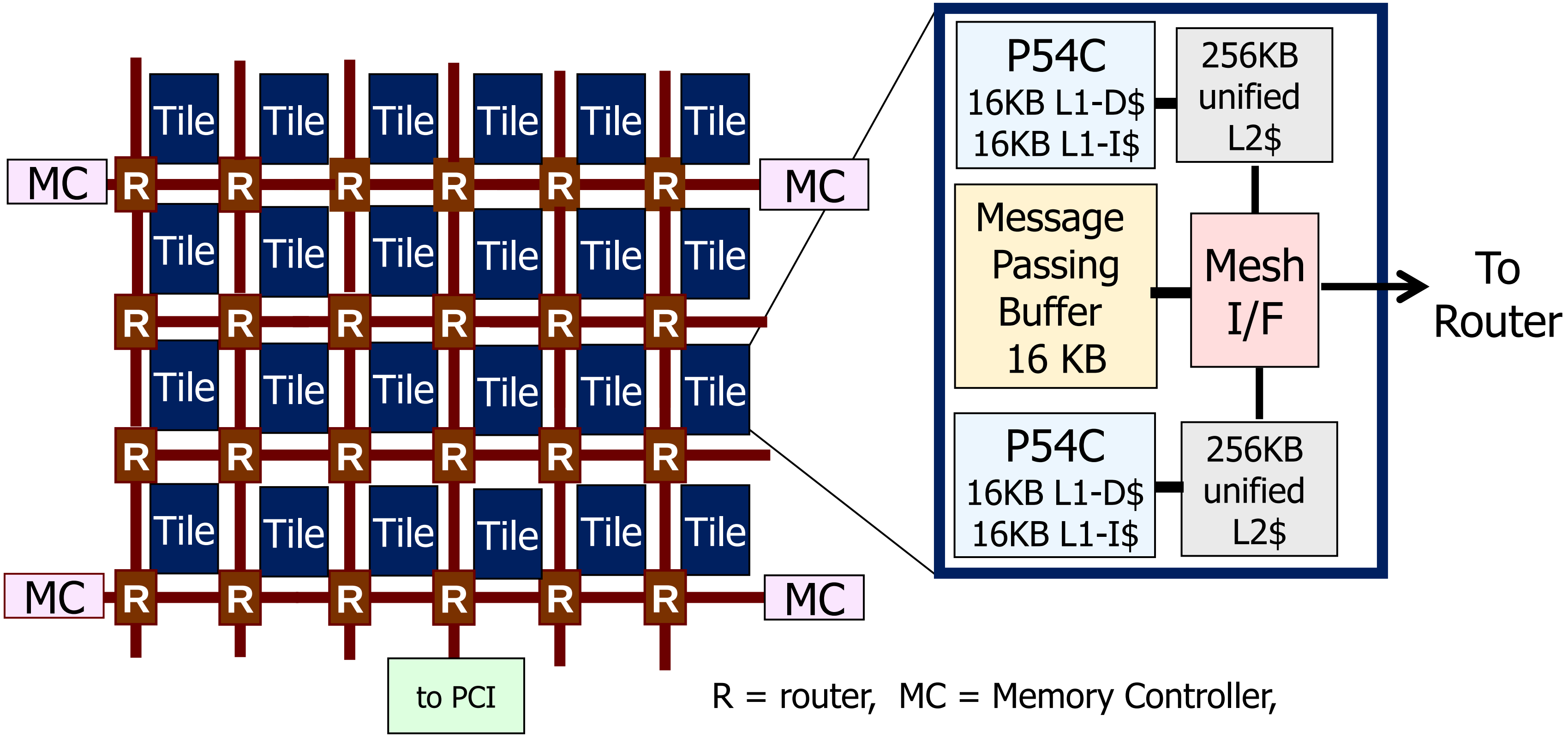
Source: NVIDIA's Next Generation CUDA Architecture Whitepaper



Hardware view of SCC

- 48 P54C cores, 6x4 mesh, 2 cores per tile
- 45 nm, 1.3 B transistors, 25 to 125 W
- 16 to 64 GB DRAM using 4 DDR3 MCs
- 2 Tb/s bisection bandwidth @ 2 Ghz

Technology	**45nm Process**
Transistors	**Die: 1.3B Tile: 48M**
Die Area	**567.1mm^2**



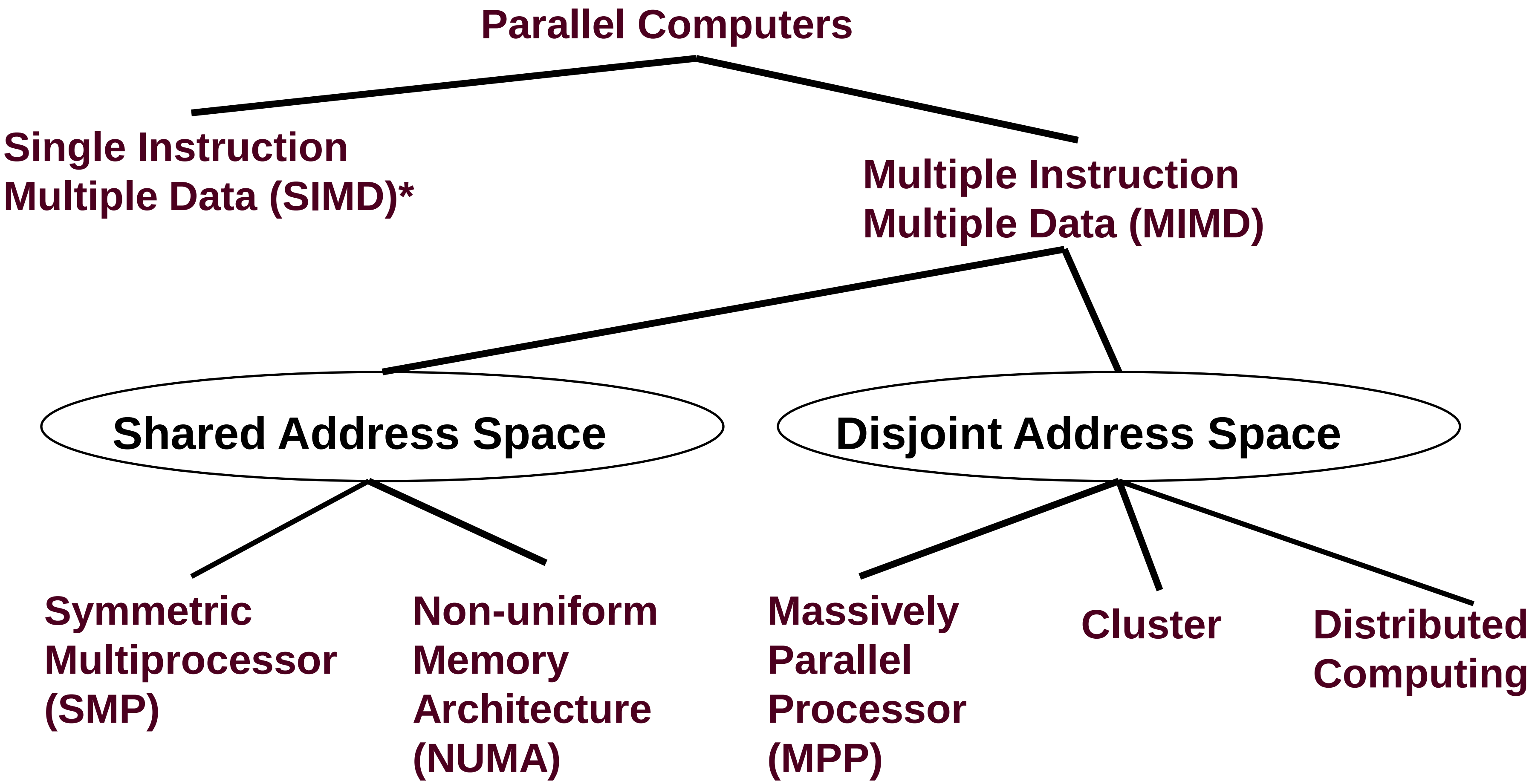
R = router, MC = Memory Controller,

P54C = second generation Pentium® core, CC = cache cntrl.

Moving “beyond the single board”

- Parallel computers are classified in terms of streams of data and streams of instructions:
  - MIMD Computers: Multiple streams of instructions acting on multiple streams of data.
  - SIMD Computers: A single stream of instructions acting on multiple streams of data.
- Parallel Hardware comes in many forms:
  - On chip: Instruction level parallelism (e.g. IPF)
  - Multiprocessor: Multiple processors inside a single computer.
  - Multicomputer: networks of computers working together.

Hardware for parallel computing



Parallel Computers

Single Instruction Multiple Data (SIMD)*

Multiple Instruction Multiple Data (MIMD)

Shared Address Space

Disjoint Address Space

Symmetric Multiprocessor (SMP)

Non-uniform Memory Architecture (NUMA)

Massively Parallel Processor (MPP)

Cluster

Distributed Computing

***SIMD has failed as a way to organize large scale computers with multiple processors. It has succeeded, however, as a mechanism to increase instruction level parallelism in modern microprocessors (MMX, SSE, AVX, etc.).**

Examples: SIMD MPP

"… we want to build a computer that will be proud of us", Danny Hillis

Thinking machines CM-2: The Classic Symmetric SIMD supercomputer (mid-80's):

Description: Up to 64K bit-serial processing elements.

Strength: Supports deterministic programming models … single thread of control for ease of understanding.

Weakness: Poor floating point performance. Programming model was not general enough. TMC struggled throughout the 90's and filed for bankruptcy in 1994.

Third party names are the property of their owners.

Examples: Symmetric Multi-Processor

Cray 2: The Classic Symmetric Multi-Processor (mid-80’s):

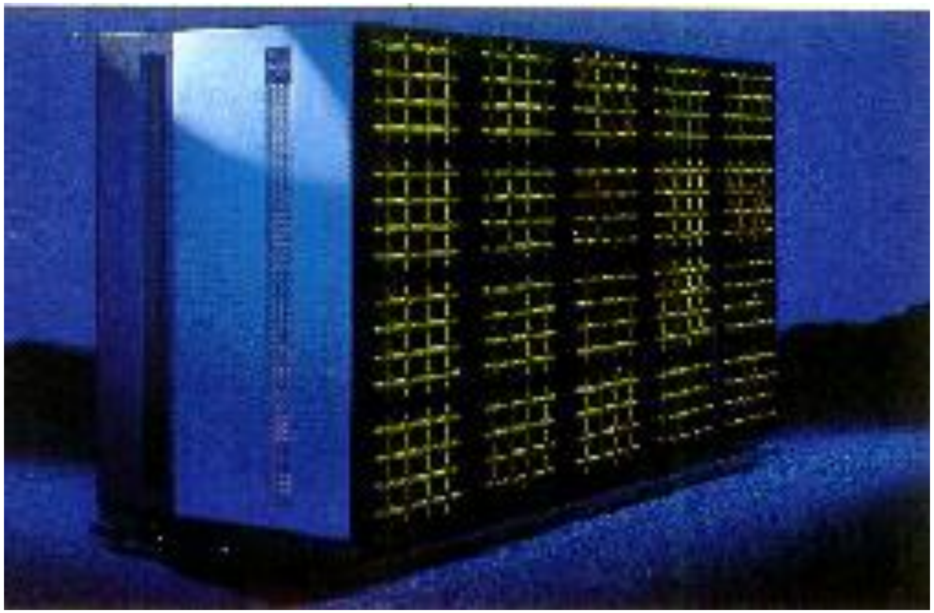
Description: multiple Vector processors connected to a custom high speed memory. 500 MFLOP processors.

Strength: Simple memory architecture makes this the easiest supercomputer in history to program. Truly an SMP (no caches).

Weakness: Poor scalability. VERY expensive due to the fact that everything (memory to processors) were custom.

Third party names are the property of their owners.

Examples: Massively Parallel Processors



Paragon MP: The Classic MPP (early-90’s):

Description: 3 i860 CPU’s (a vector inspired microprocessor) connected by a custom mesh interconnect. 40 MFLOP processors*.

Strength: A massively scalable machine (3000+ processors). The lights were pretty, but useful helping to show bottlenecks in the code.

Weakness: Hard to program (NX message passing and later MPI). Expensive due to low volume microprocessor, custom back-plane and packaging.

Third party names are the property of their owners.

Examples: Cluster

NCSA's Itanium cluster: (early-00's):

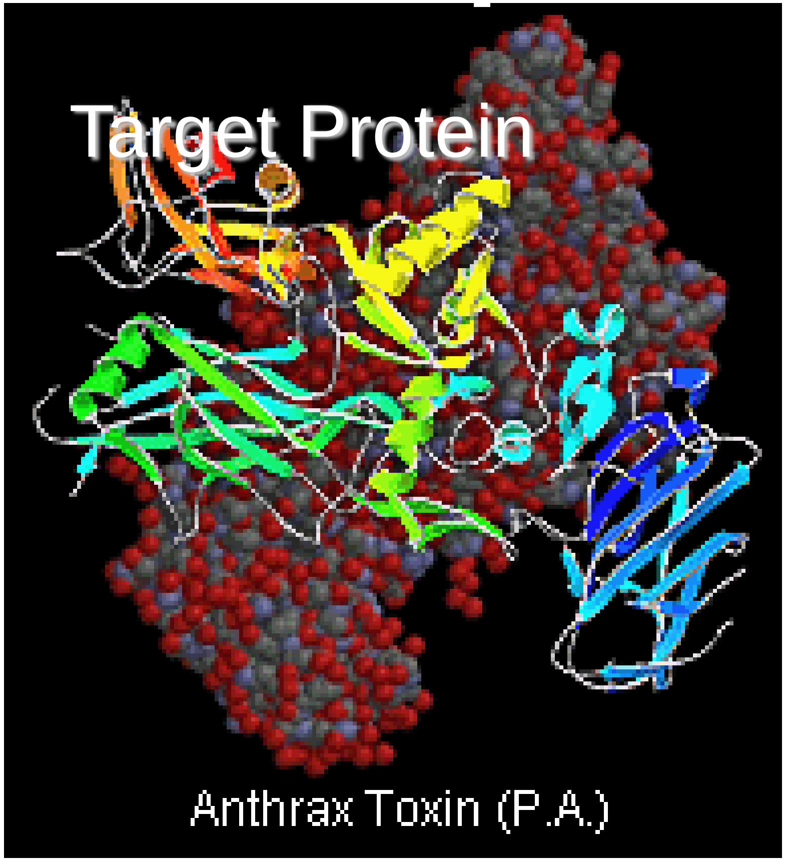
Description: 160 dual IPF nodes connected by a Myracom network. 3.2 GFLOP per processors.

Strength: Highly scalable, nothing custom so hardware costs are reasonable.

Weakness: Hard to program (MPI). Lack of application software. Cluster middleware is fragile and still evolving.

Third party names are the property of their owners.

Examples: distributed computing (e.g. GRID)



Intel’s Cure@home program.

Description: Thousands of home PC’s donated to solve important problems.

Strength: Highly scalable – the ultimiate costs performance since it uses compute-cycles that would otherwise be wasted.

Weakness: Only coarse grained embarrassingly parallel algorithms can be used. Security constraints difficult to enforce.

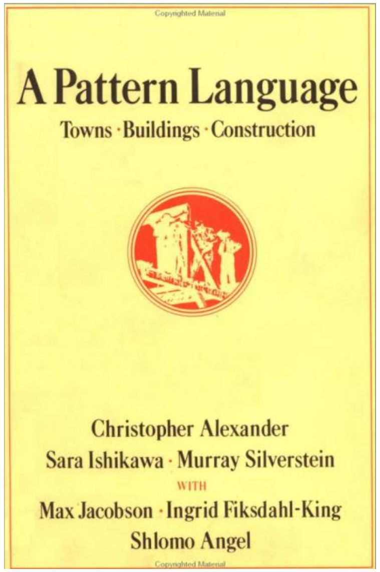
Agenda – parallel theory

- How to sound like a parallel programmer
- An overly brief look at parallel architecture
- ➡ Understanding design patterns for parallel programming

Getting started with parallel algorithms

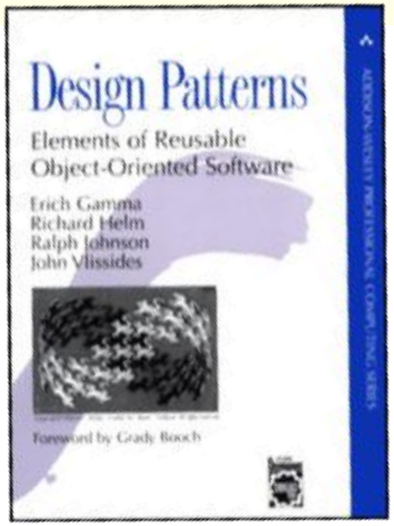
- Concurrency is a general concept
  - … multiple activities that can occur and make progress at the same time.
- A parallel algorithm is any algorithm that uses concurrency to solve a problem of a given size in less time
- Scientific programmers have been working with parallelism since the early 80’s
  - Hence we have almost 30 years of experience to draw on to help us understand parallel algorithms.

A formal discipline of design


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

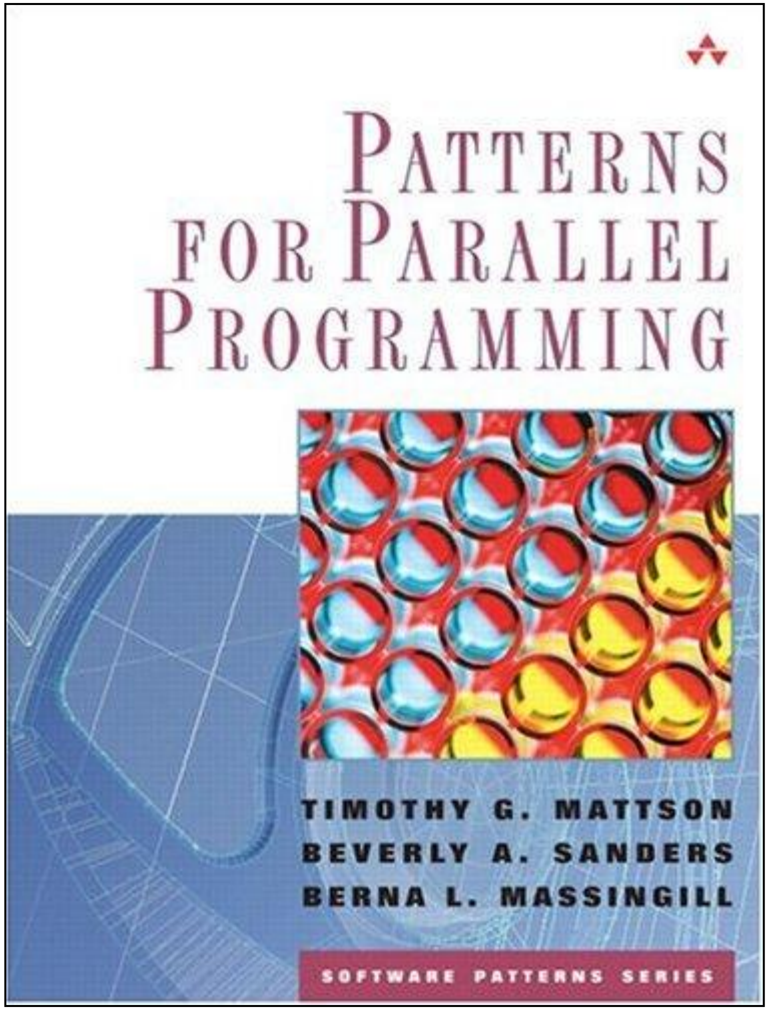
- Christopher Alexander’s approach to (civil) architecture:
  - A design pattern “describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“ *Page x*, *A Pattern Language,* Christopher Alexander
- A pattern language is an organized way of tackling an architectural problem using patterns

- The gang of 4 used patterns to bring order to the chaos of object oriented design.
- The book “over night” turned object oriented design from “an art” to a systematic design discipline.

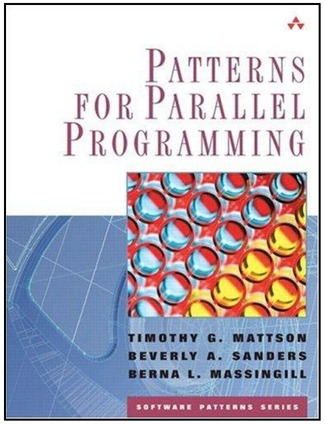

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Can Design patterns bring order to parallel programming?

- The book “Patterns for Parallel Programming” contains a design pattern language to capture how experts think about parallel programming.
- It is an attempt to be to parallel programming what the GOF book was to object oriented programming.
- The patterns were mined from established practice in scientific computing … hence its a useful set of patterns but not complete (e.g. its weak on graph algorithms).

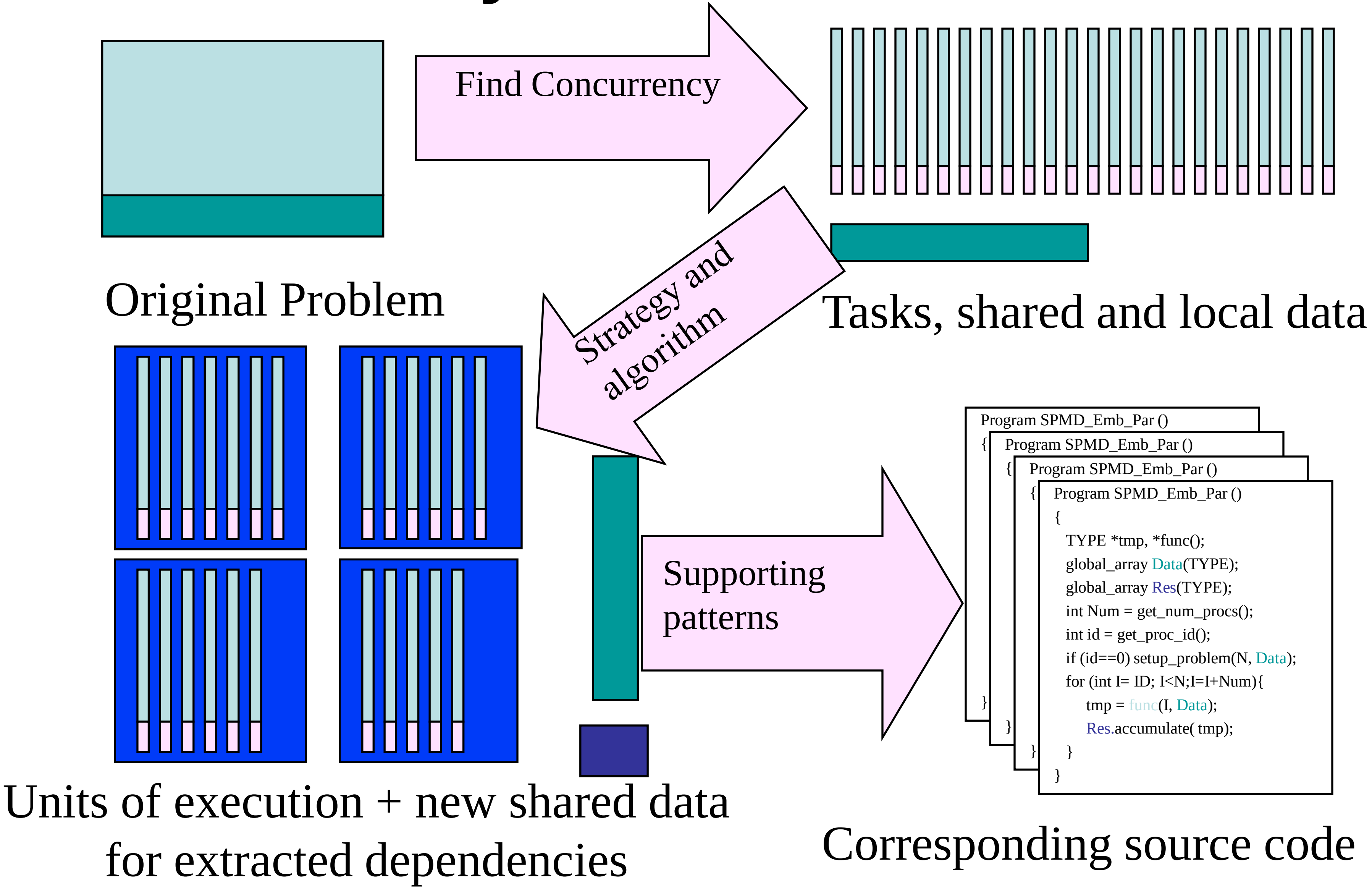


Basic approach from the book



- Identify the concurrency in your problem:
  - Find the tasks, data dependencies and any other constraints.
- Develop a strategy to exploit this concurrency:
  - Which elements of the parallel design will be used to organize your approach to exploiting concurrency.
- Identify and use the right algorithm pattern to turn your strategy into the design of a specific algorithm.
- Choose the supporting patterns to move your design into source code.
  - This step is heavily influenced by the target platform

Concurrency in Parallel software:


Find Concurrency
Original Problem
Tasks, shared and local data
Strategy and algorithm
Units of execution + new shared data for extracted dependencies
Supporting patterns
Corresponding source code

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

Strategies for exploiting concurrency

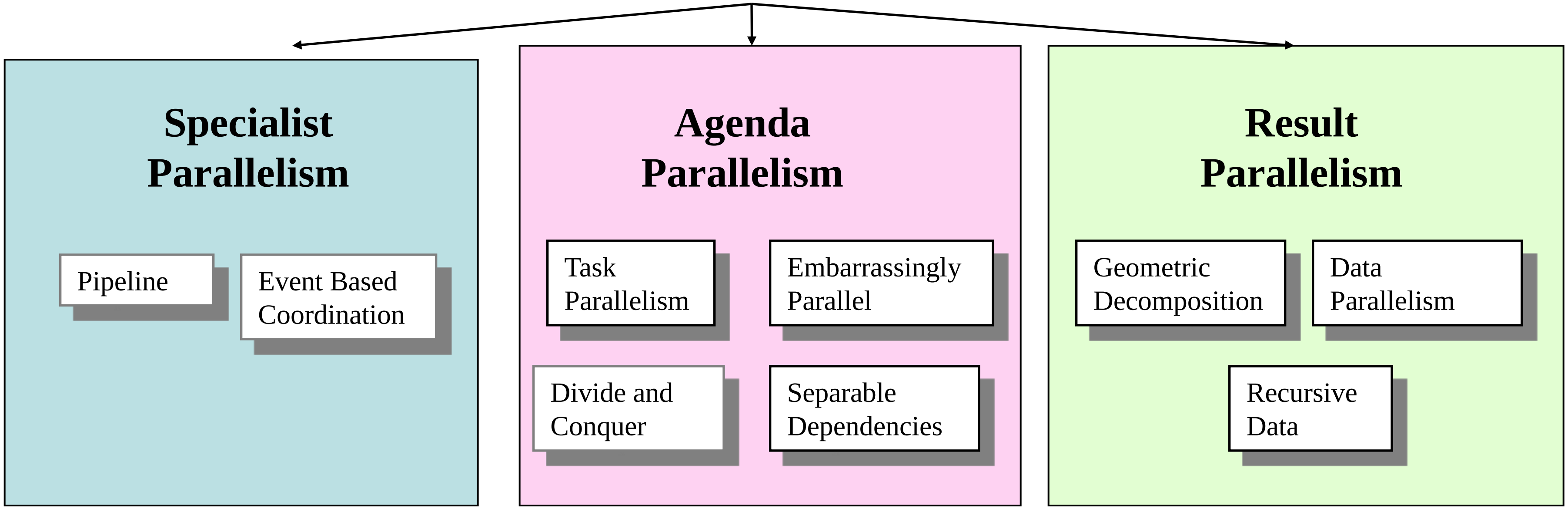
- Given the results from your “finding concurrency” analysis, there are many different ways to turn them into a parallel algorithm.
- In most cases, one of three Distinct Strategies are used
  - Agenda parallelism: The collection of tasks that are to be computed.
  - Result parallelism: Updates to the data.
  - Specialist parallelism: The flow of data between a fixed set of tasks.

