PARLab Parallel Boot Camp

Introduction to OpenMP

Tim Mattson

Microprocessor and Programming Research Lab

Intel Corp.

Disclaimer

- The views expressed in this presentation are my own and do not represent the views of the Intel Corporation (or its lawyers).

Outline

- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- OpenMP memory model ← beware the flush
- OpenMP 3.0
- The NUMA crisis and OpenMP

OpenMP* Overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OM

C$OM

C$O

ED

C

#p

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Makes writing multi-threaded applications in Fortran, C and C++ as easy as we can make it.
- 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

SMP – Symmetric multiprocessor

KAP – parallelization tool from KAI.

API – application programming interface

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

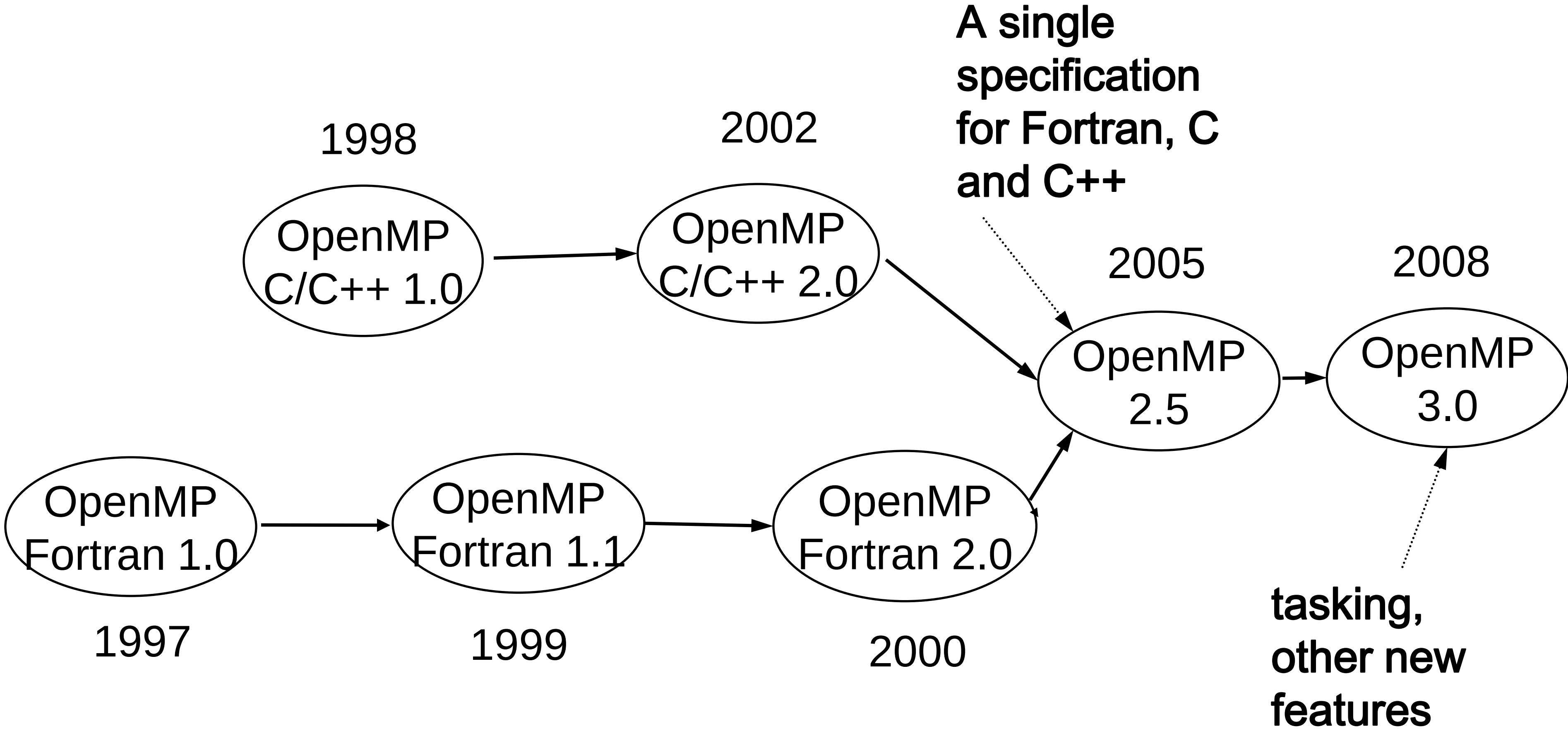
IBM

Intel

OpenMP

1997

OpenMP Release History



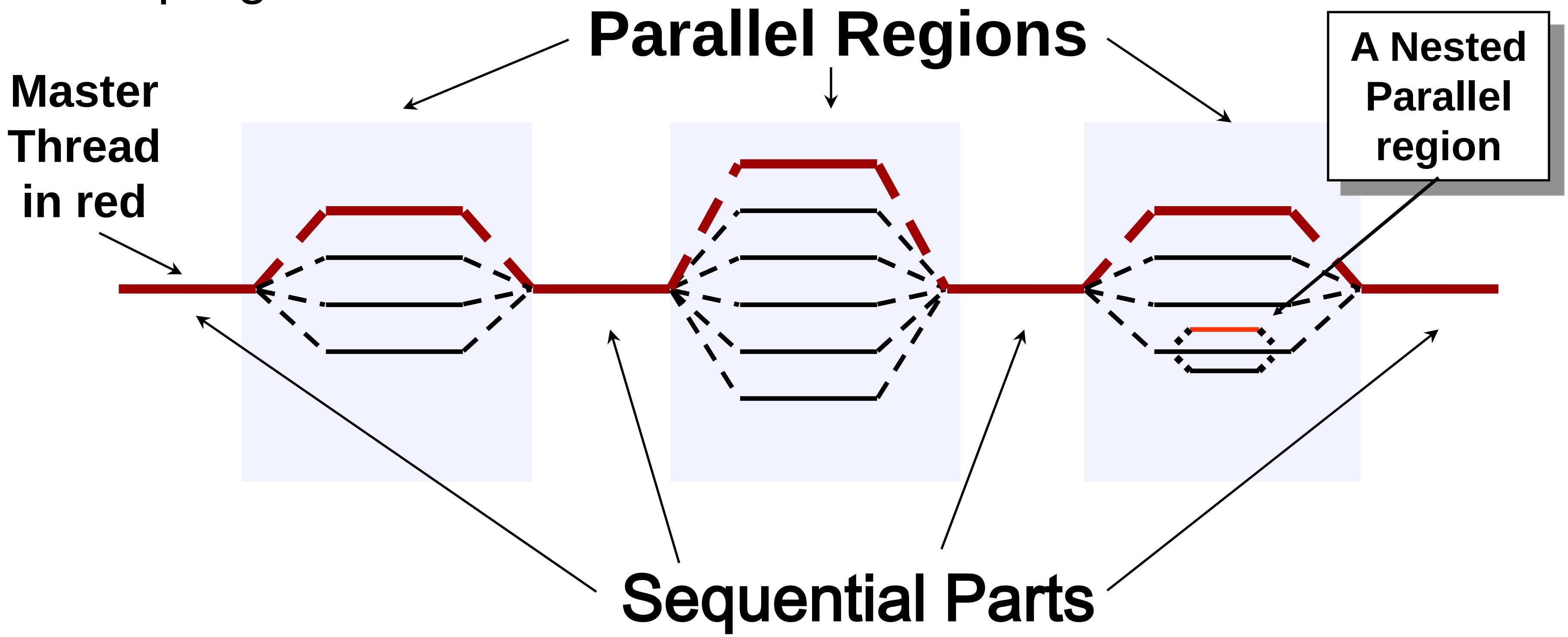
… and we are currently working on OpenMP 3.X

Outline

- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- OpenMP memory model ← beware the flush
- OpenMP 3.0
- The NUMA crisis and OpenMP

OpenMP Execution Model:

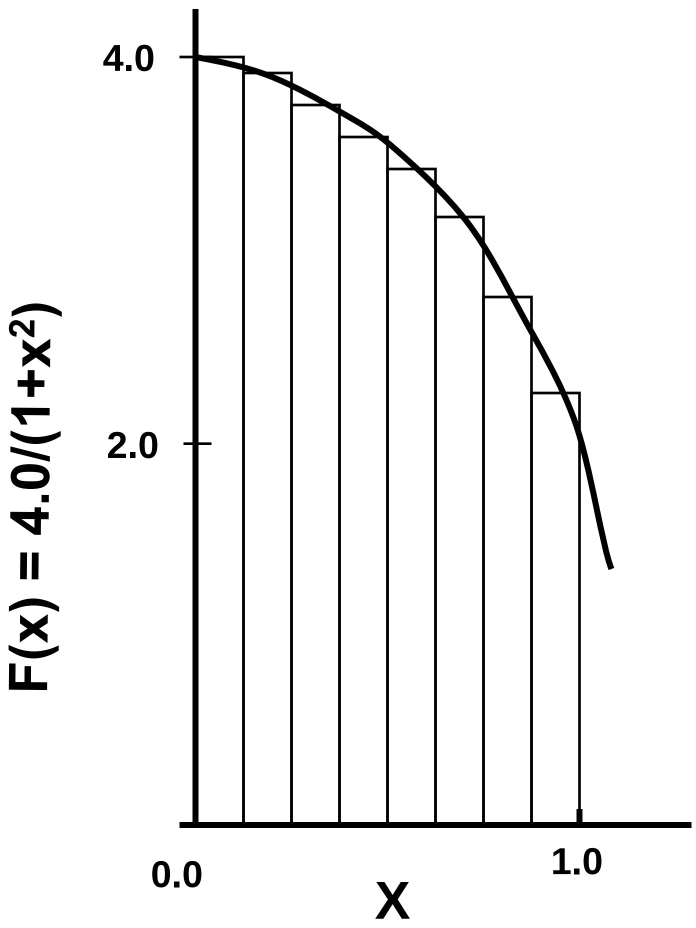
- **Master thread** spawns a **team of threads** as needed.
- Parallelism added incrementally until performance are met: i.e. the sequential program evolves into a parallel program.



The essence of OpenMP

- Create threads that execute in a shared address space:
  - The only way to create threads is with the "parallel construct"
  - Once created, all threads execute the code inside the construct.
- Split up the work between threads by one of two means:
  - SPMD (Single program Multiple Data) … all threads execute the same code and you use the thread ID to assign work to a thread.
  - Workshare constructs split up loops and tasks between threads.
- Manage data environment to avoid data access conflicts
  - Synchronization so correct results are produced regardless of how threads are scheduled.
  - Carefully manage which data can be private (local to each thread) and shared.

Example Problem: 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.

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{           int i;   double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;
            x = 0.5 * step;
            for (i=0;i<= num_steps; i++){
                     x+=step;
                     sum += 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

How to write a parallel program

Stages of parallel programming
Identify the concurrent tasks in a problem.
Organize the problem and structure source code to expose the concurrent tasks.
Express the concurrency and its safe execution in the source code .
Execute the concurrency on parallel hardware, evaluate performance

PI Program: identify Concurrency

```
static long num_steps = 100000;
double step;
void main ()
{         int i;   double x, pi, sum = 0.0;

          step = 1.0/(double) num_steps;
          x = 0.5 * step;
          for (i=0;i<= num_steps; i++){
                    x+=step;
                    sum += 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

Loop iterations can in principle be executed concurrently

PI Program: Expose Concurrency, part 1

```
static long num_steps = 100000;
double step;
void main ()
{        double pi, sum = 0.0;
         step = 1.0/(double) num_steps;

         int i;   double x;
         for (i=0;i<= num_steps; i++){
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
         }
         pi = step * sum;
}
```

Isolate data that must be shared from data local to a task

Redefine x to remove loop carried dependence

This is called a reduction … results from each iteration accumulated into a single global.

PI Program: Expose Concurrency, part 2

Deal with the reduction

```
static long num_steps = 100000;
#define NUM 4   //expected max thread count
double step;
void main ()
{       double pi, sum[NUM] = {0.0};
        step = 1.0/(double) num_steps;

        int i, ID=0;    double x;
        for (i=0;i<= num_steps; i++){
                x = (i+0.5)*step;
                sum[ID] += 4.0/(1.0+x*x);
        }
        for(int i=0, pi=0.0;i<NUM;i++)
                pi += step * sum[i];
}
```

Common Trick: promote scalar "sum" to an array indexed by the number of threads to create thread local copies of shared data.

PI Program: Express Concurrency using OpenMP