Ref: N. Carriero and D. Gelernter, **How to Write Parallel Programs: A First Course**, 1990.

The Algorithm Design Patterns

Start with a basic concurrency decomposition

- A problem decomposed into a set of tasks
- A data decomposition aligned with the set of tasks … designed to minimize interactions between tasks and make concurrent updates to data safe.
- Dependencies and ordering constraints between groups of tasks.



Specialist Parallelism

Pipeline

Event Based Coordination

Agenda Parallelism

Task Parallelism

Embarrassingly Parallel

Divide and Conquer

Separable Dependencies

Result Parallelism

Geometric Decomposition

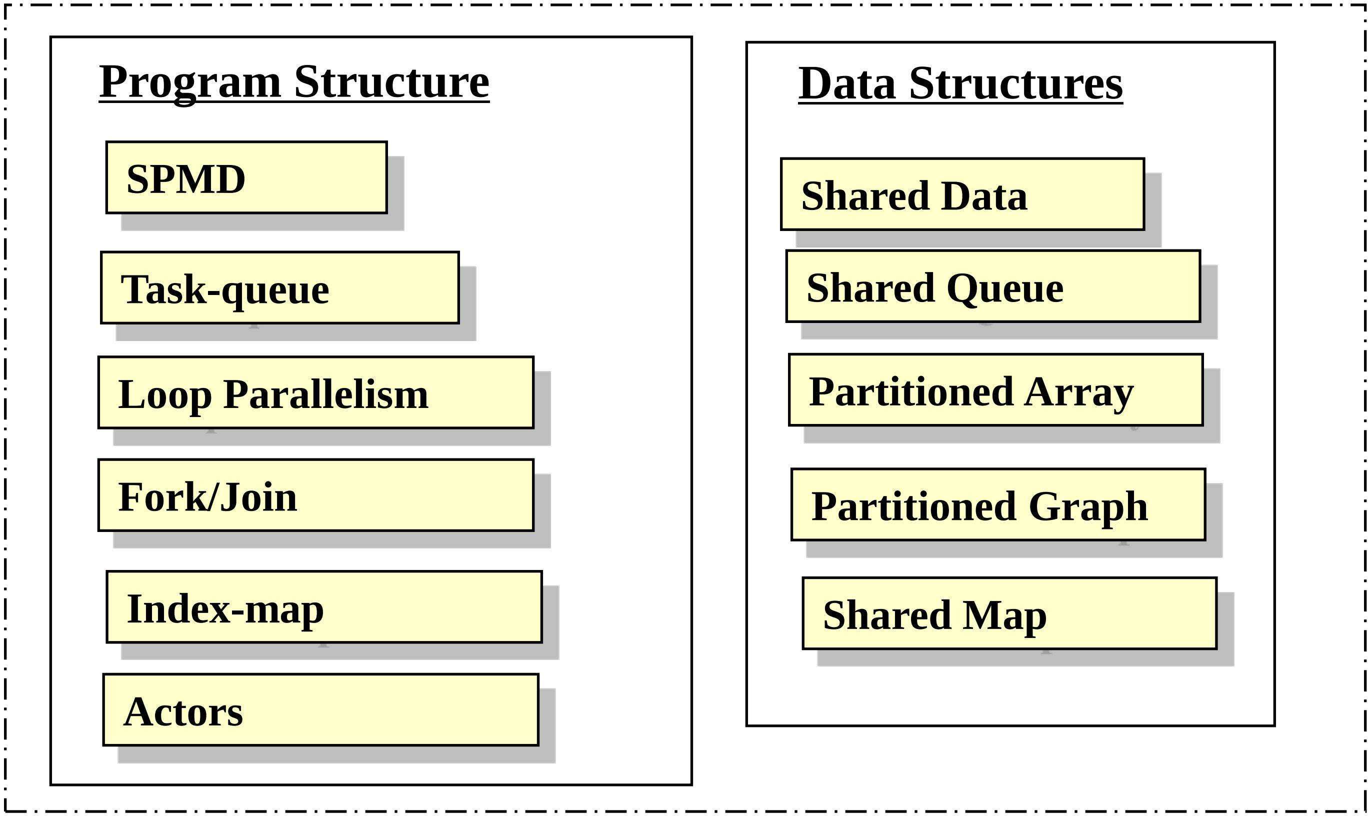
Data Parallelism

Recursive Data

Implementation strategy Patterns

(Supporting Structures)

Patterns that support Implementing Algorithm strategies as code.



Program Structure

- SPMD
- Task-queue
- Loop Parallelism
- Fork/Join
- Index-map
- Actors

Data Structures

- Shared Data
- Shared Queue
- Partitioned Array
- Partitioned Graph
- Shared Map

Our approach for today …

- Once you understand the basic patterns, you can implement them in any language … the parallel programming language we use just doesn’t matter that much
- We will use OpenMP to explore these patterns and help you become “expert” parallel programmers.
- Why OpenMP?
  - Its easy to learn … you quickly move form learning constructs to writing code.
  - Its everywhere … OK, its everywhere as long as you focus on shared memory machines.

Outline

/storage/software/tim/omp.tar

- Intro to parallel programming
- ➡ An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment

- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

OpenMP* Overview:

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$O
C$O
C$
ED
C
#p
```

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

```
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

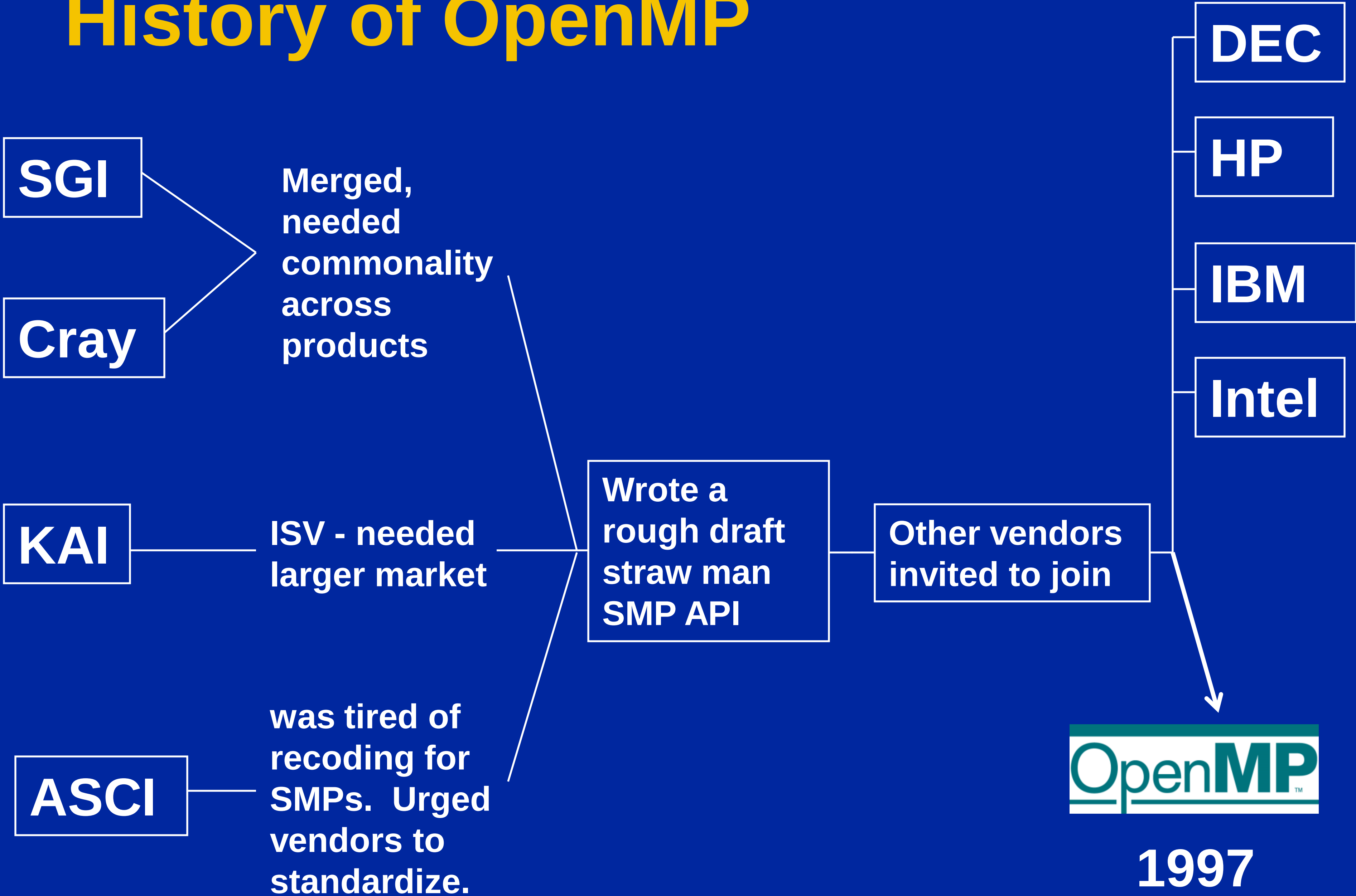
* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP pre-history

- OpenMP based upon SMP directive standardization efforts PCF and aborted ANSI X3H5 – late 80's
  - Nobody fully implemented either standard
  - Only a couple of partial implementations
- Vendors considered proprietary API's to be a competitive feature:
  - Every vendor had proprietary directives sets
  - Even KAP, a "portable" multi-platform parallelization tool used different directives on each platform

PCF – Parallel computing forum KAP – parallelization tool from KAI.

History of OpenMP



SGI

Cray

Merged, needed commonality across products

KAI

ISV - needed larger market

ASCI

was tired of recoding for SMPs. Urged vendors to standardize.

Wrote a rough draft straw man SMP API

Other vendors invited to join

DEC

HP

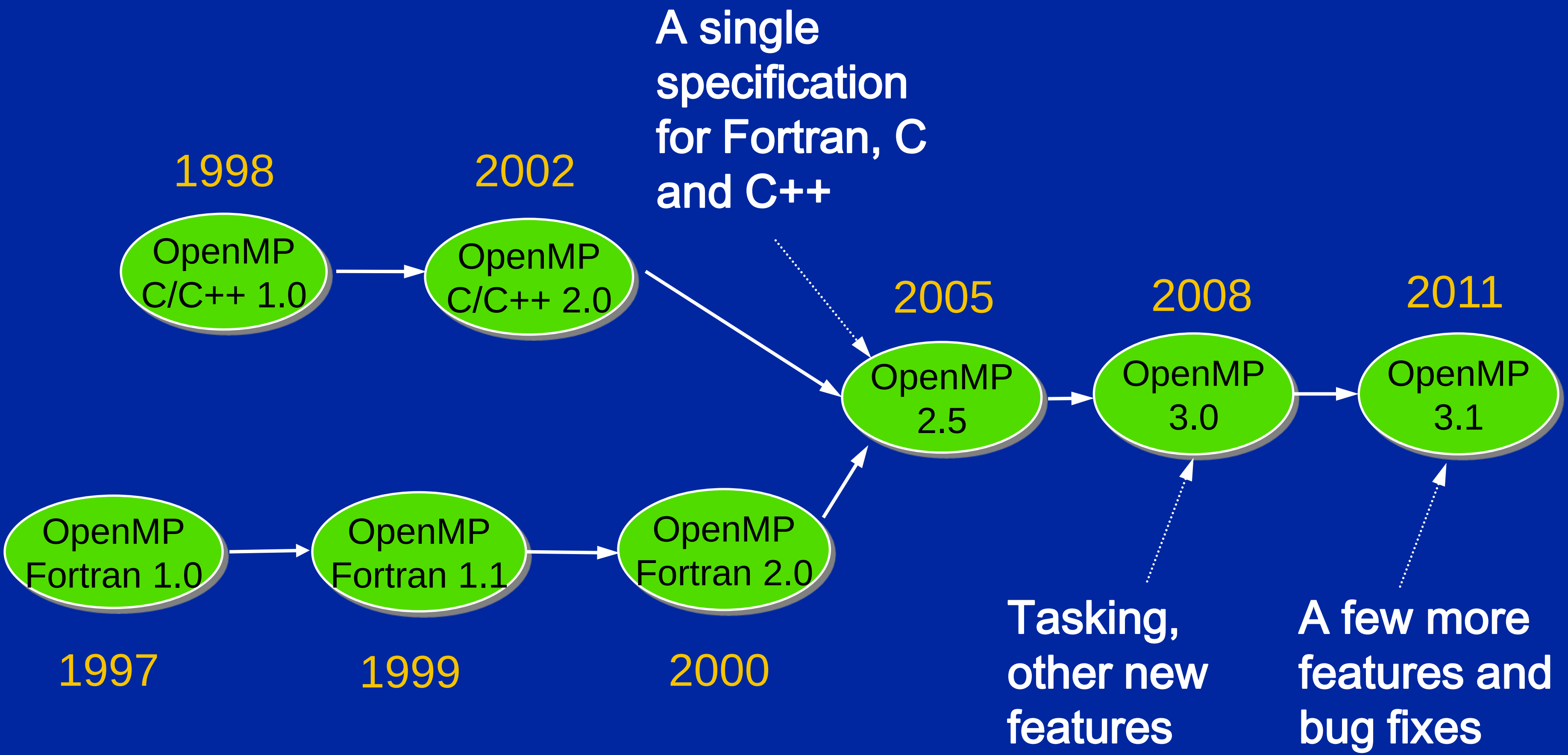
IBM

Intel

OpenMP

1997

OpenMP Release History



A single specification for Fortran, C and C++

1998

OpenMP C/C++ 1.0

2002

OpenMP C/C++ 2.0

1997

OpenMP Fortran 1.0

1999

OpenMP Fortran 1.1

2000

OpenMP Fortran 2.0

2005

OpenMP 2.5

2008

OpenMP 3.0

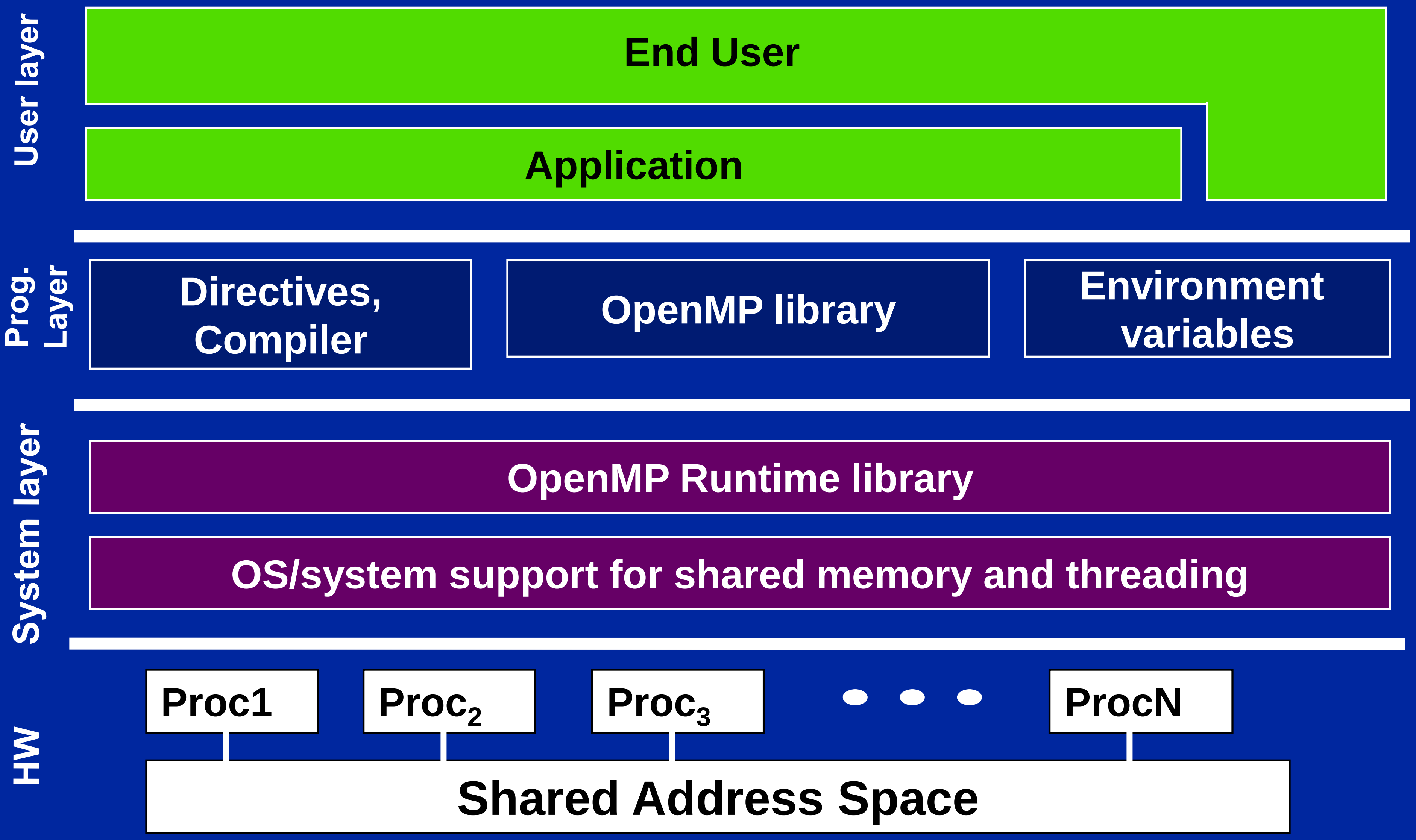
2011

OpenMP 3.1

Tasking, other new features

A few more features and bug fixes

OpenMP Basic Defs: Solution Stack



User layer

End User

Application

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1

Proc$_2$

Proc$_3$

ProcN

Shared Address Space

OpenMP core syntax

- **Most of the constructs in OpenMP are compiler directives.**

  #pragma omp *construct [clause [clause]…]*

  - **Example**

    #pragma omp parallel num_threads(4)

- **Function prototypes and types in the file:**

  #include <omp.h>

- **Most OpenMP* constructs apply to a "structured block".**
  - **Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.**
  - **It's OK to have an exit() within the structured block.**

Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{



     int ID = 0;

     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- **Write a multithreaded program that prints “hello world”.**

```
        #include “omp.h”
void main()
{
         #pragma omp parallel
         {

    int ID = 0;

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

Switches for compiling and linking

gcc -fopenmp	**gcc**
pgcc -mp	**pgi**
icl /Qopenmp	**intel(windows)**
icc –openmp	**intel (linux)**

Exercise 1: Solution

A multi-threaded “Hello world” program

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

```
#include “omp.h”
void 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.

Outline

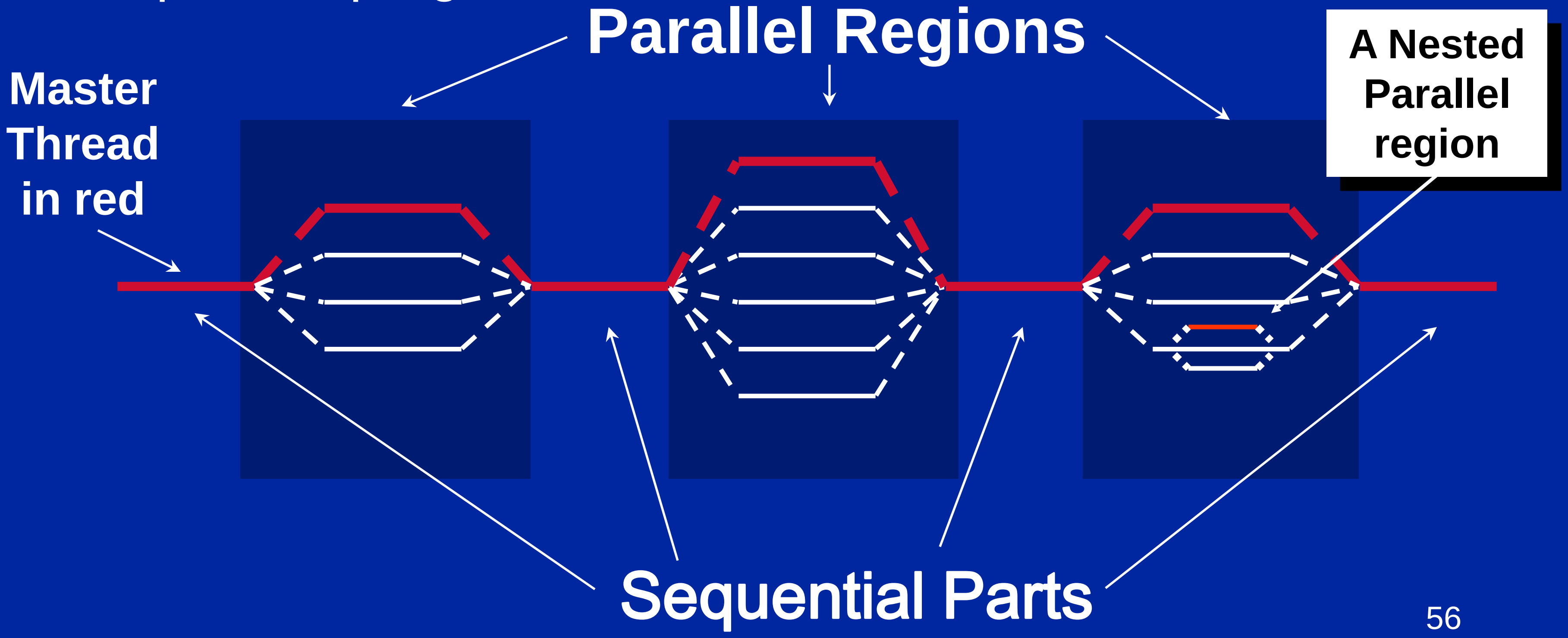
- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment

- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

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

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();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

Thread Creation: Parallel Regions

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

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

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

Runtime function returning a thread ID

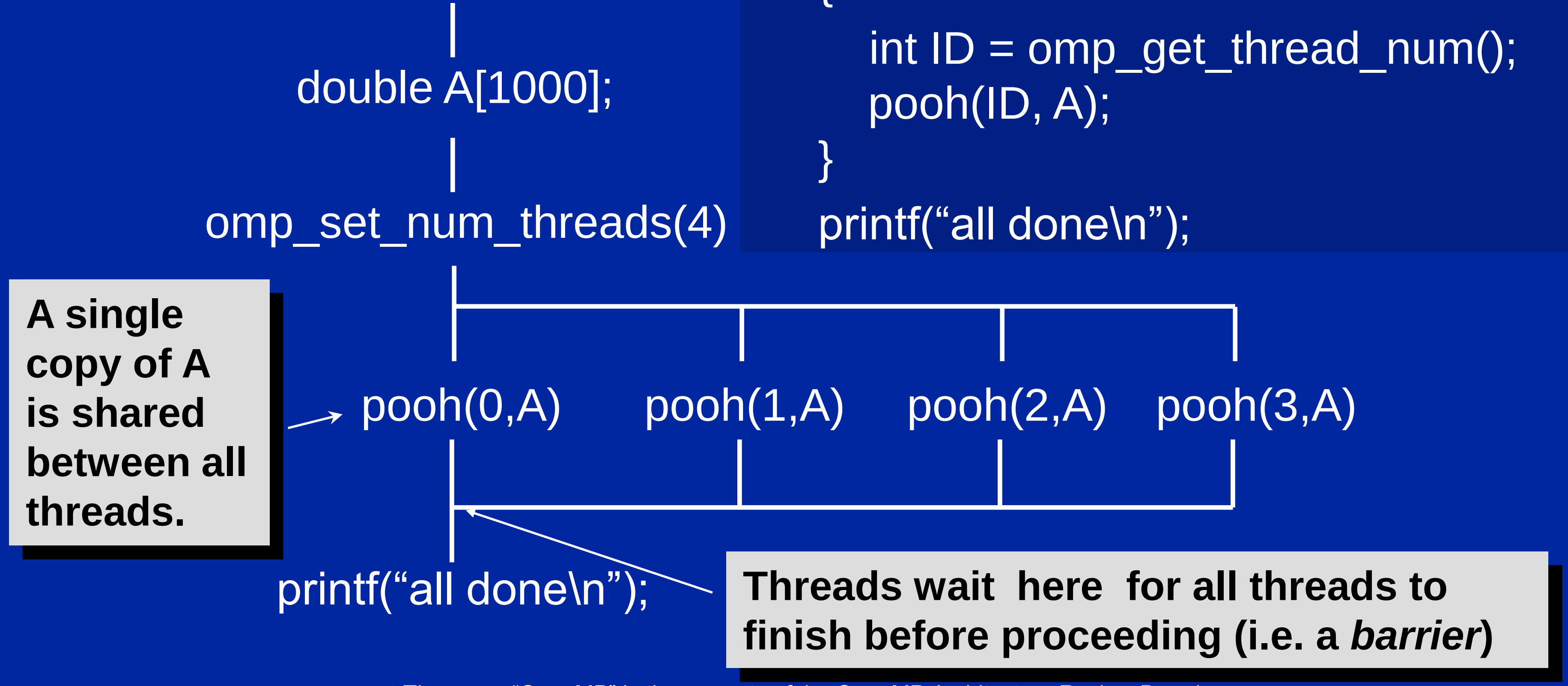
- **Each thread calls pooh(ID,A) for ID = 0 to 3**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

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”);
```



double A[1000];

omp_set_num_threads(4)

pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)

printf(“all done\n”);

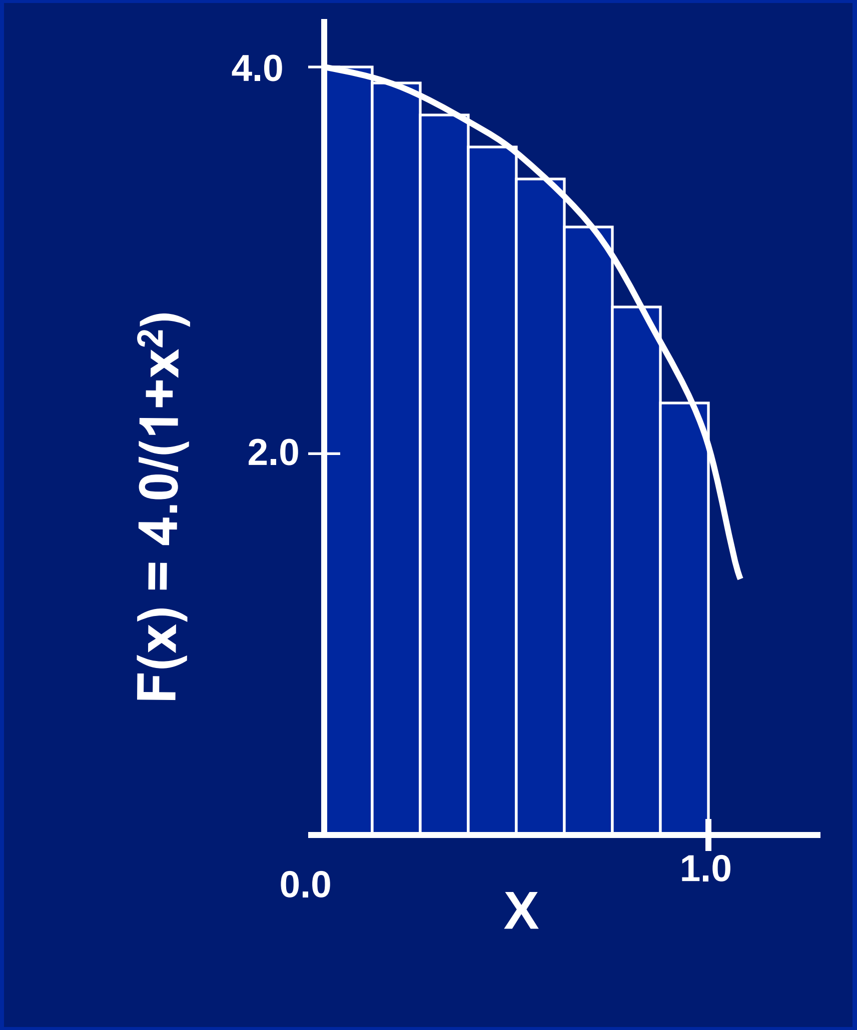
A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

Exercises 2 to 4:

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Exercises 2 to 4: 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;
}
```

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment
- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Discussed later

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```
float res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+nthrds){
            B =  big_job(i);
#pragma omp critical
              consume (B, res);
     }
}
```

Threads wait their turn – only one at a time calls consume()

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel

{
         double tmp, B;

        B =  DOIT();

        tmp = big_ugly(B);

 #pragma omp atomic
           X +=  tmp;

}
```

Atomic only protects the read/update of X

Exercise 3

- In exercise 2, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence “sloshing independent data” back and forth between threads.
- Modify your “pi program” from exercise 2 to avoid false sharing due to the sum array.

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- ➡ Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment
- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct …. Available in OpenMP 3.0

Discussed later

The loop worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel

{
#pragma omp for
          for (I=0;I<N;I++){
                    NEAT_STUFF(I);
          }
}
```

Loop construct name:

- C/C++: for
- Fortran: do

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing Constructs

A motivating example

Sequential code

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

OpenMP parallel 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;
            if (id == Nthrds-1)iend = N;
            for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

loop worksharing constructs: The schedule clause

- **The schedule clause affects 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 (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

loop work-sharing constructs:
The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can “learn” from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

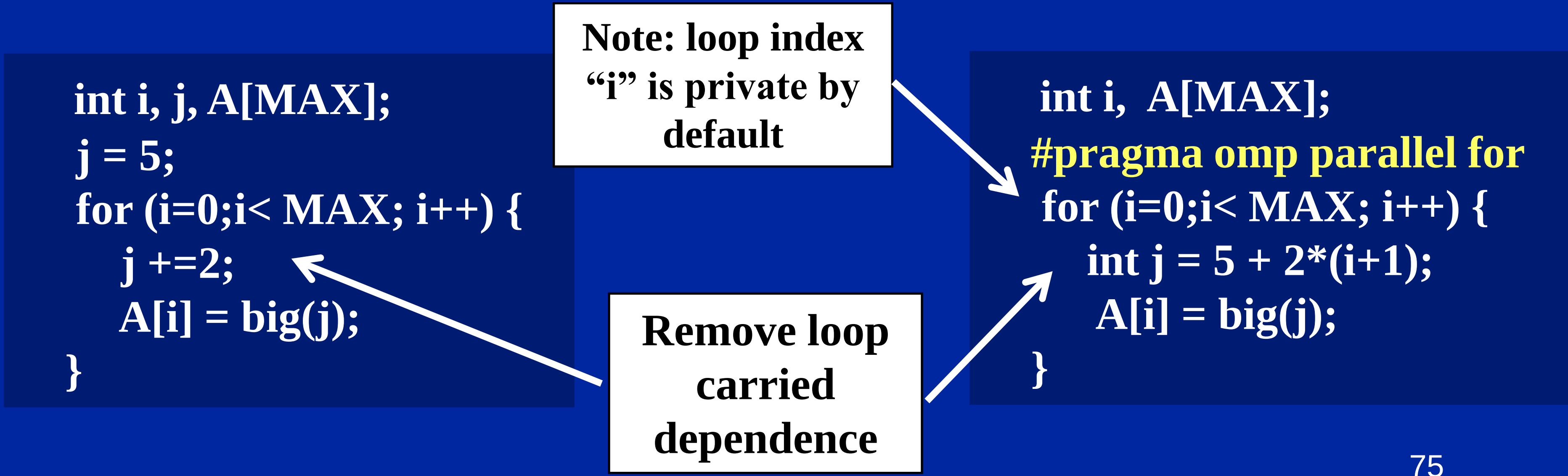
```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

These are equivalent

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test



```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
   j +=2;
   A[i] = big(j);
}
```

Note: loop index “i” is private by default

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
   int j = 5 + 2*(i+1);
   A[i] = big(j);
}
```

Remove loop carried dependence

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
          .....
  }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop may not have good load balance

Reduction

- How do we handle this case?

```
   double  ave=0.0, A[MAX];    int i;
   for (i=0;i< MAX; i++) {
       ave + = A[i];
  }
 ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed
- This is a very common situation … it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in "list" must be shared in the enclosing parallel region.

```
  double  ave=0.0, A[MAX];    int i;
  #pragma omp parallel for reduction (+:ave)
  for (i=0;i< MAX; i++) {
      ave + = A[i];
 }
ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

Exercise 4: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Exercise 5: Optimizing loops

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment
- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
          do_many_things();
#pragma omp master
          {     exchange_boundaries();   }
#pragma omp  barrier
          do_many_other_things();
}
```

Sections worksharing Construct

- **The *Sections* worksharing construct gives a different structured block to each thread.**

```
#pragma omp parallel
{

   #pragma omp sections
   {
   #pragma omp section
            X_calculation();
   #pragma omp section
             y_calculation();
   #pragma omp section
             z_calculation();
   }

}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

Single worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
          do_many_things();
#pragma omp single
          {     exchange_boundaries();   }
          do_many_other_things();
}
```

Synchronization: ordered

- The ordered region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

Synchronization: Lock routines

A lock implies a memory fence (a “flush”) of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don’t need to use a flush on the lock variable.

Synchronization: Simple Locks

- Protect resources with locks.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
      printf(“%d %d”, id, tmp);
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

Runtime Library routines

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - **Are we in an active parallel region?**
    - omp_in_parallel()
  - **Do you want the system to dynamically vary the number of threads from one parallel construct to another?**
    - omp_set_dynamic, omp_get_dynamic();
  - **How many processors in the system?**
    - omp_num_procs()

…plus a few less commonly used routines.

Runtime Library routines

- **To use a known, fixed number of threads in a program, (1) tell the system that you don’t want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.**

```
#include <omp.h>
void main()
{   int num_threads;
     omp_set_dynamic( 0 );
     omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {     int id=omp_get_thread_num();
#pragma omp single
            num_threads = omp_get_num_threads();
          do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

Environment Variables

- **Set the default number of threads to use.**
  - OMP_NUM_THREADS *int_literal*
- **Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.**
  - OMP_SCHEDULE "schedule[, chunk_size]"

… Plus several less commonly used environment variables.

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- ➡ Data Environment
- OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

Data environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

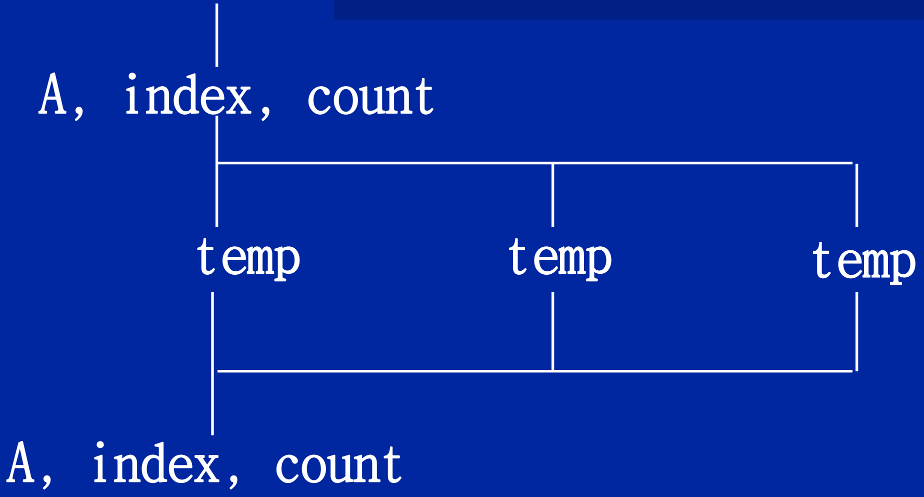
Data sharing: Examples

```
  double A[10];
  int main() {
  int index[10];
 #pragma omp parallel
        work(index);
  printf(“%d\n”, index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



A, index, count

temp temp temp

A, index, count

Data sharing:
Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
    - SHARED
    - PRIVATE
    - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
    - LASTPRIVATE
- The default attributes can be overridden with:
    - DEFAULT (PRIVATE | SHARED | NONE)

      DEFAULT(PRIVATE) *is Fortran only*

All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

Data Sharing: Private Clause

- **private(var) creates a new local copy of var for each thread.**
  - **The value of the private copies is uninitialized**
  - **The value of the original variable is unchanged after the region**

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
                  tmp += j;
      printf(“%d\n”, tmp);
}
```

tmp was not initialized

tmp is 0 here

Data Sharing: Private Clause When is the original variable valid?

- **The original variable's value is unspecified if it is referenced outside of the construct**
  - **Implementations may reference the original variable or a copy ….. a dangerous programming practice!**

```
int tmp;
void danger() {
      tmp = 0;
#pragma omp parallel private(tmp)
      work();
     printf(“%d\n”, tmp);
}
```

tmp has unspecified value

```
extern int tmp;
void work() {
      tmp = 5;
}
```

unspecified which copy of tmp

Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
      if ((i%2)==0) incr++;
      A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
  double x; int i;
#pragma omp parllel for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  *lastterm = x;
}
```

"x" has the value it held for the "last sequential" iteration (i.e., for i=(n-1))

Data Sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
                  variables A,B, and C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Outside this parallel region ...

- The values of "B" and "C" are unspecified if referenced in the region but outside the construct.

Data Sharing: Default Clause

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - Exception: #pragma omp task
- To change default: **DEFAULT(PRIVATE)**
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **DEFAULT(NONE)***:* *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Data Sharing: Default Clause Example

```
      itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      .........
C$OMP END PARALLEL
```

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      .........
C$OMP END PARALLEL
```

These two code fragments are equivalent

Exercise 6: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn’t do it right.
- Find and fix the errors (hint … the problem is with the data environment).

Exercise 6 (cont.)

- Once you have a working version, try to optimize the program?
  - Try different schedules on the parallel loop.
  - Try different mechanisms to support mutual exclusion … do the efficiencies change?

Exercise 7: Molecular dynamics

- The program supplied in the folder “MolDyn” is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine `forces` (in forces.c)
- Parallelise this routine using a parallel for construct and atomics. Think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables.
- Optimize the program (hint: Experiment with different schedules kinds).

Exercise 7 (cont.)

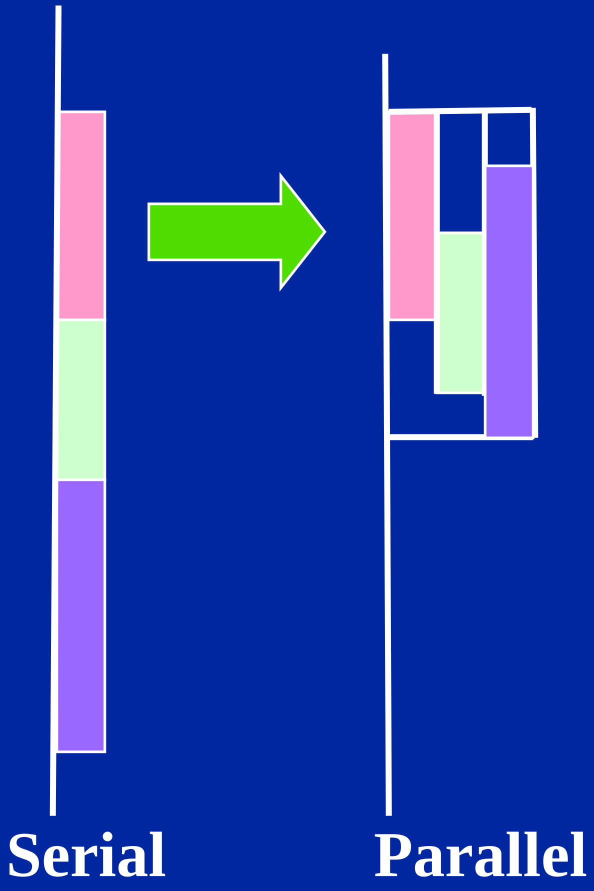
- Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  - code other than the forces loop must be executed by a single thread (or workshared).
  - how does the data sharing change?
- The atomics are a bottleneck on most systems.
  - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number.
  - Which thread(s) should do the final accumulation into f?

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment
- ➡ OpenMP tasks
- The OpenMP Memory model
- A survey of parallel programming models

What are tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
    - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above
    - Tasks are composed of:
        - code to execute
        - data environment
        - internal control variables (ICV)



Task Construct – Explicit Task View

- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – lets call this thread “L”
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region’s single construct

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task private(p)
      process(p);
    p = p->next;  //block 3
    }
  }
}
```

Simple Task Example

```
#pragma omp parallel num_threads(8)
// assume 8 threads
{
 #pragma omp single private(p)
 {
   …
   while (p) {
   #pragma omp task
     {
       processwork(p);
     }
     p = p->next;
   }
 }
}
```

A pool of 8 threads is created here

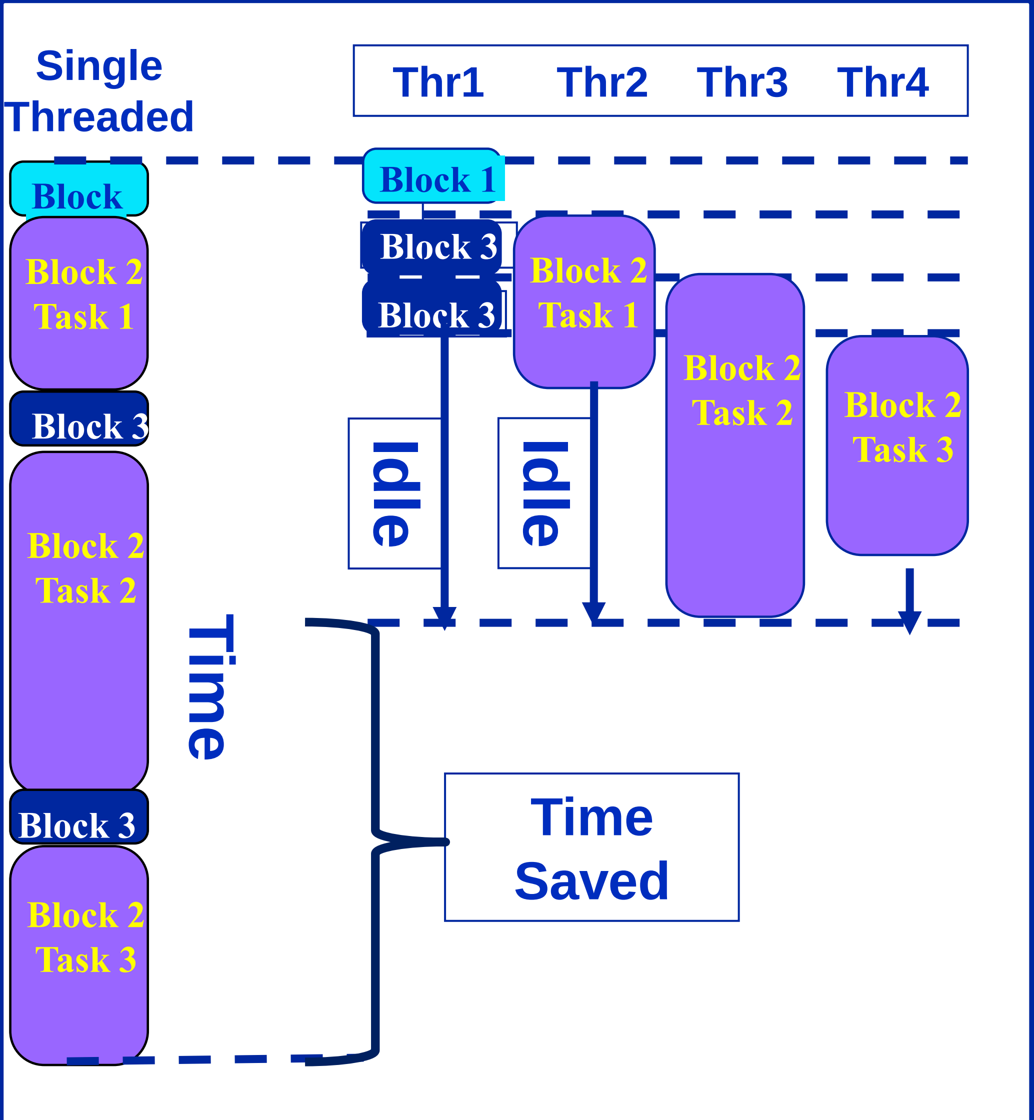
One thread gets to execute the while loop

The single “while loop” thread creates a task for each instance of processwork()

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task
      process(p);
    p = p->next;  //block 3
    }
  }
}
```



When are tasks guaranteed to complete

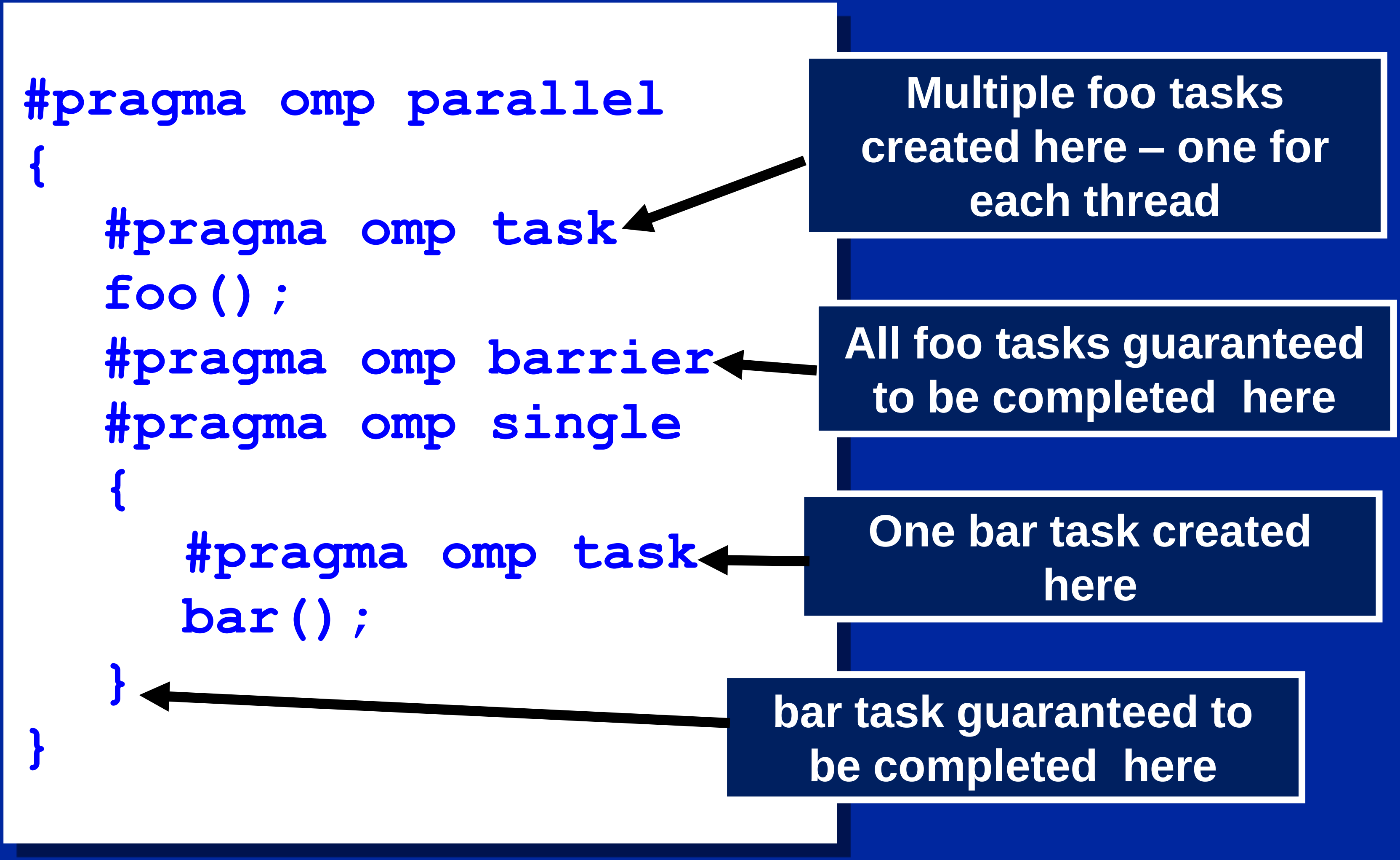
- Tasks are gauranteed to be complete at thread barriers:

```
#pragma omp barrier
```

- … or task barriers

```
#pragma omp taskwait
```

Task Completion Example



```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Data Scoping with tasks: Fibonacci example.

```
int fib ( int n )
{

int x,y;
   if ( n < 2 ) return n;
#pragma omp task
   x = fib(n-1);
#pragma omp task
   y = fib(n-2);
#pragma omp taskwait
   return x+y
}
```

n is private in both tasks

x is a private variable
y is a private variable

What’s wrong here?

Can’t use private variables outside of tasks

Data Scoping with tasks: Fibonacci example.

```
int fib ( int n )
{

int x,y;
   if ( n < 2 ) return n;
#pragma omp task shared (x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib(n-2);
#pragma omp taskwait
   return x+y
}
```

n is private in both tasks

x & y are shared
Good solution
we need both values to
compute the sum

What’s wrong here?

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single private(e)
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

Exercise 8: tasks in OpenMP

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.

Exercise 9: linked lists the hard way

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using constructs defined in OpenMP 2.5 (loop worksharing constructs … i.e. don’t use OpenMP 3.0 tasks).
- Once you have a correct program, optimize it.

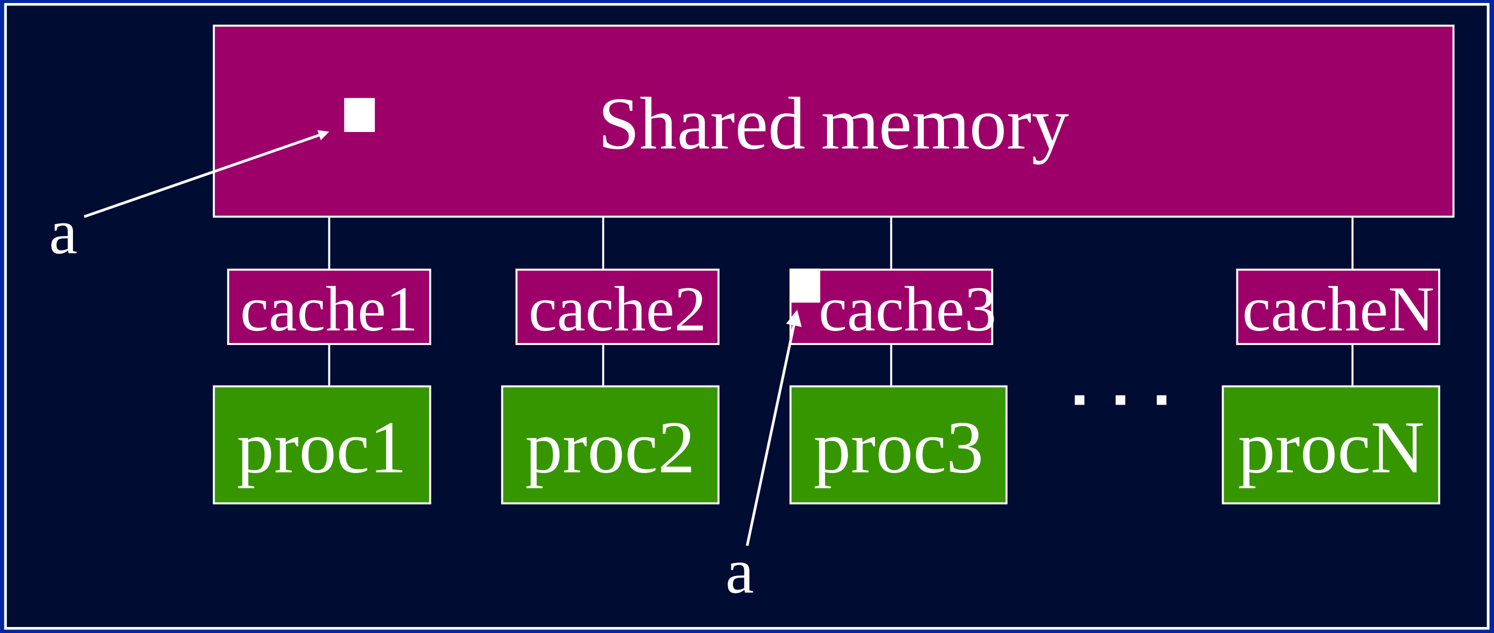
Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment

- OpenMP tasks
- ➡ The OpenMP Memory model
- A survey of parallel programming models

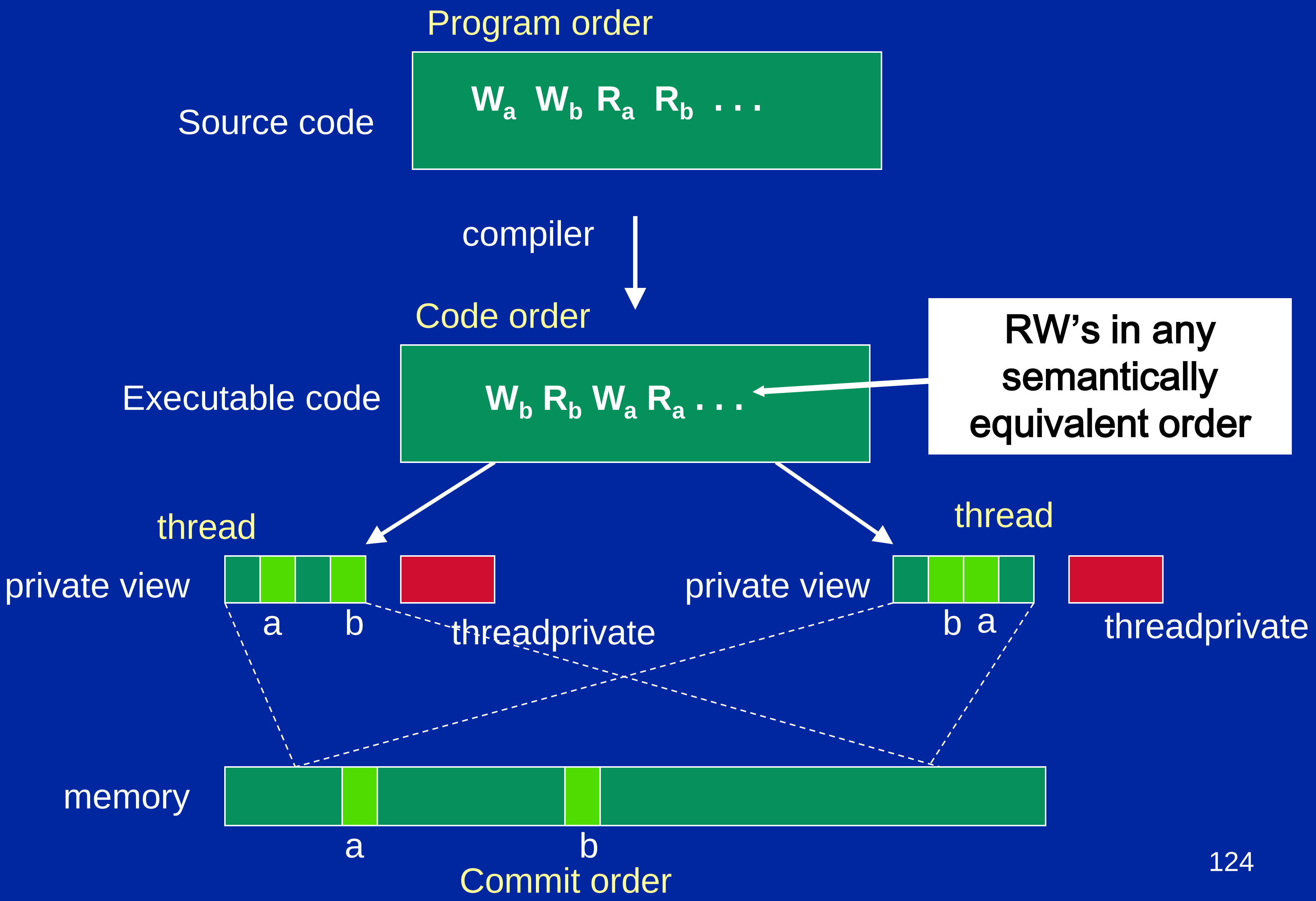
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- A memory model is defined in terms of:
  - ◆ **Coherence**: Behavior of the memory system when a single address is accessed by multiple threads.
  - ◆ **Consistency**: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP Memory Model: Basic Terms



Program order

Source code

W_a W_b R_a R_b . . .

compiler

Code order

Executable code

W_b R_b W_a R_a . . .

RW's in any semantically equivalent order

thread

private view

a b

threadprivate

thread

private view

b a

threadprivate

memory

a b

Commit order

Consistency: Memory Access Re-ordering

- **Re-ordering:**
  - **Compiler re-orders program order to the code order**
  - **Machine re-orders code order to the memory commit order**
- **At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.**
- **Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)**
  - **… i.e. how do the values "seen" by a thread change as you change how ops follow (→) other ops.**
  - **Possibilities include:**
    - **R→R, W→W, R→W, R→S, S→S, W→S**

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of weak consistency:
  - S ops must be in sequential order across threads.
  - Can not reorder S ops with R or W ops on the same thread
    - Weak consistency guarantees

      S→W, S→R , R→S, W→S, S→S
- The Synchronization operation relevant to this discussion is flush.

Flush

- **Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set".**
- **The flush set is:**
  - **"all thread visible variables" for a flush construct without an argument list.**
  - **a list of variables when the "flush(list)" construct is used.**
- **The action of Flush is to guarantee that:**
  - **All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes**
  - **All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush.**
  - **Flushes with overlapping flush sets can not be reordered.**

Memory ops: R = Read, W = write, S = synchronization

Synchronization: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

flush(A);   // flush to memory to make sure other
            //  threads can pick up the right value
```

Note: OpenMP’s flush is analogous to a fence in other shared memory API’s.

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- Compiler generally cannot move instructions:
  - past a barrier
  - past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.

Pair wise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise 10: producer consumer

- Parallelize the “prod_cons.c” program.
- This is a well known pattern called the producer consumer pattern
  - One thread produces values that another thread consumes.
  - Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

Exercise 10: prod_cons.c

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *)malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
}
```

Outline

- Intro to parallel programming
- An Introduction to OpenMP
- Creating threads
- Basic Synchronization
- Parallel loops (intro to worksharing)
- The rest of worksharing and synchronization
- Data Environment

- OpenMP tasks
- The OpenMP Memory model
- ⇨ A survey of parallel programming models

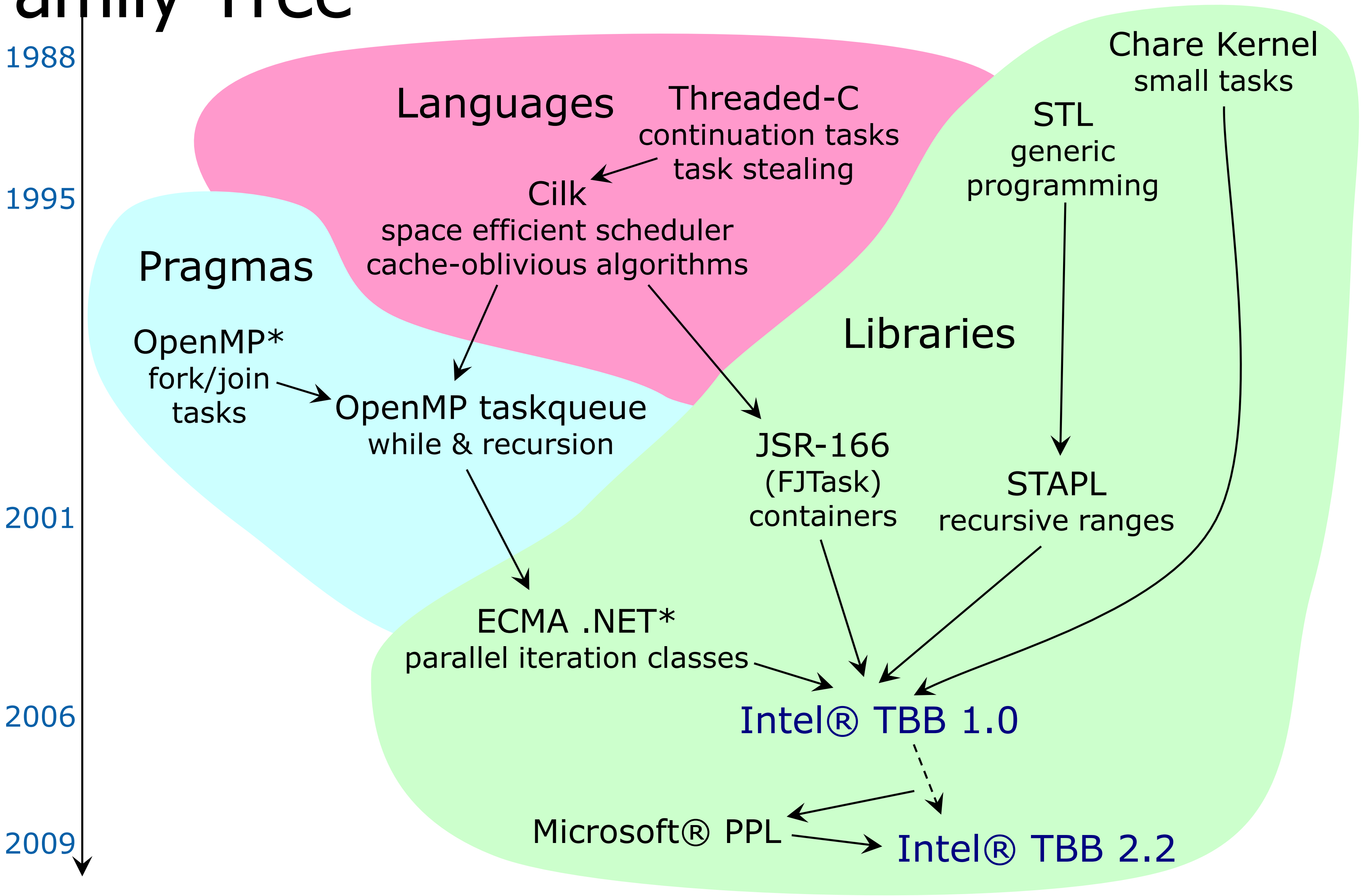
a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

Intel® Threading Building Blocks (TBB)

- It is a *template* ***library*** for generic programming with C++
- It provides a *high-level abstraction* for parallel programming
  - You specify tasks patterns instead of threads
  - Hides low level details of thread management (balances load of logical tasks across a set of physical threads)
  - Full support for nested parallelism
- It facilitates scalable performance
  - Strives for efficient use of cache, and balances load
  - Portable across Linux*, Mac OS*, Windows*, and Solaris*
- Can be mixed with native threads and OpenMP
- Open source and licensed versions available

Family Tree



*Other names and brands may be claimed as the property of others

Limitations

- TBB is not intended for
  - I/O bound processing
  - Real-time processing
  - Concurrent algorithms

- General limitations
  - Direct use only from C++
  - Distributed memory not supported (target is desktop)
  - Requires more work than “sprinkling” pragmas

Intel® TBB 2.2 Components

Generic Parallel Algorithms

parallel_for, parallel_for_each
parallel_reduce
parallel_scan
parallel_do
pipeline
parallel_sort
parallel_invoke

Task scheduler

task_group
task
task_scheduler_init
task_scheduler_observer

Synchronization Primitives

atomic, mutex, recursive_mutex
spin_mutex, spin_rw_mutex
queuing_mutex, queuing_rw_mutex
null_mutex, null_rw_mutex

Threads

tbb_thread

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_bounded_queue
concurrent_vector

Thread Local Storage

combinable
enumerable_thread_specific

Memory Allocation

tbb_allocator
zero_allocator
cache_aligned_allocator
scalable_allocator

Task-based Programming

- Tasks are light-weight entities at user-level
  - TBB parallel algorithms map tasks onto threads automatically
  - Task scheduler manages the thread pool
    - Scheduler is *unfair* to favor tasks that have been most recent in the cache
  - Oversubscription and undersubscription of core resources is prevented by task-stealing technique of TBB scheduler

Generic Programming

- Best known example is C++ STL
- Enables distribution of broadly-useful high-quality algorithms and data structures
- Write best possible algorithm with fewest constraints
  - Do not force particular data structure on user
  - Classic example: STL std::sort
- Instantiate algorithm to specific situation
  - C++ template instantiation, partial specialization, and inlining make resulting code efficient
- Standard Template Library, overall, is not *thread-safe*

Generic Programming - Example

- Programmer defines the generic template, and the compiler creates versions for data types used.

T must define a copy constructor and a destructor

```
template <typename T> T max (T x, T y) {
  if (x < y) return y;
  return x;
}

int main() {
  int i = max(20,5);
  double f = max(2.5, 5.2);
  MyClass m = max(MyClass(“foo”),MyClass(“bar”));
  return 0;
}
```

T must define operator<

TBB Parallel patterns

- Task scheduler powers high level parallel patterns that are pre-packaged, tested, and tuned for scalability
  - parallel_for: parallel execution of independent loop iterations
  - parallel_reduce: parallel independent loop iterations that include a reduction.
  - parallel_do: load-balanced parallel execution of independent loop iterations with unknown or dynamically changing bounds (e.g. applying function to the element of linked list)
  - parallel_scan: template function that computes parallel prefix
  - pipeline: data-flow pipeline pattern
  - parallel_sort: parallel sort
  - parallel_invoke: evaluates up to 10 functions, possibly in parallel and waits for all of them to finish.

The parallel_for Template

```
template <typename Range, typename Body>
void parallel_for(const Range& range, const Body &body);
```

- Requires definition of:
  - A range type to iterate over
    - Must define a copy constructor and a destructor
    - Defines **is_empty()**
    - Defines **is_divisible()**
    - Defines a splitting constructor, **R(R &r, split)**
  - A body type that operates on the range (or a subrange)
    - Must define a copy constructor and a destructor
    - Defines **operator()**

Body is Generic

- Requirements for parallel_for Body

Body::Body(const Body&)	Copy constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*) const	Apply the body to *subrange*.

- parallel_for partitions original range into subranges, and deals out subranges to worker threads in a way that:
  - □ Balances load
  - □ Uses cache efficiently
  - □ Scales

Range is Generic

- Requirements for parallel_for Range

R::R (const R&)	Copy constructor
R::~R()	Destructor
bool R::is_empty() const	True if range is empty
bool R::is_divisible() const	True if range can be partitioned
R::R (R& r, split)	Splitting constructor; splits r into two subranges

- Library provides predefined ranges
  - □ blocked_range and blocked_range2d
- You can define your own ranges

An Example using parallel_for (1 of 3)

- Independent iterations and fixed/known bounds

```
const int N = 100000;

void change_array(float array, int M) {
    for (int i = 0; i < M; i++){
        array[i] *= 2;
    }
}

int main (){
    float A[N];
    initialize_array(A);
    change_array(A, N);
    return 0;
}
```

An Example using parallel_for (2 of 3)

- Include and initialize the library

```
#include “tbb/task_scheduler_init.h”
#include “tbb/blocked_range.h”
#include “tbb/parallel_for.h”

using namespace tbb;
```

```
int main (){
    float A[N];
    initialize_array(A);
    change_array(A, N);
    return 0;
}
```

```
int main (){
    task_scheduler_init init;
    float A[N];
    initialize_array(A);
    parallel_change_array(A, N);
    return 0;
}
```

Include Library Headers

Use namespace

Initialize scheduler

blue = original code
green = provided by TBB
red = boilerplate for library

An Example using parallel_for (3 of 3)

- Use the **parallel_for** algorithm

blue = original code
green = provided by TBB
red = boilerplate for library

```
void change_array(float *array, int M) {
    for (int i = 0; i < M; i++){
        array[i] *= 2;
    }
}
```

```
class ChangeArrayBody {
    float *array;
public:
    ChangeArrayBody (float *a): array(a) {}
    void operator()( const blocked_range <int>& r ) const{
        for (int i = r.begin(); i != r.end(); i++ ){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, int M) {
 parallel_for (blocked_range <int>(0, M),
              ChangeArrayBody(array), auto_partitioner());
}
```

Define Task

Use algorithm

Use auto_partitioner()

An Example using parallel_for (3b of 3)

blue = original code
green = provided by TBB
red = boilerplate for library

- Use the **`parallel_for`** algorithm

```
class ChangeArrayBody {
    float *array;
public:
    ChangeArrayBody (float *a): array(a) {}
    void operator()( const blocked_range <int>& r ) const{
        for (int i = r.begin(); i != r.end(); i++ ){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, int M) {
 parallel_for (blocked_range <int>(0, M),
               ChangeArrayBody(array),
               auto_partitioner());
}
```

The parallel_reduce Template

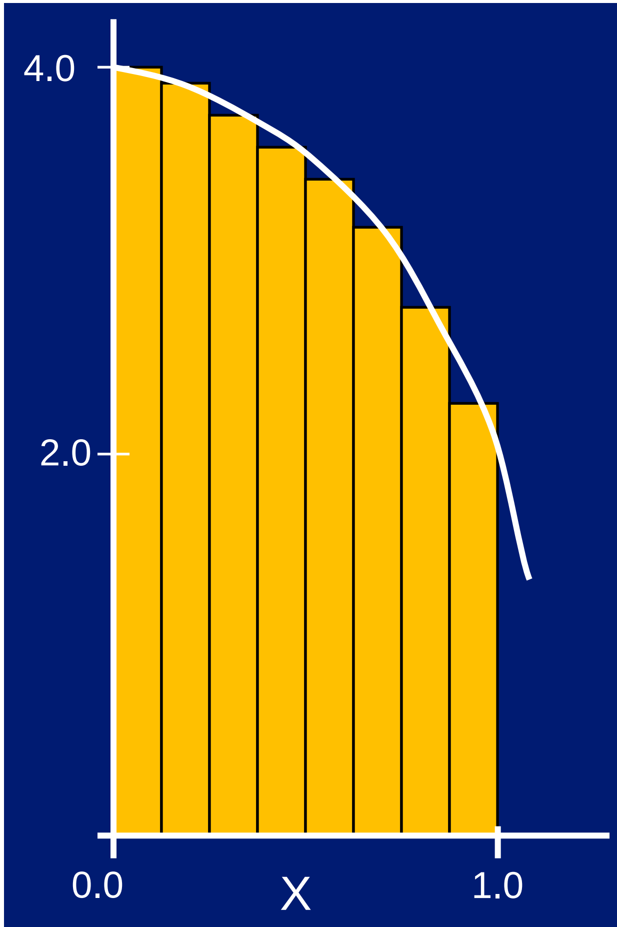
```
template <typename Range, typename Body>
void parallel_reduce (const Range& range, Body &body);
```

- Requirements for parallel_reduce Body

Body::Body(const Body&, split)	**Splitting constructor**
Body::~Body()	**Destructor**
void Body::operator() (Range& *subrange*) const	**Accumulate results from *subrange***
void Body::join(Body& *rhs*);	**Merge result of *rhs* into the result of this.**

- Reuses Range concept from parallel_for

Numerical Integration Example



```
static long num_steps=100000;
double step, pi;

void main(int argc, char*
argv[])
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf(“Pi = %f\n”,pi);
}
```

parallel_reduce Example

blue = original code
green = provided by TBB
red = boilerplate for library

```
#include "tbb/parallel_reduce.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"

using namespace tbb;

int main(int argc, char* argv[])
{
  double pi;
  double width = 1./(double)num_steps;
  MyPi step((double *const)&width);
  task_scheduler_init init;

  parallel_reduce(blocked_range<size_t>(0,num_steps), step,
                                                   auto_partitioner() );
  pi = step.sum*width;

  printf("The value of PI is %15.12f\n",pi);
  return 0;
}
```

parallel_reduce Example

```
class MyPi {
  double *const my_step;
public:
  double sum;
  void operator()( const blocked_range<size_t>& r ) {
    double step = *my_step;
    double x;
    for (size_t i=r.begin(); i!=r.end(); ++i)
    {
       x = (i + .5)*step;
       sum += 4.0/(1.+ x*x);
    }
  }

  MyPi( MyPi& x, split ) : my_step(x.my_step), sum(0) {}

  void join( const MyPi& y ) {sum += y.sum;}

  MyPi(double *const step) : my_step(step), sum(0) {}

};
```

blue = original code
green = provided by TBB
red = boilerplate for library

accumulate results

join

Scalable Memory Allocators

- Serial memory allocation can easily become a bottleneck in multithreaded applications
  - Threads require mutual exclusion into shared heap
- False sharing - threads accessing the same cache line
  - Even accessing distinct locations, cache line can ping-pong
- Intel® Threading Building Blocks offers two choices for scalable memory allocation
  - Similar to the STL template class **`std::allocator`**
  - **`scalable_allocator`**
    - Offers scalability, but not protection from false sharing
    - Memory is returned to each thread from a separate pool
  - **`cache_aligned_allocator`**
    - Offers both scalability and false sharing protection

Concurrent Containers

- TBB Library provides highly concurrent containers
  - STL containers are not concurrency-friendly: attempt to modify them concurrently can corrupt container
  - Standard practice is to wrap a lock around STL containers
    - Turns container into serial bottleneck
- Library provides fine-grained locking or lockless implementations
  - Worse single-thread performance, but better scalability.
  - Can be used with the library, OpenMP, or native threads.

Synchronization Primitives

- Parallel tasks must sometimes touch shared data
  - When data updates might overlap, use mutual exclusion to avoid race
- High-level generic abstraction for HW atomic operations
  - Atomically protect update of single variable
- Critical regions of code are protected by scoped locks
  - The range of the lock is determined by its lifetime (scope)
  - Leaving lock scope calls the destructor, making it exception safe
  - Minimizing lock lifetime avoids possible contention
  - Several mutex behaviors are available

Atomic Execution

- atomic<T>
  - □ T should be integral type or pointer type
  - □ Full type-safe support for 8, 16, 32, and 64-bit integers

Operations

‘= x’ and ‘x = ’	read/write value of x
x.fetch_and_store (y)	z = x, x = y, return z
x.fetch_and_add (y)	z = x, x += y, return z
x.compare_and_swap (y,p)	z = x, if (x==p) x=y; return z

```
atomic <int> i;
. . .
int z = i.fetch_and_add(2);
```

Mutex Concepts

- Mutexes are C++ objects based on scoped locking pattern
- Combined with locks, provide mutual exclusion

M()	**Construct unlocked mutex**
~M()	**Destroy unlocked mutex**
typename M::scoped_lock	**Corresponding scoped_lock type**
M::scoped_lock ()	**Construct lock w/out acquiring a mutex**
M::scoped_lock (M&)	**Construct lock and acquire lock on mutex**
M::~scoped_lock ()	**Release lock if acquired**
M::scoped_lock::acquire (M&)	**Acquire lock on mutex**
M::scoped_lock::release ()	**Release lock**

Mutex Flavors

- spin_mutex
  - Non-reentrant, unfair, spins in the user space
  - VERY FAST in lightly contended situations; use if you need to protect very few instructions
- queuing_mutex
  - Non-reentrant, fair, spins in the user space
  - Use Queuing_Mutex when scalability and fairness are important
- queuing_rw_mutex
  - Non-reentrant, fair, spins in the user space
- spin_rw_mutex
  - Non-reentrant, fair, spins in the user space
  - Use ReaderWriterMutex to allow non-blocking read for multiple threads

spin_mutex Example

```
#include "tbb/spin_mutex.h“
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreelistMutexType FreelistMutex;

Node* AllocateNode (){
  Node* n;
  {
    FreelistMutexType::scoped_lock mylock(FreeListMutex);
    n = FreeList;
    if ( n ) FreeList = n->next;
  }
  if ( !n ) n = new Node();
  return n;
}

void FreeNode( Node* n ) {
  FreelistMutexType::scoped_lock mylock(FreeListMutex);
  n->next = FreeList;
  FreeList = n;
}
```

blue = original code
green = provided by TBB
red = boilerplate for library

One last question…

How do I know how many threads are available?

- Do not ask!
  - Not even the scheduler knows how many threads really are available
    - There may be other processes running on the machine
  - Routine may be nested inside other parallel routines
- Focus on dividing your program into tasks of sufficient size
  - Task should be big enough to amortize scheduler overhead
  - Choose decompositions with good depth-first cache locality and potential breadth-first parallelism
- Let the scheduler do the mapping

a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

Parallel API's: MPI
the Message Passing Interface

MPI_Type_contiguous

MPI_Bcast

MPI_Recv_init

MPI_Group_size

MPI_S

RLD

MPI_

npare

C$(

artall

MPI_

_Pack

MPI_

k)

MPI: An API for Writing Clustered Applications

- A library of routines to coordinate the execution of multiple processes.
- Provides point to point and collective communication in Fortran, C and C++
- Unifies last 15 years of cluster computing and MPP practice

MPI_Sendrecv_replace

MPI_Ssend

MPI_Waitall

MPI_Alltoallv

MPI_Send

The minimal set of MPI functions

- There are hundreds of functions in MPI, but most programs use the following seven MPI functions:
  - `MPI_Init`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Recv`
  - `MPI_Reduce`
  - `MPI_Finalize`

Initializing the MPI Library

```
Fortran:
     MPI_INIT (ierr)

C:
     int MPI_Init (int* argc, char* argv[])
```

- **`MPI_Init`** prepares the system for MPI execution
- No MPI functions may be called before **`MPI_Init`**
- Almost all MPI functions return an error code (C), or an error variable. When debugging, first check their values.

Shutting Down MPI

```
Fortran:
      MPI_FINALIZE (ierr)

C:
      int MPI_Finalize (void)
```

- **`MPI_Finalize`** frees any memory allocated by the MPI library
- No MPI functions may be called after calling **`MPI_Finalize`**
- **`You should close every MPI program with a call to MPI_Finalize`**

Sizing the MPI Communicator

```
Fortran:
     MPI_COMM_SIZE (comm, size, ierr)
          integer :: comm, size, ierr
C:
     int MPI_Comm_size (
               MPI_Comm comm, int* size)
```

- **`MPI_Comm`**, an *opaque data type,* is defined in **`mpi.h.`** It defines a communication context (group of processes, given a particular name). Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the specified communicator

Determining MPI Process Rank

```
Fortran:
     MPI_COMM_RANK (comm, rank, ierr)
          integer :: comm, rank, ierr
C:
     int MPI_Comm_rank (
               MPI_Comm comm, int* rank)
```

- MPI_Comm_rank returns rank (sequence number) of calling process within the specified communicator
- Processes are numbered from 0 to N-1 in an N-process run

A trivial MPI program

- Almost all MPI programs start and end like this one …

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

int main (int argc, char* argv[])
{
   int numProc, myRank;

   MPI_Init (&argc, &argv); /* Initialize the library */
   MPI_Comm_rank (MPI_COMM_WORLD, &myRank); /* Who am I?” */
   MPI_Comm_size (MPI_COMM_WORLD, &numProc); /*How many? */

   printf (“Hello. Process %d of %d here.\n”, myRank, numProc);

   MPI_Finalize (); /* Wrap it up. */
}
```

Sending Data

```
Fortran:
      MPI_SEND (buf, count, datatype,
                          dest, tag, comm, ierr)
      <type> buf(*)
      integer :: count, datatype, ierr,
                          dest, tag, comm
C:
      int MPI_Send (void* buf, int count,
            MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm)
```

- **`MPI_Send`** performs a blocking send of the specified data ("count" copies of type "datatype," stored in "buf") to the specified destination (rank "dest" within communicator "comm"), with message ID "tag"

Receiving Data

```
Fortran:
      MPI_RECV (buf, count, datatype, source,
                  tag, comm, status, ierr)
      <type> buf(*)
      integer :: count, datatype, ierr, source,
                  tag, comm,
                  status(MPI_STATUS_SIZE)
C:
      int MPI_Recv (void* buf, int count,
            MPI_Datatype datatype, int source,
            int tag, MPI_Comm comm,
            MPI_Status* status)
```

- **`MPI_Recv`** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in “status”

Data Reduction

```
Fortran:
       MPI_REDUCE (sendbuf, recvbuf, count,
                    datatype, operation, root,
                    comm, ierr)
       <type> sendbuf(*), recvbuf(*)
       integer :: count, datatype, operation,
                    root, comm, ierr
C:
       int MPI_Reduce (void* sendbuf,
             void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op,
             int root, MPI_Comm comm)
```

- **`MPI_Reduce`** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **`MPI_Allreduce`** places result in all processes (avoid unless necessary)

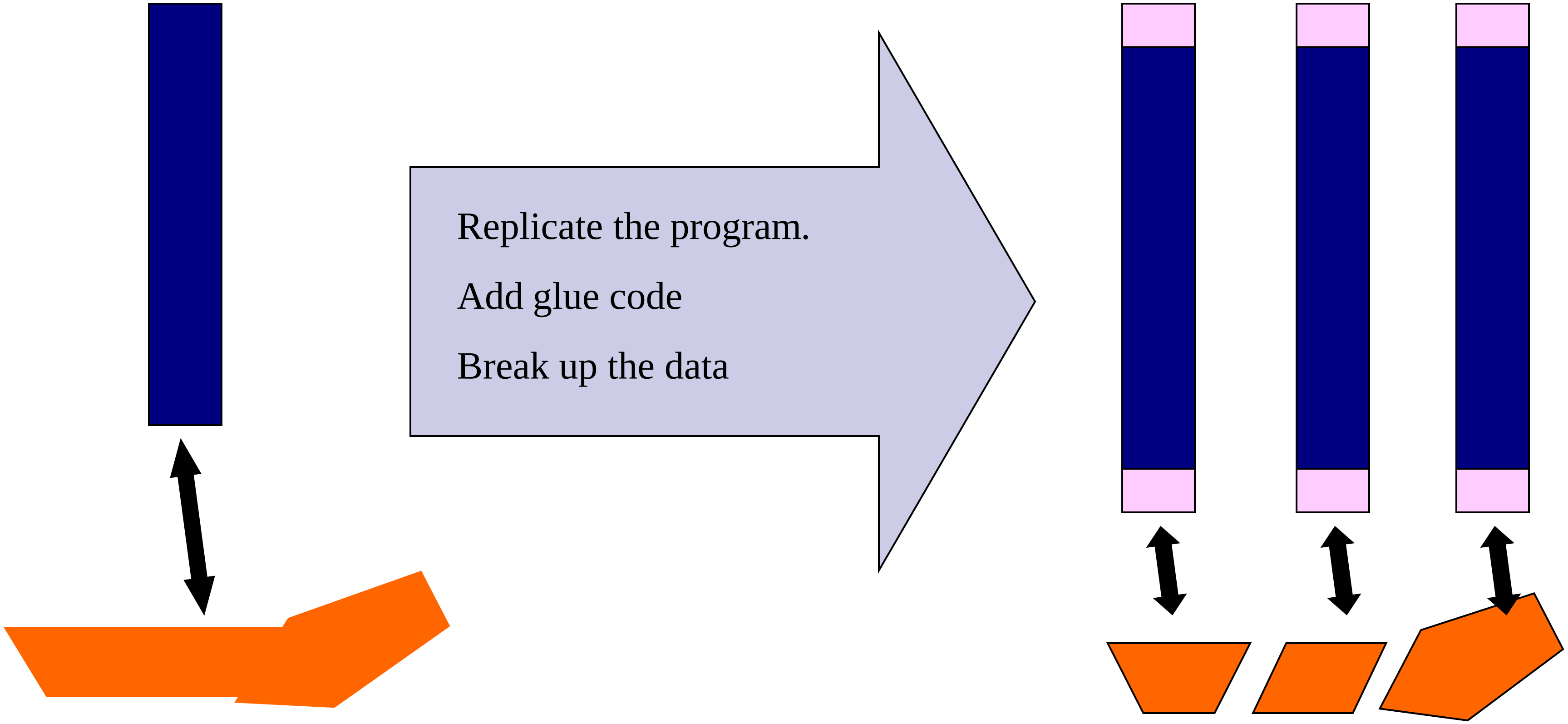
MPI Reduction Operations

Operation	**Function**
`MPI_SUM`	Summation
`MPI_PROD`	Product
`MPI_MIN`	Minimum value
`MPI_MINLOC`	Minimum value and location
`MPI_MAX`	Maximum value
`MPI_MAXLOC`	Maximum value and location
`MPI_LAND`	Logical AND
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

How do people use MPI? The SPMD Model

A sequential program working on a data set

- A parallel program working on a decomposed data set.
- Coordination by passing messages.



Replicate the program.

Add glue code

Break up the data

Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                 x = (i+0.5)*step;
                 sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

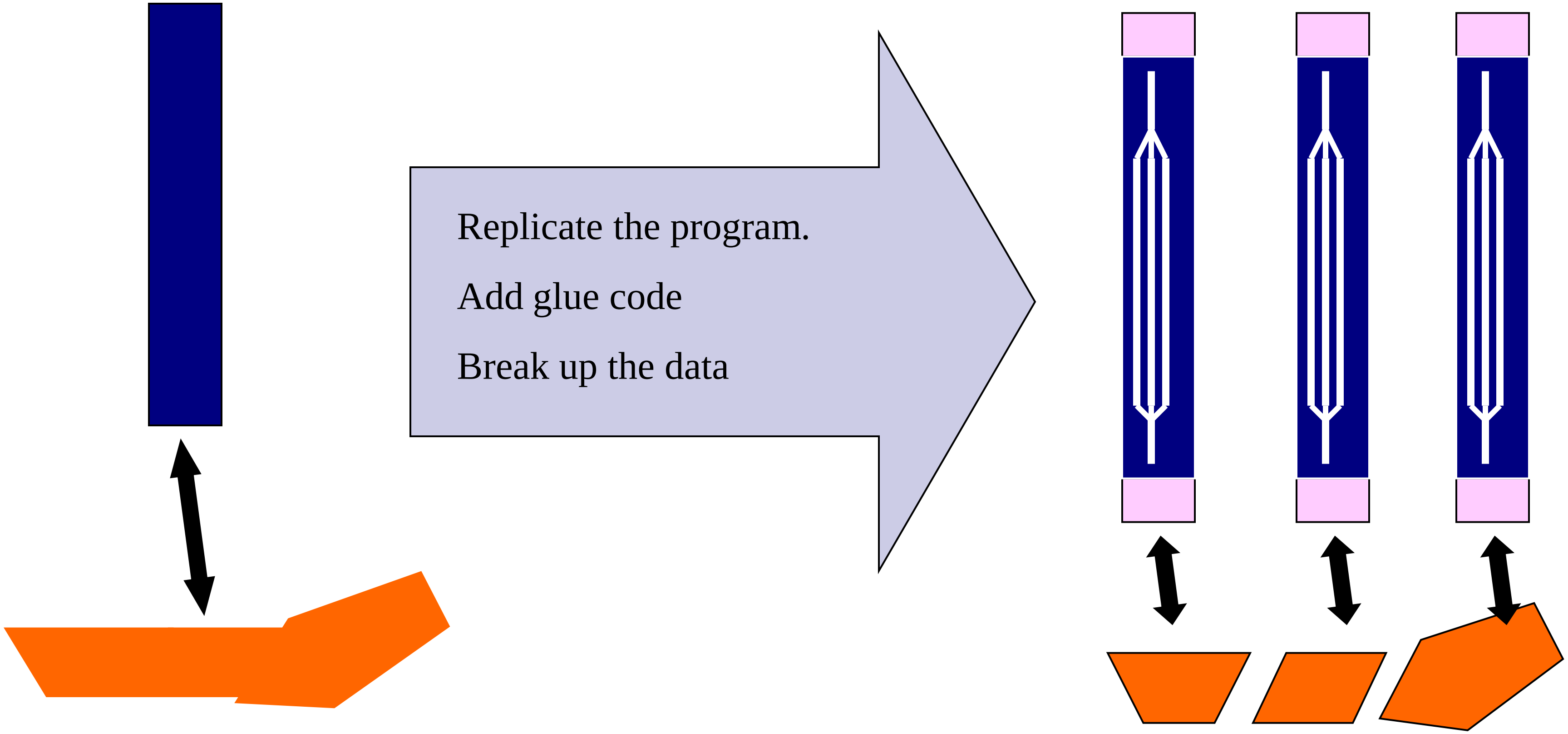
a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

How do people mix MPI and OpenMP?

A sequential program
working on a data set

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.



Replicate the program.

Add glue code

Break up the data

Pi program with MPI and OpenMP

```
#include <mpi.h>
#include “omp.h”
void main (int argc, char *argv[])
{
          int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
          step = 1.0/(double) num_steps ;
          MPI_Init(&argc, &argv) ;
          MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
          MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
          my_steps = num_steps/numprocs ;
#pragma omp parallel for reduction(+:sum) private(x)
          for (i=my_id*my_steps; i<(m_id+1)*my_steps ; i++)
          {
                    x = (i+0.5)*step;
                    sum += 4.0/(1.0+x*x);
          }
          sum *= step ;
          MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD) ;
}
```

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so

Key issues when mixing OpenMP and MPI

1. Messages are sent to a process not to a particular thread.
   - □ Not all MPIs are threadsafe. MPI 2.0 defines threading modes:
     - MPI_Thread_Single: no support for multiple threads
     - MPI_Thread_Funneled: Mult threads, only master calls MPI
     - MPI_Thread_Serialized: Mult threads each calling MPI, but they do it one at a time.
     - MPI_Thread_Multiple: Multiple threads without any restrictions
   - □ Request and test thread modes with the function:

     MPI_init_thread(desired_mode, delivered_mode, ierr)
2. Environment variables are not propagated by mpirun. You'll need to broadcast OpenMP parameters and set them with the library routines.

Dangerous Mixing of MPI and OpenMP

- **The following will work only if MPI_Thread_Multiple is supported … a level of support I wouldn’t depend on.**

```
MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;
#pragma omp parallel
{
    int tag, swap_neigh, stat, omp_id = omp_thread_num();
    long buffer [BUFF_SIZE], incoming [BUFF_SIZE];
    big_ugly_calc1(omp_id, mpi_id, buffer);
                                                          // Finds MPI id and tag so
    neighbor(omp_id, mpi_id, &swap_neigh, &tag);  // messages don’t conflict

    MPI_Send (buffer,   BUFF_SIZE, MPI_LONG, swap_neigh,
                 tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_LONG, swap_neigh,
                 tag,  MPI_COMM_WORLD, &stat);

    big_ugly_calc2(omp_id, mpi_id, incoming, buffer);
#pragma critical
   consume(buffer, omp_id, mpi_id);
}
```

Messages and threads

- Keep message passing and threaded sections of your program separate:
  - □ Setup message passing outside OpenMP parallel regions (MPI_Thread_funneled)
  - □ Surround with appropriate directives (e.g. critical section or master) (MPI_Thread_Serialized)
  - □ For certain applications depending on how it is designed it may not matter which thread handles a message. (MPI_Thread_Multiple)
    - Beware of race conditions though if two threads are probing on the same message and then racing to receive it.

Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

```
MPI_Init(&argc, &argv) ;     MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  big_calc(I);
}

    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, swap_neigh,
                tag, MPI_COMM_WORLD);
     MPI_Recv (incoming, buffer_count, MPI_DOUBLE, swap_neigh,
                tag,  MPI_COMM_WORLD, &stat);

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  other_big_calc(I, incoming);
}

consume(U, mpi_id);
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

```
MPI_Init(&argc, &argv) ;     MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++)    U[I] =  big_calc(I);

#pragma master
{
    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, neigh, tag,  MPI_COMM_WORLD);
     MPI_Recv (incoming, count, MPI_DOUBLE, neigh,  tag,  MPI_COMM_WORLD,
                                                                   &stat);
}
#pragma omp barrier
#pragma omp for
   for (I=0;I<N;I++)   U[I] =  other_big_calc(I, incoming);

#pragma omp master
   consume(U, mpi_id);
}
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Hybrid OpenMP/MPI works, but is it worth it?

- Literature* is mixed on the hybrid model: sometimes its better, sometimes MPI alone is best.
- There is potential for benefit to the hybrid model
  - MPI algorithms often require replicated data making them less memory efficient.
  - Fewer total MPI communicating agents means fewer messages and less overhead from message conflicts.
  - Algorithms with good cache efficiency should benefit from shared caches of multi-threaded programs.
  - The model maps perfectly with clusters of SMP nodes.
- But really, it’s a case by case basis and to large extent depends on the particular application.

*L. Adhianto and Chapman, 2007

a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

Overview of POSIX threads

- POSIX: Portable Operating System Interface for Unix
  - Interface to Operating System utilities

- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS Platforms

- Pthreads contain support for
  - Exploiting parallelism
  - Synchronization
  - No explicit support for communication … since it is a shared memory interface, a pointer to shared data is passed to a thread.

Forking POSIX threads

- Signature

  -
    ```
    int pthread_create (pthread_t *,
                        const pthread_attr_t *,
                        void * (*)(void *),
                        void *);
    ```
- Example call:

  -
    ```
    Errcode = pthread_create(&thread_id,
                             &thread_attribute,
                             &thread_fun,
                             &fun_arg);
    ```
    - **thread_id** is the thread id or handle (used to halt, etc.)
    - **thread_attribute** various attribtures
      - Standard default values obtained by passing a NULL pointer
      - Sample attribute: minimum stack size
    - **thread_fun** the function to bre run (takes and returns void*)
    - **Fun_arg** an argumnet can be passed to thread_fun when it starts
    - **Errorcode** will be set nonozero if the create operation fails.

Simple Threading Example

```
void * SayHello(void *foo) {
   printf( “hello, world\n”);
   return NULL;
}

int main(){
   pthread_t threads[16];
   int tn;
   for(tn=0; tn<16; tn++) {
      pthread_create(&threads[tn],  NULL, SayHello, NULL);
   }
   return 0;
}
```

Compile using gcc -lpthread

Shared data and threads

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if the pointer is passed)
- Variables on the statck are private; passing pointer to these around to other threads can cause problems
- Often done by creating a large “thread data” struct, which is passed into all threads as an argument

```
char *message = “hello world\n”;

pthread_create(&thread1, NULL,
               printf_fun, (void*) message);
```

Numerical Integration: POSIX Threads (1 of 2)

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

#define NUMSTEPS 10000000
#define NUMTHREADS 4
double gStep = 0.0, gPi = 0.0;

pthread_mutex_t gLock;

void *threadFunction(void *pArg)
{
   int myNum = *((int *)pArg);
   double partialSum = 0.0, x;  // local to each thread

   for (int i = myNum; i < NUMSTEPS; i += NUMTHREADS) // cyclic distribution
   {
      x = (i + 0.5f) * gStep;
      partialSum += 4.0f / (1.0f + x*x);  //compute partial sums at each thread
   }
   pthread_mutex_lock(&gLock);
     gPi += partialSum * gStep;  // add partial to global final answer
   pthread_mutex_unlock(&gLock);

   return 0;
}
```

Source: Michael Wrinn of Intel

Numerical Integration: POSIX Threads (2 of 2)

```
int main()
{
  pthread_t threadHandles[NUMTHREADS];
  int tNum[NUMTHREADS], i;

  pthread_mutex_init(&gLock, NULL);

  gStep = 1.0 / NUMSTEPS;
  for ( i = 0; i < NUMTHREADS; ++i )
  {
    tNum[i] = i;
    pthread_create(&threadHandles[i], NULL, threadFunction,
                                         (void)&tNum[i]);
  }
  for ( i = 0; i < NUMTHREADS; ++i )
  {
    pthread_join(threadHandles[i], NULL);
  }

  pthread_mutex_destroy(&gLock);
  printf("Computed value of Pi: %12.9f\n", gPi );
  return 0;
}
```

Source: Michael Wrinn of Intel

Some additional pthreads functions

- pthread_yield();
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.

- pthread_t me;   me = pthread_self();
  - Allows a pthread to obtain its own identifier pthread_t thread;

- pthread_detach(thread);
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance.

Setting Attribute values

- Once an initialized attribute object exists, changes can be made. For example
  - □ To change the stack size for a thread to 8192 (before calling pthread_create), do this:
    - pthread_attr_setstacksize(&my_attributes,(size_t)8192);
  - □ To get the stack size, do this:
    - Stack_t my_stack_size;
    - Pthread)_attr_getstacksize(&my_attributes, &my_stack__size);
- Other attributes
  - □ Guard size – use to protect against stack overflow.
  - □ Scheduling policiy – FIF”O or Round Robin
  - □ Lazy stack allocation – allocate on demand (lazy) or all at once , “up front”.
  - □ Scheduling parameters – in particular, thread priotity

Basic Synchronization: Mutexes

- Mutexes: mutual exclusion locks. Used to protect access to common data structures

- To create a mutex in Pthreads:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER
pthread_mutex_init (&amutex, NULL);
```

- To use the mutex:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To dealoocate a mutex

```
Int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Basic Synchronization: Barrier

- Barrier: Global Synchronization
  - □ Especially common with programs that utilize the SPMD pattern

```
pthread_t me;    me = pthread_self();
 work_on_subgrid(me);
 barrier;
 read_neighboring_values();
 barrier;
```

- Barriers in pthreads
  - □ Example of creating a barreir and initializing it for three threads.

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, 3);
```

  - □ Note: the NULL value in the second arguments specifies that the default attributes are to be used.
  - □ To wait at a barrier

```
pthread_barrier_wait(&b);
```

a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- ➡ Windows 32 threads
- Cilk
- OpenCL

Native Thread Libraries

- Linux and Windows both include native threads for shared address space programming
- API provides:
  - Thread creation (fork)
  - Thread destruction (join)
  - Synchronization.
- Programmer is in control … these are very general.
- Downside: programmer MUST control everything.

Solution: Win32 API, PI (fork/join pattern)

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
    int i, start;
  double x, sum = 0.0;


  start = *(int *) arg;
  step = 1.0/(double) num_steps;

  for (i=start;i<= num_steps; i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

Protect update to shared data

Put work into a function

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
            thread_handles[i] = CreateThread(0, 0,
                                      (LPTHREAD_START_ROUTINE) Pi,
                                      &threadArg[i], 0, &threadID);
}

  WaitForMultipleObjects(NUM_THREADS,
                                      thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Launch threads to execute the function

Wait until the threads are done

a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

Cilk in one slide

- Extends C to create a parallel language but maintains serial semantics.
- A fork-join style task oriented programming model perfect for recursive algorithms (e.g. branch-and-bound) … shared memory machines only!
- Solid theoretical foundation … can prove performance theorems.

cilk	**Marks a function as a "cilk" function that can be spawned**
spawn	**Spawns a cilk function … only 2 to 5 times the cost of a regular function call**
sync	**Wait until immediate children spawned functions return**

- "Advanced" key words

inlet	**Define a function to handle return values from a cilk task**
cilk_fence	**A portable memory fence.**
abort	**Terminate all currently existing spawned tasks**

- Includes locks and a few other odds and ends.

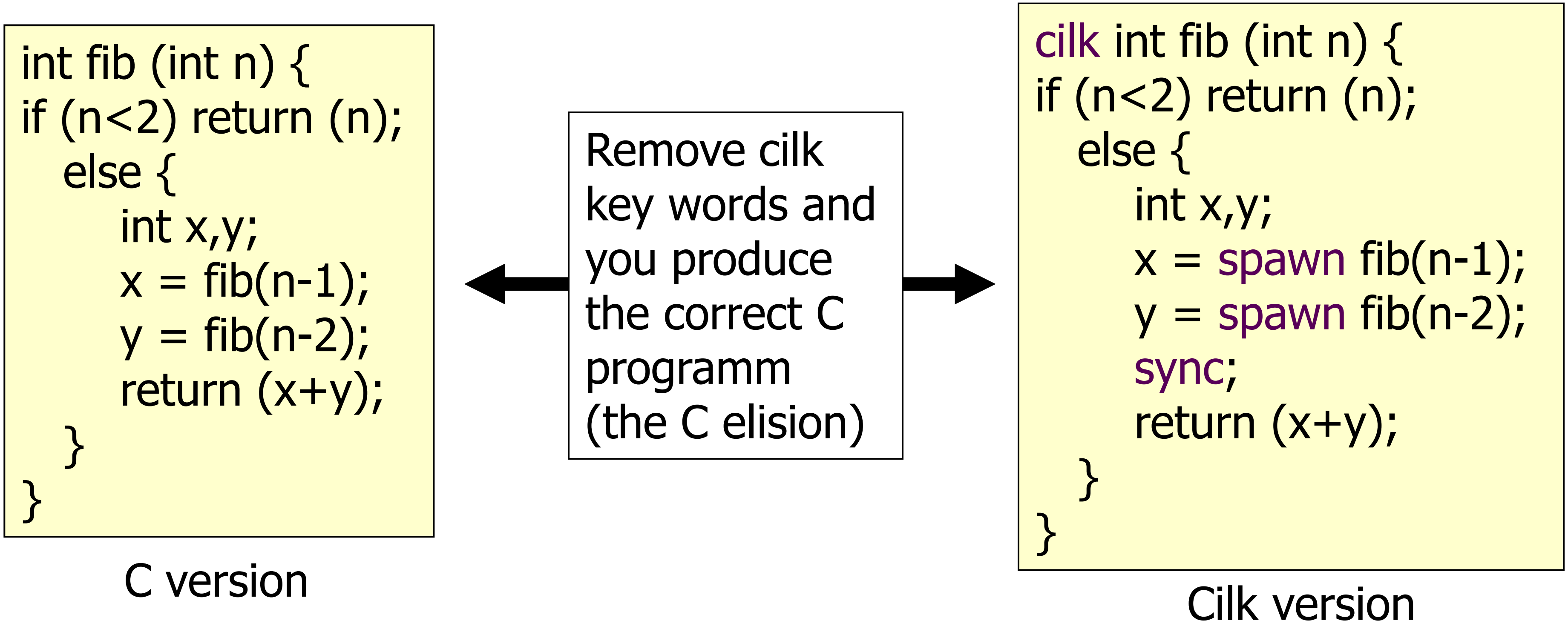
Recursion is at the heart of cilk

- Cilk makes it inexpensive to spawn new tasks.
- Instead of loops, recursively generate lots of tasks.
- Creates nested queues of tasks. A scheduler intelligently uses workstealing to keep all the cores busy as they work on these tasks.

With cilk, the programmer worries about expressing concurrency, not the details of how it is implemented

A simple Cilk example: Example

- Compute Fibonacci numbers … recursively split the problem until its small enough to compute directly



```
int fib (int n) {
if (n<2) return (n);
   else {
       int x,y;
       x = fib(n-1);
       y = fib(n-2);
       return (x+y);
   }
}
```

C version

Remove cilk key words and you produce the correct C programm (the C elision)

```
cilk int fib (int n) {
if (n<2) return (n);
   else {
       int x,y;
       x = spawn fib(n-1);
       y = spawn fib(n-2);
       sync;
       return (x+y);
   }
}
```

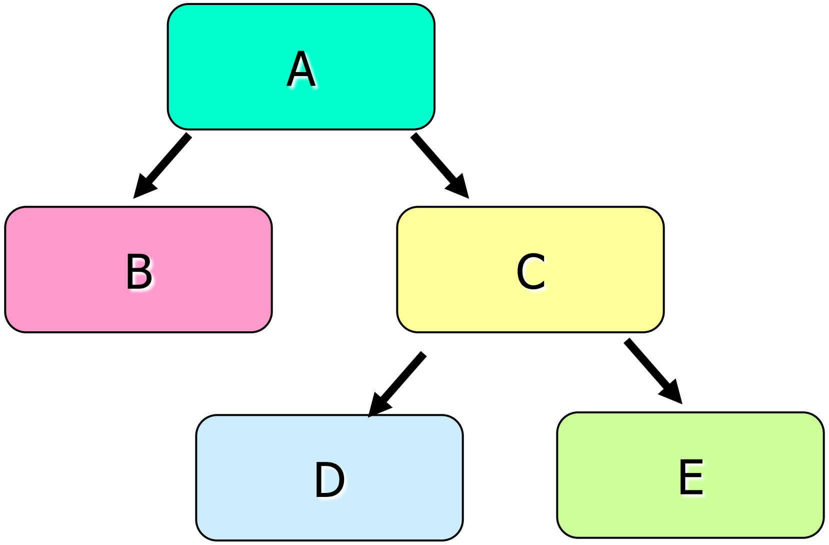
Cilk version

Cilk supports an incremental parallelism software methodology.

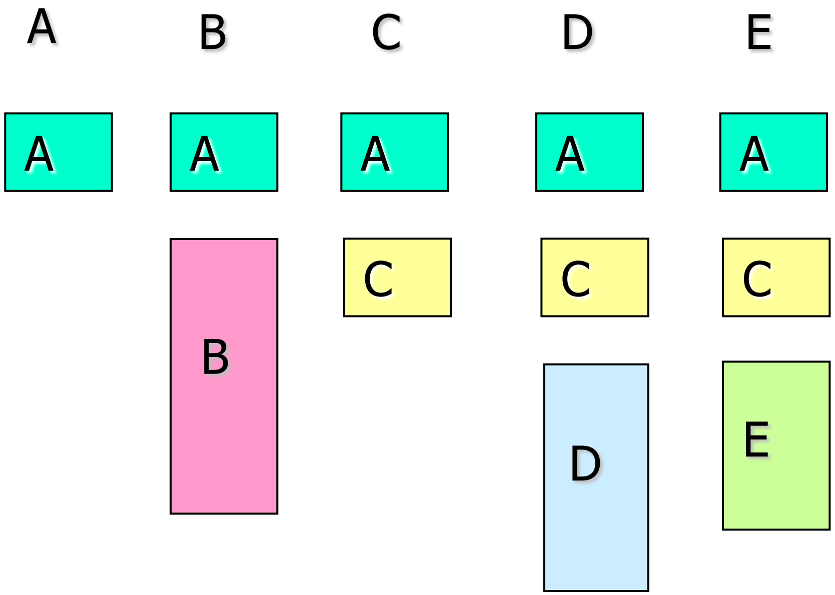
Based on Charles E. Leiserson, multithreaded programming in Cilk, lecture 1, July 13, 2006

Cactus stack

- Cilk supports C’s rule for pointers: a pointer to stack space can be passed from parent to child, but not from child to parent (Cilk also supports malloc)



Call Graph



Views of the stack

Based on Charles E. Leiserson, multithreaded programming in Cilk, lecture 1, July 13, 2006

Common pattern for Cilk

- Start with a program with a loop.

```
      void vadd (real *A, real *B, int n){
            int i; for(i=0; i<n; i++) A[i] +=
      B[i];
      }
```

- Convert to a recursive structure … splitting range in half until the remaining chunk is small enough to compute directly.
- Add Cilk keywords

```
cilk void vadd (real *A, real *B, int n){
         if (n<MIN) {
               int i; for(i=0; i<n; i++) A[i] +=
 B[i];
spawn} else {
spawn       vadd(A, B, n/2);
           sync; vadd(A+n/2, B+n/2, n-n/2);
         }
     }
```

PI Program: Cilk

```
static long num_steps = 1073741824;    // I’m lazy … make it a power of 2
double step = 1.0/(double) num_steps;
cilk double pi_comp(int istep, int nstep){
     double x, sum;
     if(nstep < MIN_SIZE)
                for (int i=istep, sum=0.0; i<= nstep; i++){
                              x = (i+0.5)*step;
                              sum += 4.0/(1.0+x*x);
                }
               return sum;
    else {
         sum1 = spawn pi_comp(istep, nstep/2);
         sum2 = spawn pi_comp(istep+nstep/2, nstep/2);
      }
     sync;
     return sum1+sum2;
}
int main ()
     double pi, sum = spawn pi_comp(0,num_steps);
      sync;
      pi = step * sum;
}
```

Recursively split range of the loop until its small enough to just directly compute

Wait until child tasks are done then return the sum … implements a balanced binary tree reduction!

Is Cilk efficient?

- It can be.
  - The recursive splitting procedure is usually cache friendly.
  - The scheduler does a great job of balancing the load
- The cilk scheduler …
  - The cilk scheduler maps tasks onto processors dynamically at runtime
  - The scheduler is provably good:
    - Each thread maintains a double ended queue (a deque) of work. Pulls work off the bottom of the queue.
    - When a queue is empty, it pulls work off the top of a randomly selected queue

inlets

- Inlets let you incorporate the result of a spawned task in a more complicated way than a simple assigment.

```
int max, ix = -1
inlet void update (int val, int index) {
   if(ix == -1 || val > max) {
       ix = index; max = val;
   }
}

for (i=0; i<1000000; i++) {
   update (spawn foo(i), i);
}
sync; // ix now indexes the largest foo(i)
```

The inlet keyword defines a void internal function to be an inlet

The “non-spawn” args to update() are evaluated,

Then the Cilk procedure foo(i) is spawned.

When foo(i) returns, update() is invoked

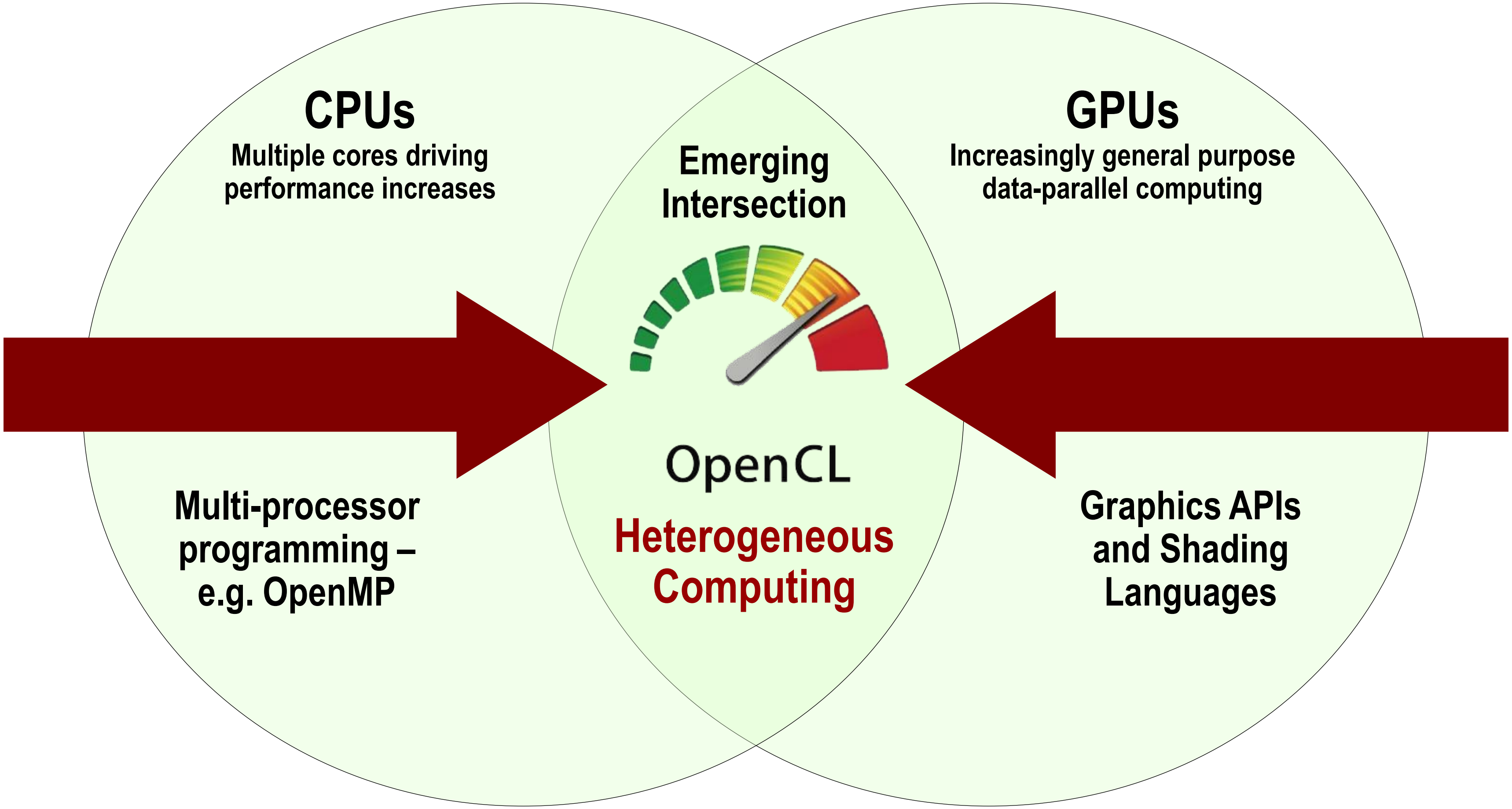
Cilk provides implicit atomicity among threads in the same frame, so no locking is necessary inside update to prevent races

Based on Charles E. Leiserson, multithreaded programming in Cilk, lecture 3, July 17, 2006

a Survey of programming models

- TBB
- MPI
- Mixing MPI and OpenMP
- Pthreads
- Windows 32 threads
- Cilk
- OpenCL

OpenCL: a language designed for the data-parallel index-map pattern



CPUs

Multiple cores driving performance increases

Emerging Intersection

GPUs

Increasingly general purpose data-parallel computing

Multi-processor programming – e.g. OpenMP

OpenCL

Heterogeneous Computing

Graphics APIs and Shading Languages

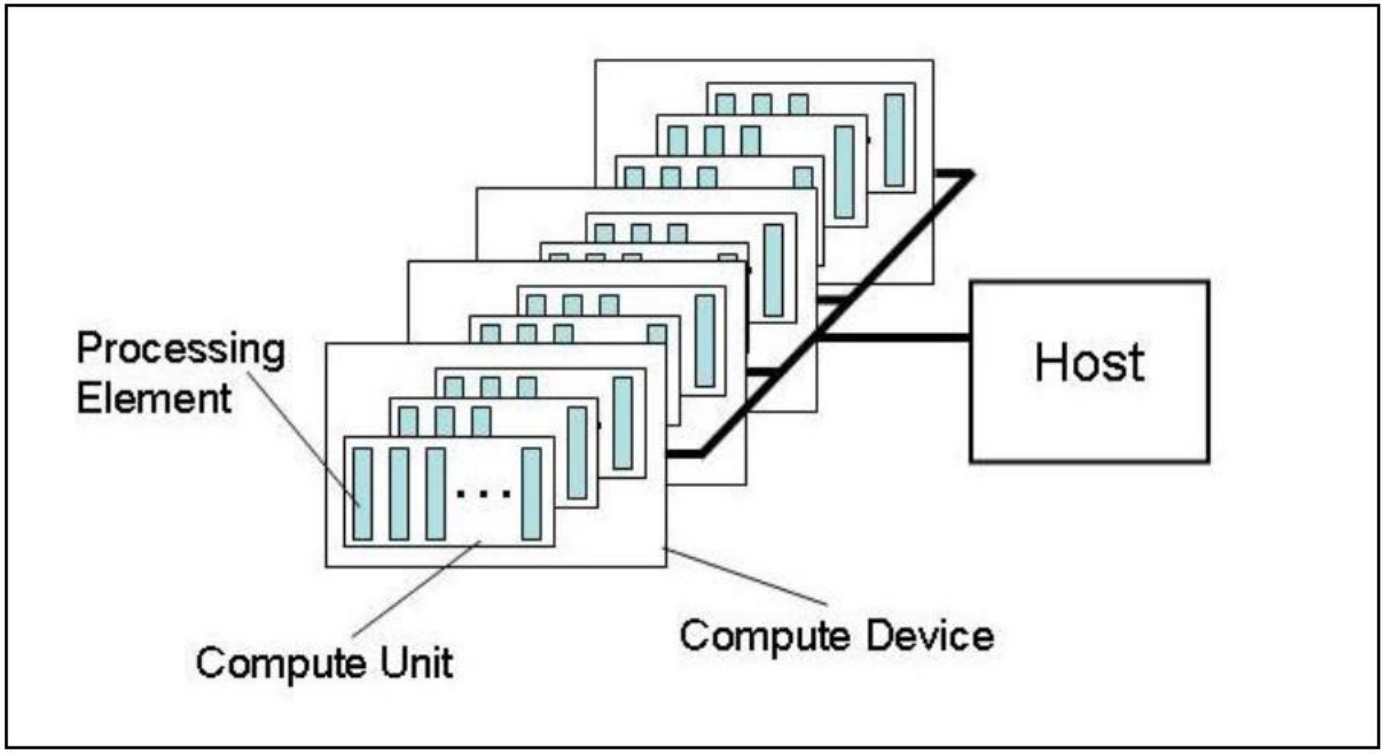
OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

3^{rd} party names are the property of their owners.

Source: SC09 OpenCL tutorial

OpenCL Platform Model



- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

Source: SC09 OpenCL tutorial

The data parallel index-map pattern:

- define a problem domain in terms of an index-map and execute a kernel invocation for each point in the domain
  - E.g., process a 1024 x 1024 image: Global problem dimensions:
    1024 x 1024 = 1 kernel execution per pixel: 1,048,576 total kernel executions

Scalar

```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

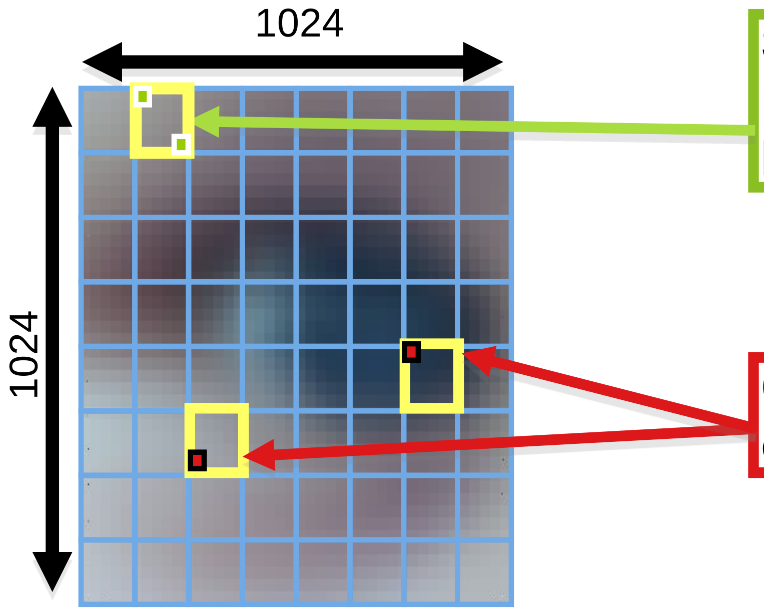
Data Parallel

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over “n” work-items
```

Source: SC09 OpenCL tutorial

An N-dimension domain of work-items

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (work group … executes together)



Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

- Choose the dimensions that are “best” for your algorithm

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
```

Define platform and queues

```
/                                                    ext
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
```

Define Memory objects

```
                                             READ_ONLY |
                                             , srcB,
   NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                              sizeof(cl_float)*n, NULL,
   NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

Create the program

```
//
er                                           , NULL, NULL,
   NULL);
```

Build the program

```
/
k                                                   LL);
```

Create and setup kernel

```
// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

```
// execute kernel
err = cl                                        1, 1,
   NULL,                                        L);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
   0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

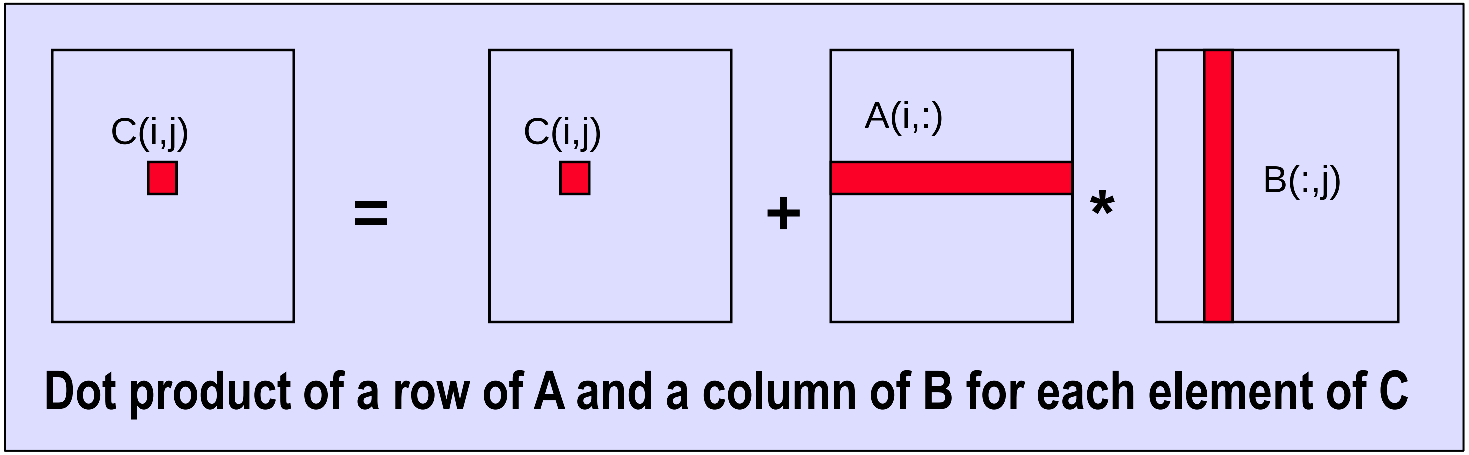
Execute the kernel

Read results on the host

It’s complicated, but most of this is “boilerplate” and not as bad as it looks.

Case Study: Matrix Multiplication: Sequential code

```
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
  int i, j, k;
  for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
      for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
      }
    }
  }
}
```



Dot product of a row of A and a column of B for each element of C

Source: SC10 OpenCL tutorial

Matrix Multiplications Performance

- **Basic, unoptimized results of C, serial matrix multiplication on a CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167

Run on an Apple MacBook Pro laptop running OSX 10 Snow Leopard.CPU is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Source: SC10 OpenCL tutorial

3rd party names are the property of their owners.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference <http://www.intel.com/performance> or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Matrix Multiplication: OpenCL kernel (1/4)

```
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++){
      for (j=0; j<Mdim; j++){
         for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```

Source: SC10 OpenCL tutorial

Matrix Multiplication: OpenCL kernel (2/4)

```
~~void mat_mul(~~
   ~~int Mdim, int Ndim, int Pdim,~~
   ~~float *A, float *B, float *C)~~
{
   int i, j, k;
   for (i=0; i<Ndim; i++){
      for (j=0; j<Mdim; j++){
         for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
```

Mark as a kernel function and specify memory qualifiers

3rd party names are the property of their owners.

Source: SC10 OpenCL tutorial

Matrix Multiplication: OpenCL kernel (3/4)

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++){
~~     for (j=0; j<Mdim; j++){~~
~~        for(k=0;k<Pdim,k++){~~     //C(i,j) = sum(over k) A(i,k) * B(k,j)
           C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
        }
~~     }~~
~~   }~~
}
```

```
   i = get_global_id(0);
   j = get_global_id(1);
```

Remove outer loops and set work item coordinates

Source: SC10 OpenCL tutorial

Matrix Multiplication: OpenCL kernel (4/4)

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
  i = get_global_id(0);
  j = get_global_id(1);
        for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
           C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
        }


}
```

Source: SC10 OpenCL tutorial

Matrix Multiplication: OpenCL kernel

Rearrange a bit and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp;
    tmp = 0.0;
       for(k=0;k<Pdim;k++)
         tmp += A[i*Ndim+k] * B[k*Pdim+j];
     C[i*Ndim+j] = tmp;
}
```

Source: SC10 OpenCL tutorial

Matrix Multiplications Performance

- **Basic results … no effort to optimize code.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
CPU: C(i,j) per work item, all global	744

Run on an Apple MacBook Pro laptop running OSX 10 Snow Leopard. GPU a GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units. CPU is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

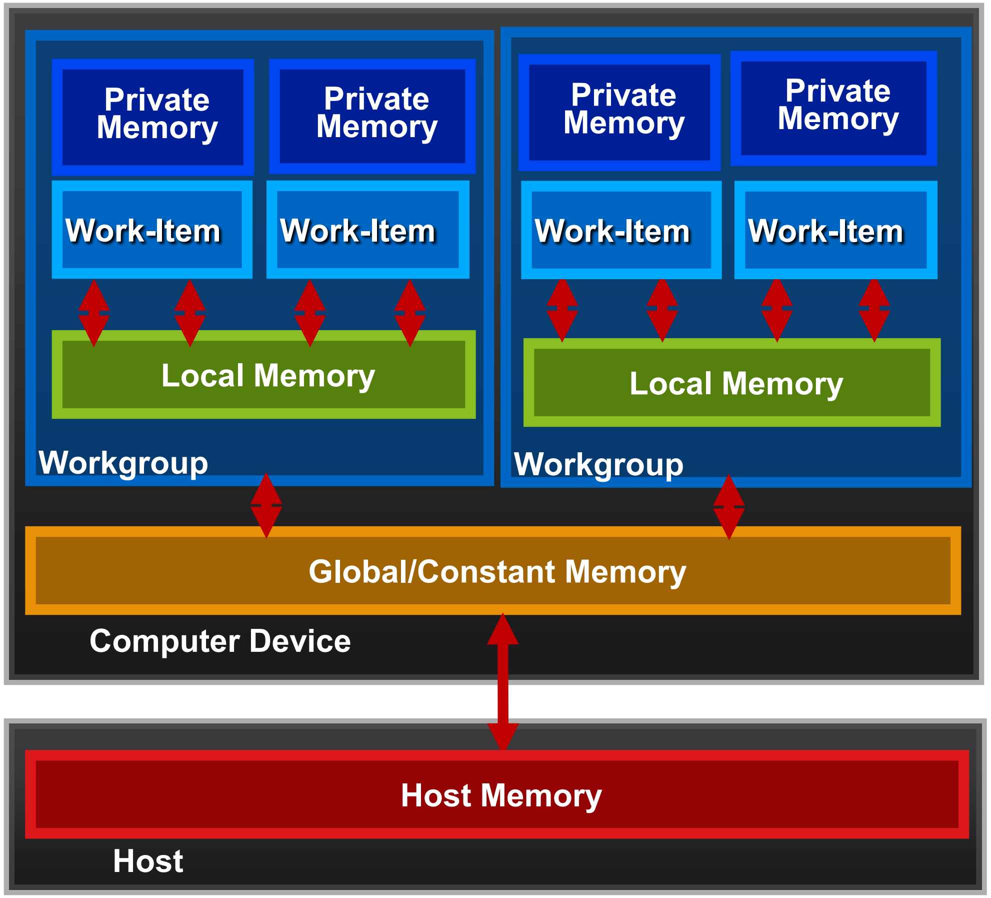
3rd party names are the property of their owners.

Source: SC10 OpenCL tutorial

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference <http://www.intel.com/performance> or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

OpenCL Memory Model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup
- Local Global/Constant Memory
  - Visible to all workgroups
- Host Memory
  - On the CPU

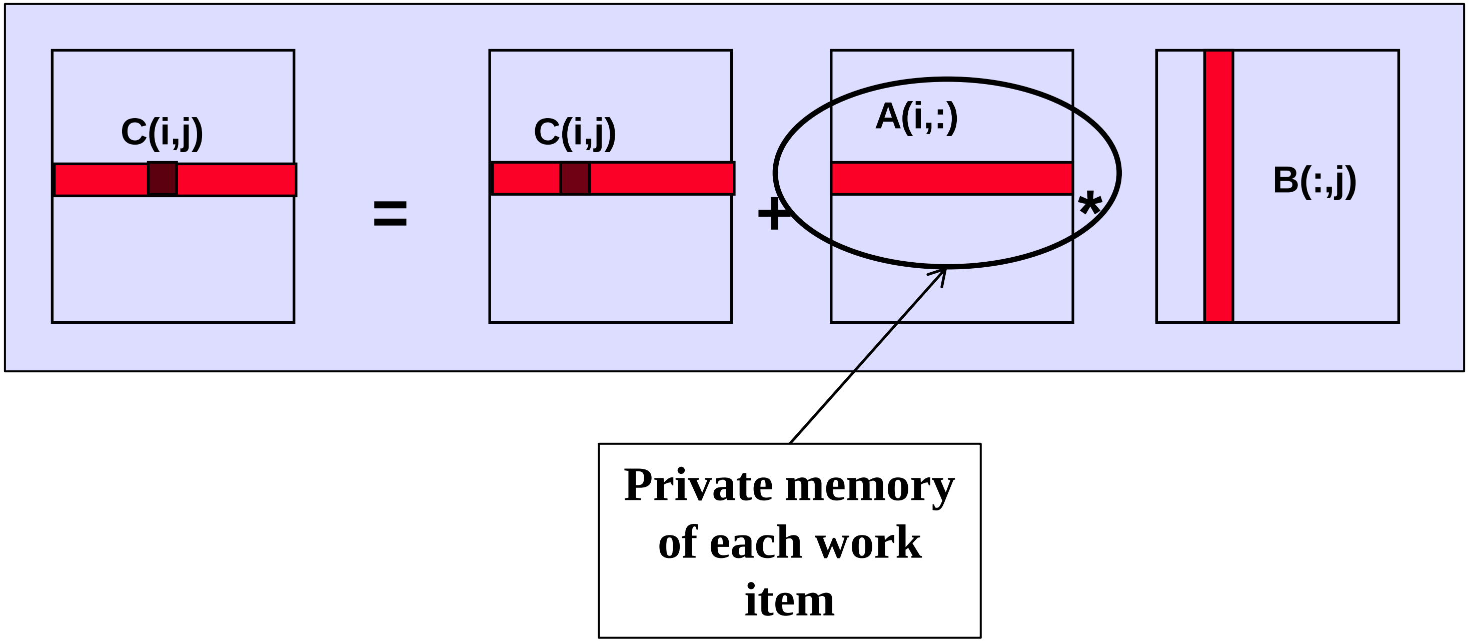


- Memory management is explicit
  You must move data from host -> global -> local *and* back

Source: SC09 OpenCL tutorial

Optimizing Matrix Multiplication

- Notice that each element of C in a row uses the same row of A.
- Let’s copy A into private memory so we don’t incur the overhead of pulling it from global memory for each C(i,j) computation.



Row of C per work item, A row private

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C)
{
    int k,j;
    int i = get_global_id(0);
    float Awrk[1000];
    float tmp;

    for(k=0;k<Pdim;k++)
                 Awrk[k] = A[i*Ndim+k];
    for(j=0;j<Mdim;j++){
         tmp = 0.0;
         for(k=0;k<Pdim;k++)
             tmp += Awrk[k] * B[k*Pdim+j];
         C[i*Ndim+j] = tmp;
     }
}
```

Setup a work array for A in private memory and copy into from global memory before we start with the matrix multiplications.

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
CPU: C(i,j) per work item	744

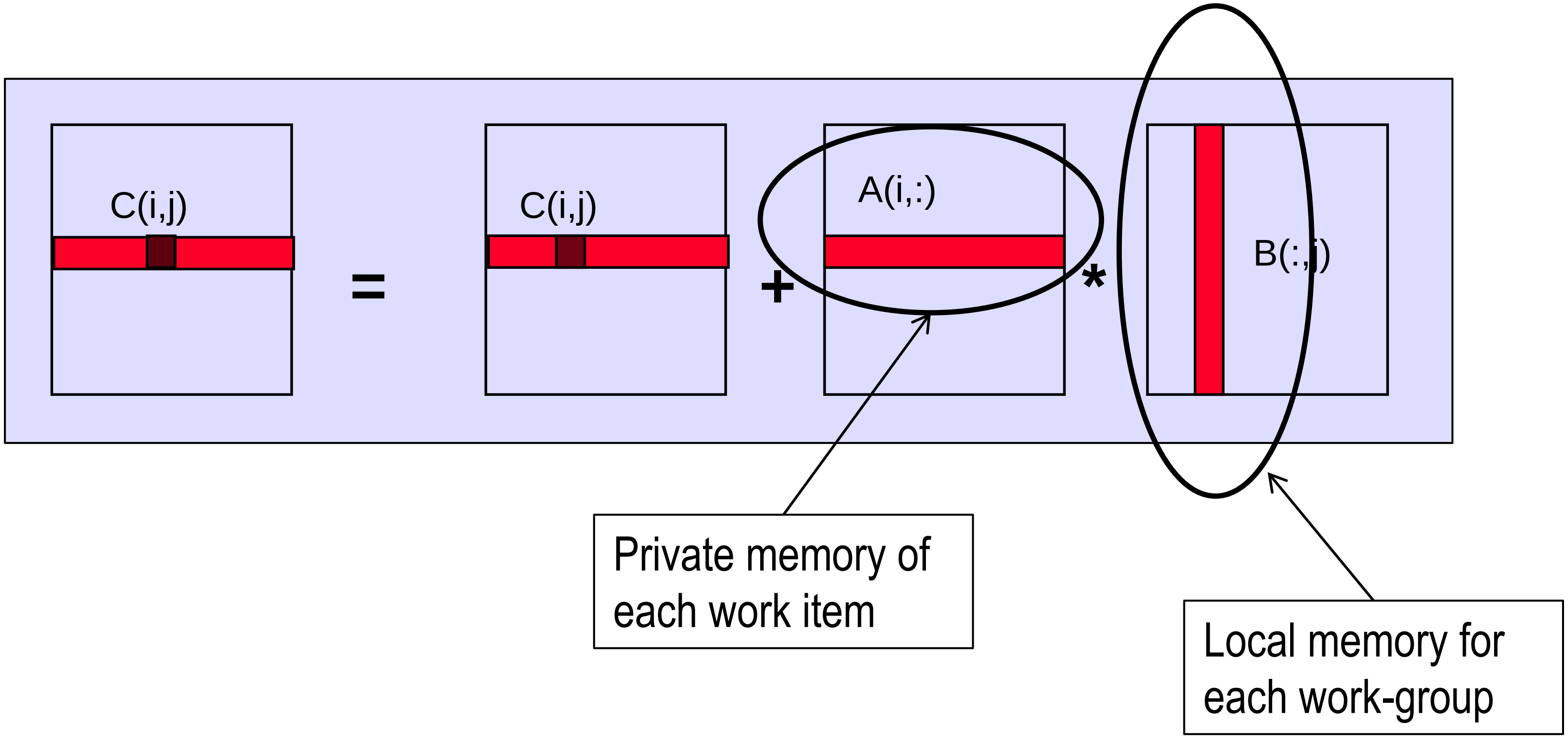
Big impact

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units

Device is Intel® Core™2 Duo CPU   T8300 @ 2.40GHz

Optimizing Matrix Multiplication

- Notice that each element of C uses the same row of A.
- Each work-item in a work-group uses the same columns of B
- Let’s store the B columns in local memory



Private memory of each work item

Local memory for each work-group

Row of C per work item, A row private, B columns local

```
__kernel mmul(
     const int Mdim,
     const int Ndim,
     const int Pdim,
     __global float* A,
     __global float* B,
     __global float* C,
     __local float* Bwrk)
{
    int k,j;
    int i = get_global_id(0);
    int iloc = get_local_id(0);
    int nloc = get_local_size(0);
    float Awrk[1000];
    float tmp;
for(k=0;k<Pdim;k++)
              Awrk[k] = A[i*Ndim+k];
       for(j=0;j<Mdim;j++){
       for(k=iloc;k<Pdim;k=k+nloc)
            Bwrk[k] = B[k*Pdim+j];
    barrier(CLK_LOCAL_MEM_FENCE);
            tmp = 0.0;
            for(k=0;k<Pdim;k++)
    tmp += Awrk[k] * Bwrk[k];
            C[i*Ndim+j] = tmp;
        }
}
```

Pass in a pointer to local memory. Work-items in a group start by copying the columns of B they need into the local memory.

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
GPU: C row per work item, A private, B local	2472
CPU: C(i,j) per work item	744

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	Speedup
CPU: Sequential C (not OpenCL)	1
GPU: C(i,j) per work item, all global	3
GPU: C row per work item, all global	1.5
GPU: C row per work item, A row private	5.2
GPU: C row per work item, A private, B local	15
CPU: C(i,j) per work item	4.5

Wow!!! OpenCL on a GPU is radically faster that C on a CPU, right?

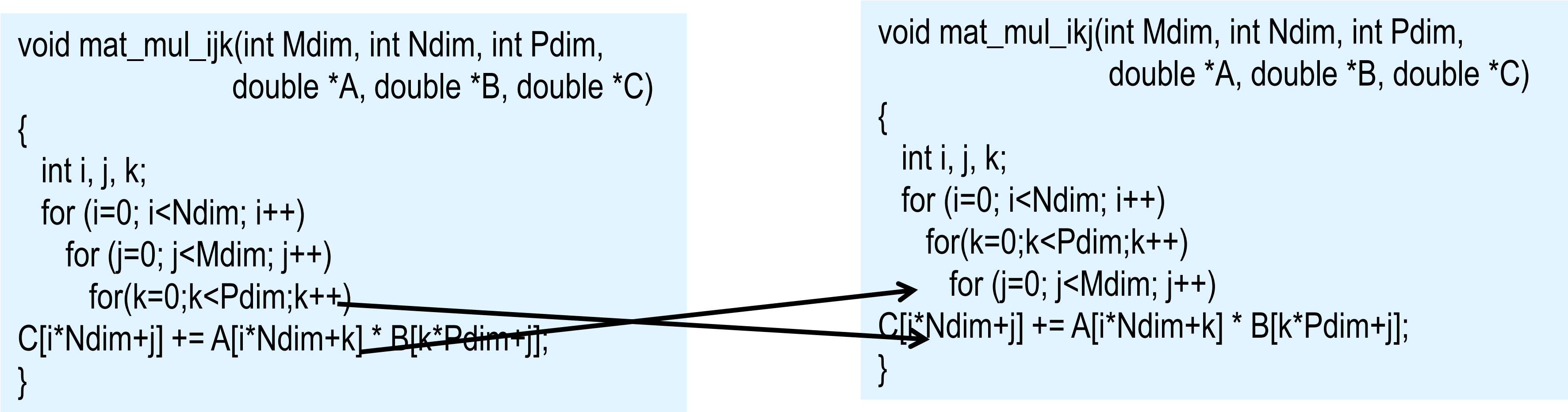
Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

3rd party names are the property of their owners.

CPU vs GPU: Let’s be fair

- **We made no attempt to optimize the CPU/C code but we worked hard to optimize OpenCL/GPU code.**
- **Lets optimize the CPU code**
  - Use compiler optimization (level O3).
  - Replace float with double (CPU ALU’s like double)
  - Reorder loops:

- Float, no opt	167 mflops
- Double, O3	272 mflops



```
void mat_mul_ijk(int Mdim, int Ndim, int Pdim,
                 double *A, double *B, double *C)
{
  int i, j, k;
  for (i=0; i<Ndim; i++)
    for (j=0; j<Mdim; j++)
      for(k=0;k<Pdim;k++)
C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
}
```

```
void mat_mul_ikj(int Mdim, int Ndim, int Pdim,
                 double *A, double *B, double *C)
{
  int i, j, k;
  for (i=0; i<Ndim; i++)
    for(k=0;k<Pdim;k++)
      for (j=0; j<Mdim; j++)
C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
}
```

- ijk:	272 mflops
- ikj:	1130 mflops
- kij:	481 mflops

Matrix Multiplications Performance

- **Results on an Apple laptop with an NVIDIA GPU and an Intel CPU.**

Case	Speedup
CPU: Sequential C (not OpenCL)	1
GPU: C(i,j) per work item, all global	0.45
GPU: C row per work item, all global	0.23
GPU: C row per work item, A row private	0.77
GPU: C row per work item, A private, B local	2.2
CPU: C(i,j) per work item	0.66

And we still are only using one core … and we are not using SSE so there is lots of room to further optimize the CPU code.

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Matrix Multiplications Performance

- **After we optimize to increase flops per memory access, we get the following results.**

Case	MFLOPS
CPU: Sequential C (not OpenCL)	1130
GPU: C(i,j) per work item, all global	511
GPU: C row per work item, all global	258
GPU: C row per work item, A row private	873
GPU: C row per work item, A private, B local	2472
CPU: C(i,j) per work item	744

Run on an Apple MacBook Pro laptop running OSX 10 Snow Leopard. GPU is a GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units. CPU is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

Source: SC10 OpenCL tutorial

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference <http://www.intel.com/performance> or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Conclusion

- We have now covered the full sweep of the OpenMP specification.
  - We've left off some minor details, but we've covered all the major topics … remaining content you can pick up on your own.
- Download the spec to learn more … the spec is filled with examples to support your continuing education.
  - www.openmp.org
- Get involved:
  - get your organization to join the OpenMP ARB.
  - Work with us through Compunity.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

OpenMP Organizations

- OpenMP architecture review board URL, the “owner” of the OpenMP specification:

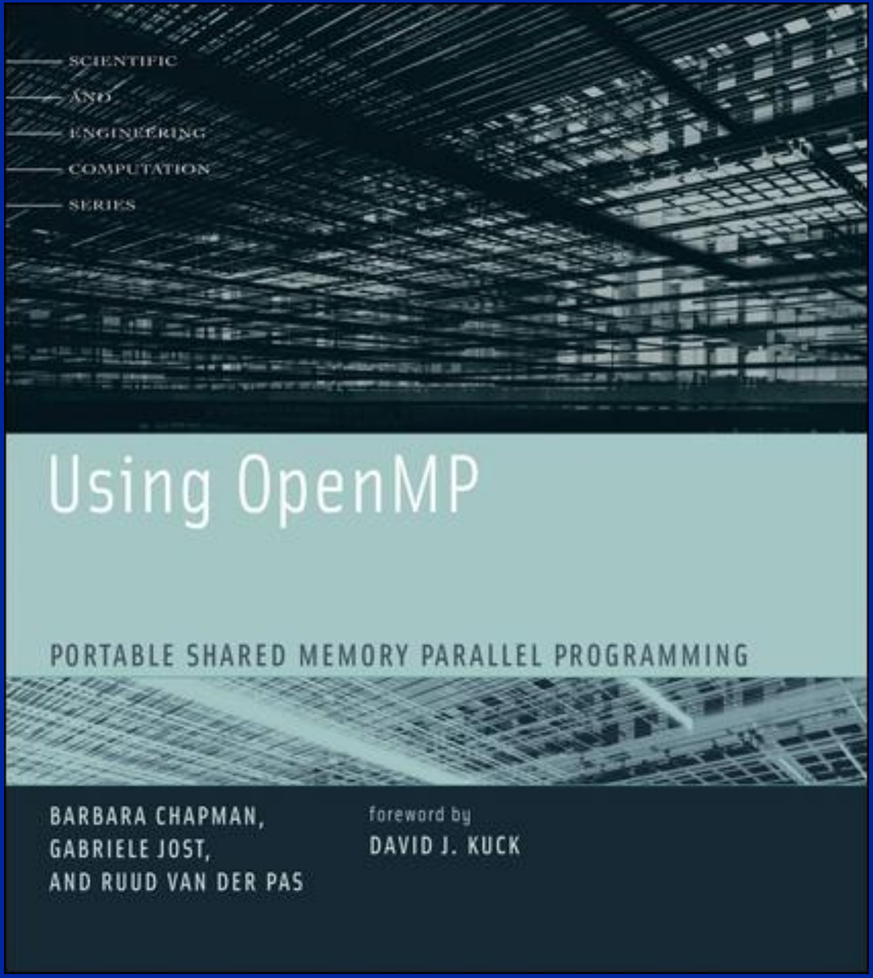
  www.openmp.org

- OpenMP User’s Group (cOMPunity) URL:

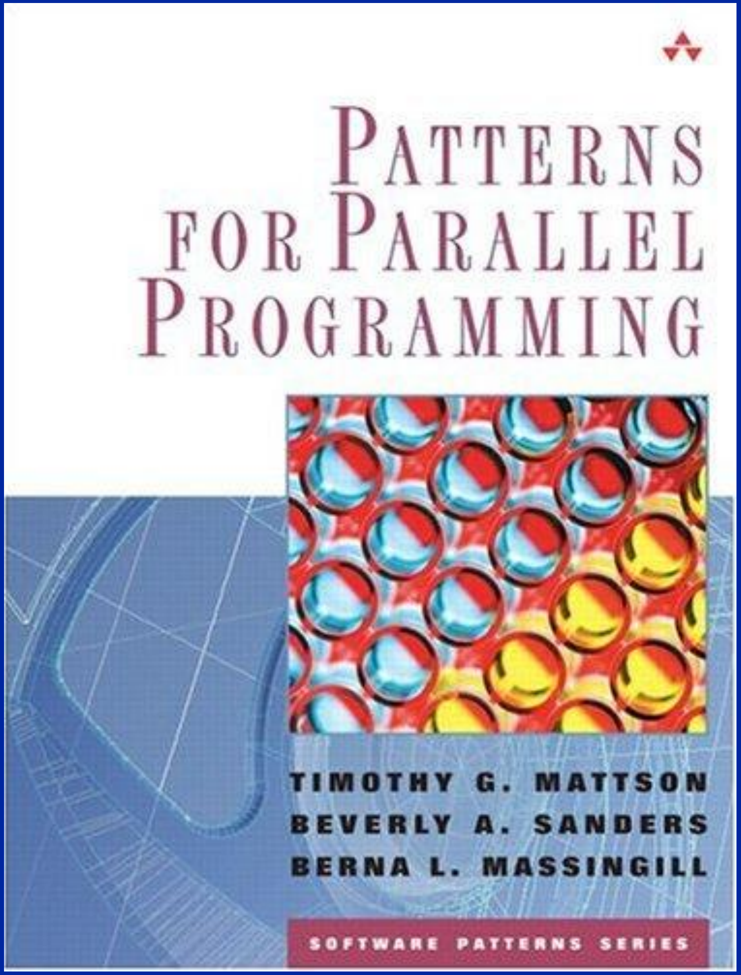
  www.compunity.org

Get involved, join compunity and help define the future of OpenMP

Books about OpenMP



- A new book about OpenMP 2.5 by a team of authors at the forefront of OpenMP's evolution.



- A book about how to “think parallel” with examples in OpenMP, MPI and java

OpenMP Papers

- Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.

- Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.

- Bentz J., Kendall R., “Parallelization of General Matrix Multiply Routines Using OpenMP”, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 1, 2005

- Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.

- Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.

- Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp.566-71. Cary, NC, USA.

OpenMP Papers (continued)

- Jost G., Labarta J., Gimenez J., What Multilevel Parallel Programs do when you are not watching: a Performance analysis case study comparing MPI/OpenMP, MLP, and Nested OpenMP, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 29, 2005
- Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40.
- Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.
- Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini. Parallel Programming with Message Passing and Directives; SIAM News, Volume 32, No 9, Nov. 1999.
- Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Lecture Notes in Computer Science Vol.1662. Springer-Verlag. 1999, pp.339-50.
- Liu Z., Huang L., Chapman B., Weng T., Efficient Implementationi of OpenMP for Clusters with Implicit Data Distribution, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 121, 2005

OpenMP Papers (continued)

- B. Chapman, F. Bregier, A. Patil, A. Prabhakar, "Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems," *Concurrency and Computation: Practice and Experience.* 14(8-9): 713-739, 2002.
- J. M. Bull and M. E. Kambites. JOMP: an OpenMP-like interface for Java. Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.
- L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid MPI and OpenMP applications, Simulation Modeling Practice and Theory, vol 15, p. 481-491, 2007.
- Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP; Concurrency: Practice and Experience, 2000; 12:1219-1239. Publisher John Wiley & Sons, Ltd.
- Mattson, T.G., How Good is OpenMP? Scientific Programming, Vol. 11, Number 2, p.81-93, 2003.
- Duran A., Silvera R., Corbalan J., Labarta J., "Runtime Adjustment of Parallel Nested Loops", Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 137, 2005

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Exercise 1: Solution

A multi-threaded “Hello world” program

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

```
#include “omp.h”
void 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)
```

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - ➡ Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

The SPMD pattern

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region “near the top of the code”.
  - Pick up thread ID and num_threads.
  - Use them to split up loops and select different blocks of data to work on.

Exercise 2: A simple SPMD 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

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - ➡ Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

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.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Exercise 3: SPMD Pi without false sharing

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

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here.   Must protect summation into pi in a critical region so updates don’t conflict

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Exercise 4: solution

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{           int i;     double x, pi, sum = 0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction(+:sum)
            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 4 simple lines!

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Matrix multiplication

```
#pragma omp parallel for private(tmp, i, j, k)
   for (i=0; i<Ndim; i++){
         for (j=0; j<Mdim; j++){
                  tmp = 0.0;
                  for(k=0;k<Pdim;k++){
                           /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
                           tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
                  }
                  *(C+(i*Ndim+j)) = tmp;
         }
   }
```

- On a dual core laptop
  - 13.2 seconds 153 Mflops one thread
  - 7.5 seconds 270 Mflops two threads

Results on an Intel dual core 1.83 GHz CPU,   Intel IA-32 compiler 10.1 build 2

Appendices

- Sources for Additional information
- Solutions to exercises
    - Exercise 1: hello world
    - Exercise 2: Simple SPMD Pi program
    - Exercise 3: SPMD Pi without false sharing
    - Exercise 4: Loop level Pi
    - Exercise 5: Matrix multiplication
    - ➡ Exercise 6: Mandelbrot area
    - Exercise 7: Molecular dynamics
    - Exercise 8: linked lists with tasks
    - Exercise 9: linked lists without tasks
    - Exercise 10: the producer-consumer pattern
- Thread Private Data
    - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Exercise 6: Area of a Mandelbrot set

- Solution is in the file mandel_par.c
- Errors:
  - Eps is private but uninitialized. Two solutions
    - It's write-only so you can make it shared.
    - Make it firstprivate
  - The loop index variable j is shared by default. Make it private.
  - The variable c has global scope so "testpoint" may pick up the global value rather than the private value in the loop. Solution … pass C as and arg to testpoint
  - Updates to "numoutside" are a race. Protect with an atomic.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Exercise 7 solution

Compiler will warn you if you have missed some variables

```
#pragma omp parallel for default (none) \
    shared(x,f,npart,rcoff,side) \
    reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
     .........
```

Loop is not well load balanced: best schedule has to be found by experiment.

See forces.c in MolDynSoln1

Exercise 7 solution (cont.)

```
……..
#pragma omp atomic
          f[j]    -= forcex;
#pragma omp atomic
          f[j+1]  -= forcey;
#pragma omp atomic
          f[j+2]  -= forcez;
        }
      }
#pragma omp atomic
      f[i]     += fxi;
#pragma omp atomic
      f[i+1]   += fyi;
#pragma omp atomic
      f[i+2]   += fzi;
    }
  }
```

All updates to f must be atomic

See forces.c in MolDynSoln1

Exercise 7 with orphaning

Move the parallel construct into Main to reduce overhead from creating/suspending threads for each call to force()

```
#pragma omp single
{
   vir    = 0.0;
   epot   = 0.0;
}
#pragma omp for reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
……….
```

Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

See forces.c in MolDynSoln2

Exercise 7 reduce sync overhead

```
        ftemp[myid][j]   -= forcex;
        ftemp[myid][j+1]  -= forcey;
        ftemp[myid][j+2]  -= forcez;
      }
    }
   ftemp[myid][i]       += fxi;
   ftemp[myid][i+1]      += fyi;
   ftemp[myid][i+2]      += fzi;
 }
```

Replace atomics with accumulation into array with extra dimension

See forces.c in MolDynSoln3

Exercise 7 The reduction step

```
….
#pragma omp for
   for(int i=0;i<(npart*3);i++){
       for(int id=0;id<nthreads;id++){
          f[i] += ftemp[id][i];
          ftemp[id][i] = 0.0;
       }
   }
```

Reduction can be done in parallel

Zero ftemp for next time round

See forces.c in MolDynSoln3

Appendices

- Sources for Additional information
- Solutions to exercises
    - Exercise 1: hello world
    - Exercise 2: Simple SPMD Pi program
    - Exercise 3: SPMD Pi without false sharing
    - Exercise 4: Loop level Pi
    - Exercise 5: Matrix multiplication
    - Exercise 6: Mandelbrot area
    - Exercise 7: Molecular dynamics
    - ➡ Exercise 8: linked lists with tasks
    - Exercise 9: linked lists without tasks
    - Exercise 10: the producer-consumer pattern
- Thread Private Data
    - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Linked lists with tasks (OpenMP 3)

- See the file Linked_omp3_tasks.c

```
#pragma omp parallel
{
  #pragma omp single
 {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
              processwork(p);
        p = p->next;
    }
  }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Mandelbrot area
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - ➡ Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Linked lists without tasks

- See the file Linked_omp25.c

```
while (p != NULL) {
    p = p->next;
     count++;
}
p = head;
for(i=0; i<count; i++) {
     parr[i] = p;
     p = p->next;
  }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

```
 std::vector<node *> nodelist;
 for (p = head; p != NULL; p = p->next)
    nodelist.push_back(p);

int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
    for (int i = 0; i < j; ++i)
          processwork(nodelist[i]);
```

Copy pointer to each node into an array

Count number of items in the linked list

Process nodes in parallel with a for loop

	C++, default sched.	C++, (static,1)	C, (static,1)
One Thread	37 seconds	49 seconds	45 seconds
Two Threads	47 seconds	32 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU,   Intel IA-32 compiler 10.1 build 2

Appendices

- Sources for Additional information
- Solutions to exercises
    - Exercise 1: hello world
    - Exercise 2: Simple SPMD Pi program
    - Exercise 3: SPMD Pi without false sharing
    - Exercise 4: Loop level Pi
    - Exercise 5: Matrix multiplication
    - Exercise 6: Mandelbrot area
    - Exercise 7: Molecular dynamics
    - Exercise 8: linked lists with tasks
    - Exercise 9: linked lists without tasks
    - Exercise 10: the producer-consumer pattern
- Thread Private Data
    - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Exercise 10: producer consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));

   #pragma omp parallel sections
   {
     #pragma omp section
      {
         fill_rand(N, A);
         #pragma omp flush
         flag = 1;
         #pragma omp flush (flag)
      }
      #pragma omp section
      {
         #pragma omp flush (flag)
         while (flag != 1){
              #pragma omp flush (flag)
         }
         #pragma omp flush
         sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

Appendices

- Sources for Additional information
- Solutions to exercises
    - ◆ Exercise 1: hello world
    - ◆ Exercise 2: Simple SPMD Pi program
    - ◆ Exercise 3: SPMD Pi without false sharing
    - ◆ Exercise 4: Loop level Pi
    - ◆ Exercise 5: Matrix multiplication
    - ◆ Exercise 7: Molecular dynamics
    - ◆ Exercise 8: linked lists with tasks
    - ◆ Exercise 9: linked lists without tasks
    - ◆ Exercise 10: the producer-consumer pattern
- ➡ Thread Private Data
    - ◆ Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- Compiler Notes

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
   counter++;
   return (counter);
}
```

Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

Data Copying: Copyprivate

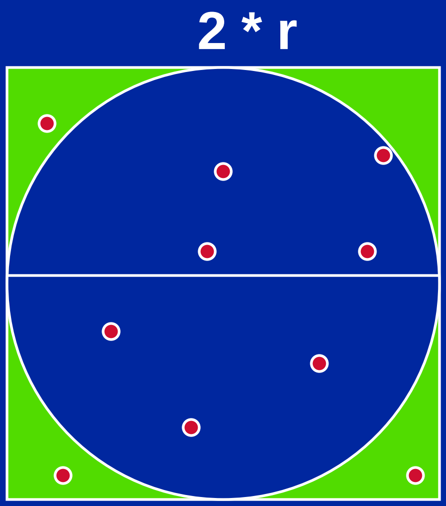
Used with a single region to broadcast values of privates from one member of a team to the rest of the team.

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
  int Nsize, choice;

  #pragma omp parallel private (Nsize, choice)
  {
      #pragma omp single copyprivate (Nsize, choice)
            input_parameters (Nsize, choice);

      do_work(Nsize, choice);
  }
}
```

Exercise 11: Monte Carlo Calculations

Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Exercise 11

- We provide three files for this exercise
  - pi_mc.c: the monte carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be threadsafe.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

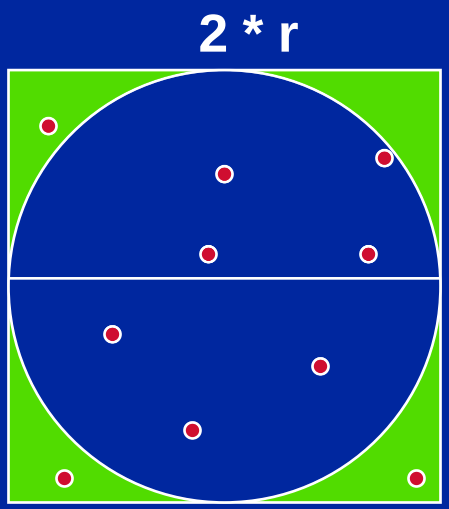
Computers and random numbers

- We use "dice" to make random numbers:
  - Given previous values, you cannot predict the next value.
  - There are no patterns in the series … and it goes on forever.
- Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  - By design, computers are not random and cannot produce random numbers.
- However, with some very clever programming, we can make "pseudo random" numbers that are as random as you need them to be … but only if you are very careful.
- Why do I care? Random numbers drive statistical methods used in countless applications:
  - Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).

Monte Carlo Calculations:

Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$

- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
  long i;     long Ncirc = 0;      double pi, x, y;
  double r = 1.0;   // radius of circle. Side of squrare is 2*r
  seed(0,-r, r);  // The circle and square are centered at the origin
  #pragma omp parallel for private (x, y) reduction (+:Ncirc)
  for(i=0;i<num_trials; i++)
  {
    x = random();        y = random();
    if ( x*x + y*y) <= r*r)   Ncirc++;
   }

   pi = 4.0 * ((double)Ncirc/(double)num_trials);
   printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

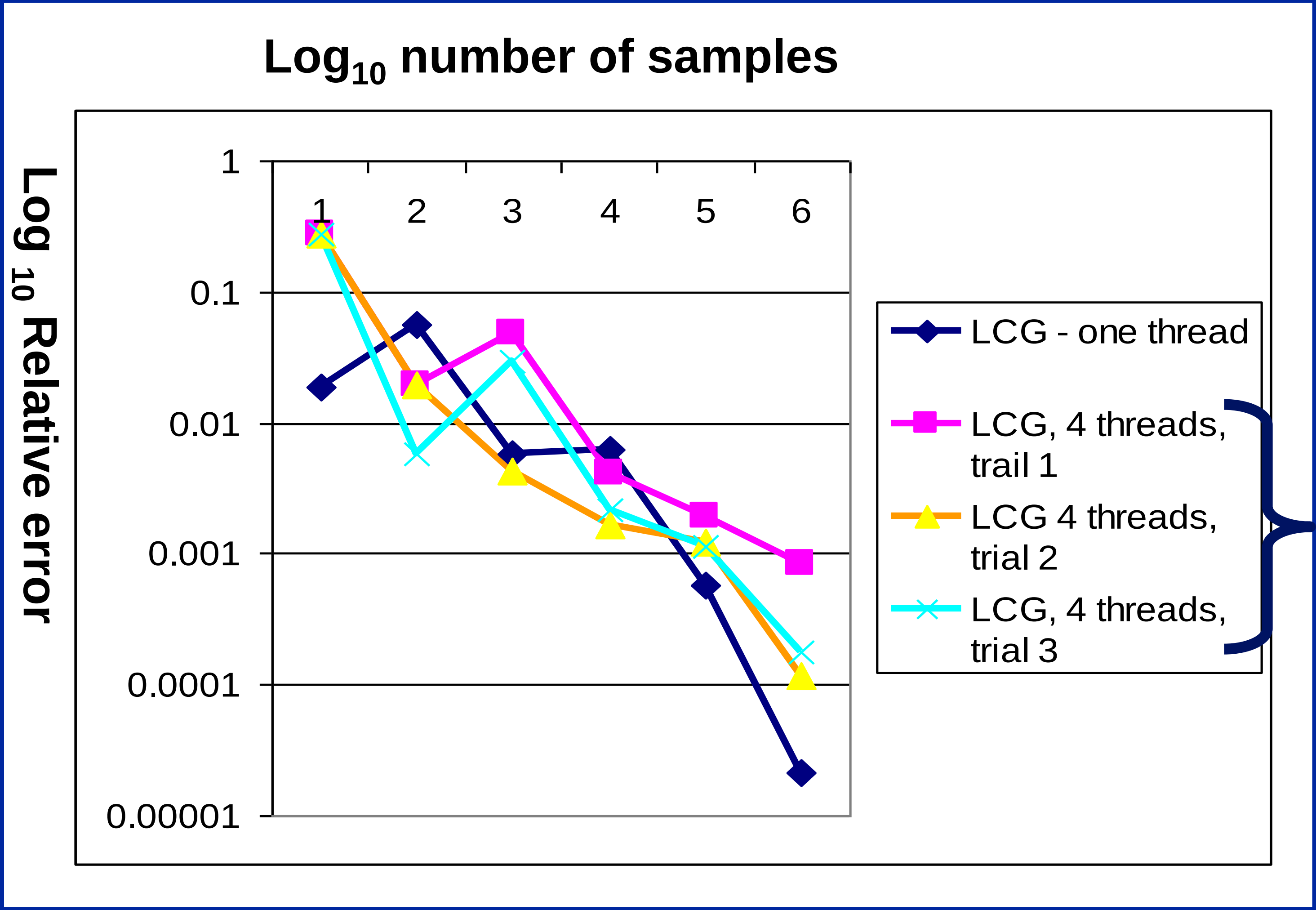
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND       = 150889;
static long PMOD         = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

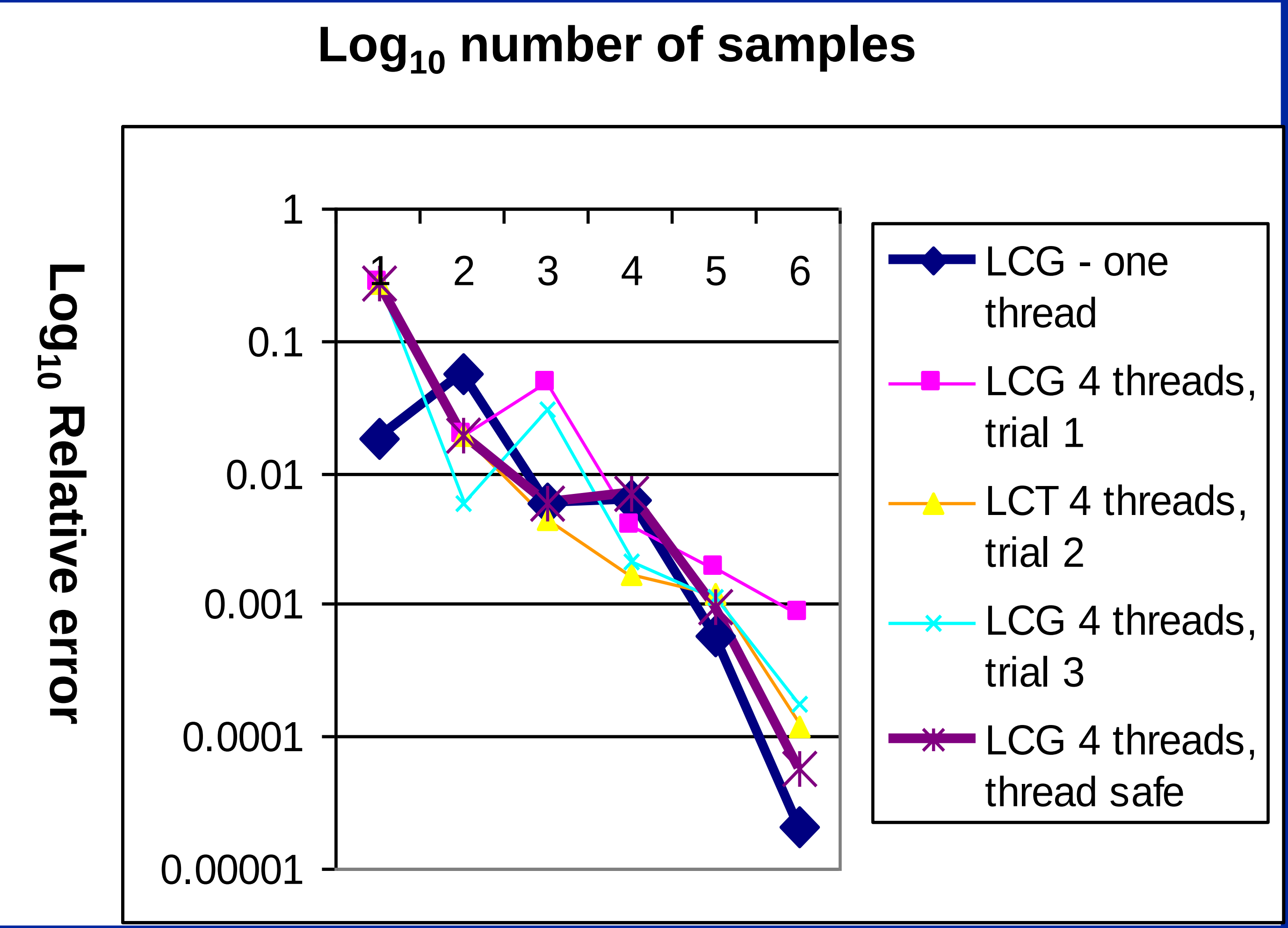
LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
   random_last = random_next;

  return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread safe random number generators



Thread safe version gives the same answer each time you run the program.

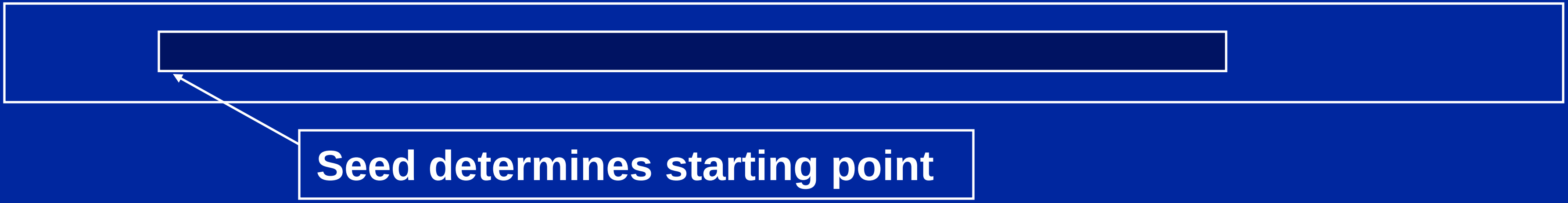
But for large number of samples, its quality is lower than the one thread result!

Why?

Pseudo Random Sequences

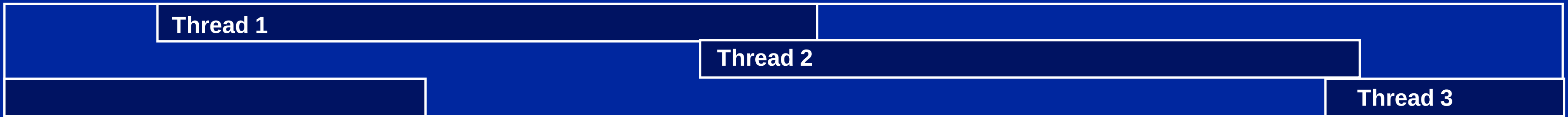
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range



Seed determines starting point

- Grab arbitrary seeds and you may generate overlapping sequences
  - E.g. three sequences … last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Replicate and Pray
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values "round robin" as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.
- Other than "replicate and pray", these are difficult to implement. Be smart … buy a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Nice for debugging, but not really needed scientifically.

Intel's Math kernel Library supports all of these methods.

MKL Random number generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                        BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Select type of RNG and set seed

Initialize a stream or pseudo random numbers

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

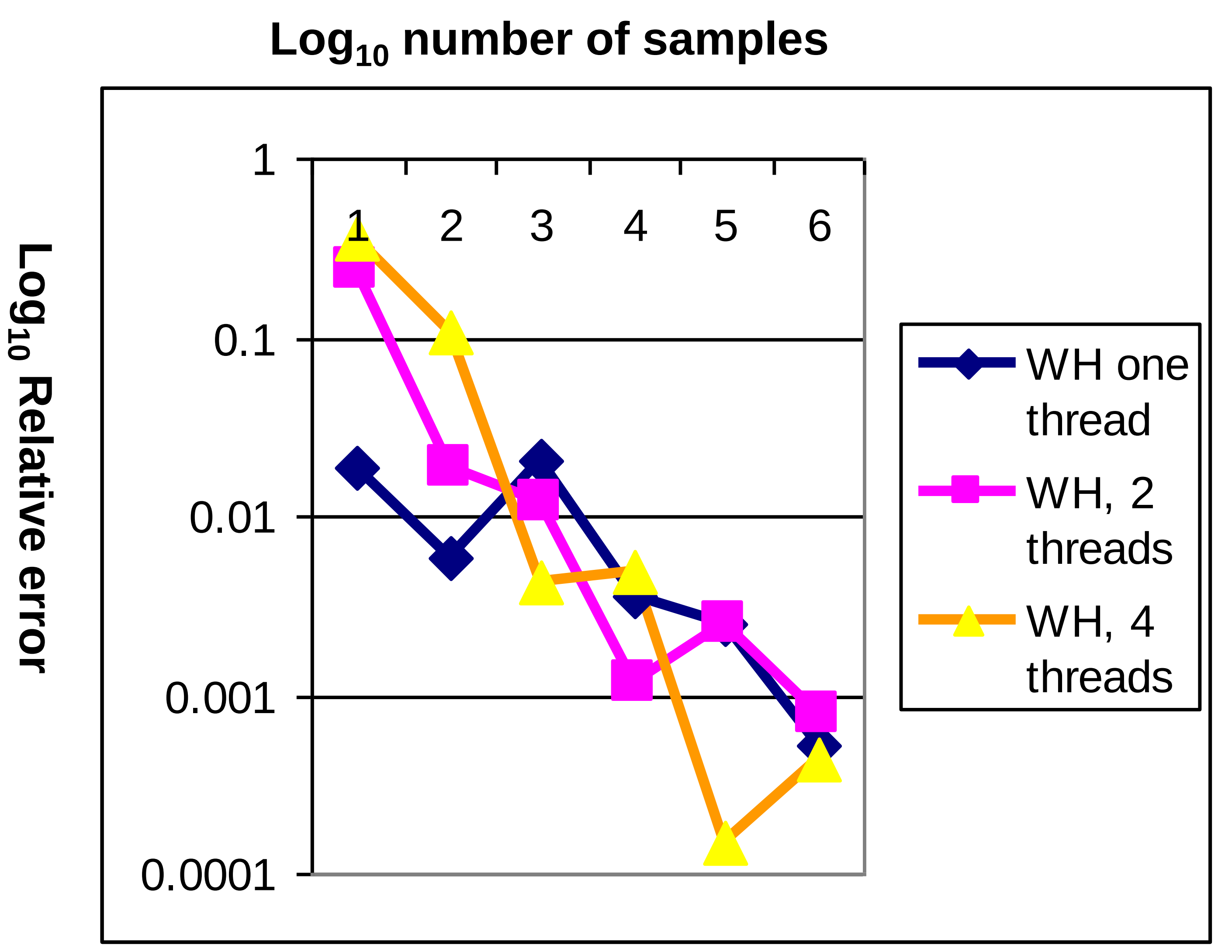
Delete the stream when you are done

Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                              …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the ith number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
     iseed = PMOD/MULTIPLIER;      // just pick a seed
     pseed[0] = iseed;
     mult_n = MULTIPLIER;
     for (i = 1; i < nthreads; ++i)
    {
         iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
         pseed[i] = iseed;
         mult_n = (mult_n * MULTIPLIER) % PMOD;
    }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 7: Molecular dynamics
  - Exercise 8: linked lists with tasks
  - Exercise 9: linked lists without tasks
  - Exercise 10: the producer-consumer pattern
- Thread Private Data
  - Exercise 11: Monte Carlo Pi and random numbers
- ➡ Fortran and OpenMP
- Compiler Notes

Fortran and OpenMP

- We were careful to design the OpenMP constructs so they cleanly map onto C, C++ and Fortran.
- There are a few syntactic differences that once understood, will allow you to move back and forth between languages.
- In the specification, language specific notes are included when each construct is defined.

OpenMP:
Some syntax details for Fortran programmers

- Most of the constructs in OpenMP are compiler directives.
    - For Fortran, the directives take one of the forms:

      C$OMP *construct [clause [clause]…]*

      !$OMP *construct [clause [clause]…]*

      *$OMP *construct [clause [clause]…]*
- The OpenMP include file and lib module

      use omp_lib

      Include omp_lib.h

OpenMP:
Structured blocks (Fortran)

- Most OpenMP constructs apply to structured blocks.
  - Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.
  - The only “branches” allowed are STOP statements in Fortran and exit() in C/C++.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
     print *,id
```

A structured block

```
C$OMP  PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id))goto 20
      go to 10
C$OMP END PARALLEL
     if(not_DONE) goto 30
20    print *, id
```

Not A structured block

OpenMP:

Structured Block Boundaries

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
      do I=1,N
            res(I)=bigComp(I)
        end do
C$OMP END PARALLEL DO
```

- The "construct/end construct" pairs is done anywhere a structured block appears in Fortran. Some examples:
  - DO … END DO
  - PARALLEL … END PARREL
  - CRICITAL … END CRITICAL
  - SECTION … END SECTION
  - SECTIONS … END SECTIONS
  - SINGLE … END SINGLE
  - MASTER … END MASTER

Runtime library routines

- The include file or module defines parameters
  - Integer parameter omp_locl_kind
  - Integer parameter omp_nest_lock_kind
  - Integer parameter omp_sched_kind
  - Integer parameter openmp_version
    - With value that matches C's _OPEMMP macro
- Fortran interfaces are similar to those used with C
  - Subroutine omp_set_num_threads (num_threads)
  - Integer function omp_get_num_threads()
  - Integer function omp_get_thread_num()\
  - Subroutine omp_init_lock(svar)
    - Integer(kind=omp_lock_kind) svar
  - Subroutine omp_destroy_lock(svar)
  - Subroutine omp_set_lock(svar)
  - Subroutine omp_unset_lock(svar)

Appendices

- Sources for Additional information
- Solutions to exercises
    - Exercise 1: hello world
    - Exercise 2: Simple SPMD Pi program
    - Exercise 3: SPMD Pi without false sharing
    - Exercise 4: Loop level Pi
    - Exercise 5: Matrix multiplication
    - Exercise 7: Molecular dynamics
    - Exercise 8: linked lists with tasks
    - Exercise 9: linked lists without tasks
    - Exercise 10: the producer-consumer pattern
- Thread Private Data
    - Exercise 11: Monte Carlo Pi and random numbers
- Fortran and OpenMP
- ➡ Compiler Notes

Compiler notes: Intel on Windows

- Intel compiler:
  - Launch SW dev environment … on my laptop I use:
    - start/intel software development tools/intel C++ compiler 11.0/C+ build environment for 32 bit apps
  - cd to the directory that holds your source code
  - Build software for program foo.c
    - icl /Qopenmp foo.c
  - Set number of threads environment variable
    - set OMP_NUM_THREADS=4
  - Run your program
    - foo.exe

To get rid of the pwd on the prompt, type

prompt = %

Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language … and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

Compiler notes: Other

- Linux and OS X with gcc:

```
> gcc -fopenmp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
```

for the Bash shell

- Linux and OS X with PGI:

```
> pgcc -mp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
```

OpenMP constructs

Where variable_list is a comma separated list of variables

- #pragma omp parallel
- #pragma omp for
- #pragma omp critical
- #pragma omp atomic
- #pragma omp barrier
- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)
- Tasks (remember … private data is made firstprivate by default)
  - pragma omp task
  - pragma omp taskwait
- #pragma threadprivate(variable_list)

Print the value of the macro

_OPENMP

And its value will be

yyyymm

For the year and month of the spec the implementation used

Put this on a line right after you define the variables in question