```
#include <omp.h>
static long num_steps = 100000;
#define NUM 4
double step;
void main ()
{            double pi, sum[NUM] = {0.0};
             step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM)
{
          int i, ID;             double x;
          ID = omp_get_thread_num();
           for (i=ID;i<= num_steps; i+=NUM){
                 x = (i+0.5)*step;
                 sum[ID] += 4.0/(1.0+x*x);
            }
}
             for(int i=0, pi=0.0;i<NUM;i++)
                    pi += step * sum[i];
}
```

automatic variables defined outside a parallel region are shared between threads

Create NUM threads

Each thread executes code in the parallel block

Simple mod to loop to deal out iterations to threads

variables defined inside a thread are private to that thread

PI Program: Fixing the NUM threads bug

```
#include <omp.h>
static long num_steps = 100000;
#define NUM 4
double step;
void main ()
{          double pi, sum[NUM] = {0.0};
           step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM)
{          int nthreads = omp_get_num_threads();
         int i, ID;             double x;
         ID = omp_get_thread_num();
          for (i=ID;i<= num_steps; i+=nthreads){
                   x = (i+0.5)*step;
                   sum[ID] += 4.0/(1.0+x*x);
           }
}
           for(int i=0, pi=0.0;i<NUM;i++)
                  pi += step * sum[i];
}
```

NUM is a requested number of threads, but an OS can choose to give you fewer.

Hence, you need to add a bit of code to get the actual number of threads

Incremental Parallelism

- Software development with incremental Parallelism:
  - Behavior preserving transformations to expose concurrency.
  - Express concurrency incrementally by adding OpenMP directives… in a large program I can do this loop by loop to evolve my original program into a parallel OpenMP program.
  - Build and time program, optimize as needed with behavior preserving transformations until you reach the desired performance.

PI Program: Execute Concurrency

```
#include <omp.h>
static long num_steps = 100000;
#define NUM 4
double step;
void main ()
{           double pi, sum[NUM] = {0.0};
            step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM)
{          int nthreads = omp_get_num_threads();
          int i, ID;              double x;
          ID = omp_get_thread_num();
           for (i=ID;i<= num_steps; i+=nthreads){
                  x = (i+0.5)*step;
                  sum[ID] += 4.0/(1.0+x*x);
           }
}
            for(int i=0, pi=0.0;i<NUM;i++)
                 pi += step * sum[i];
}
```

Build this program and execute on parallel hardware.

The performance can suffer on some systems due to false sharing of sum[ID] … i.e. independent elements of the sum array share a cache line and hence every update requires a cache line transfer between threads.

PI Program: Safe update of shared data

```
#include <omp.h>
static long num_steps = 100000;
#define NUM 4
double step;
void main ()
{           double pi, sum=0.0;
            step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM)
{         int i, ID;              double x, psum= 0.0;
          ID = omp_get_thread_num();
          ID = omp_get_thread_num();
          for (i=ID;i<= num_steps; i+=nthreads){
                  x = (i+0.5)*step;
                  psum += 4.0/(1.0+x*x);
            }
            #pragma omp critical
                  sum += psum
}
    pi = step * sum;
}
```

Replace array for sum with a local/private version of sum (psum) … no more false sharing

Use a critical section so only one thread at a time can update sum, i.e. you can safely combine psum values

Pi program: making loop-splitting and reductions even easier

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
#pragma omp  parallel   for   private(i, 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;
}
```

Private clause creates data local to a thread

Reduction used to manage dependencies

Outline

- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- OpenMP memory model ← beware the flush
- OpenMP 3.0
- The NUMA crisis and OpenMP

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

Putting the master thread to work

- 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();
}
```

Runtime Library routines and ICVs

- 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

Internal Control Variables (ICVs) … define state of runtime system to a thread. Consistent pattern: set with "omp_set" or an environment variable, read with "omp_get"

Optimizing loop parallel programs

Short range force computation for a particle system using the cut-off method

```
#include <omp.h>
#pragma omp parallel
{
// define neighborhood as the num_neighbors particles
// within "cutoff" of each particle "i".
#pragma omp for
        for( int i = 0; i < n; i++ )
        {
            Fx[i]=0.0;   Fy[i]=0.0;
            for (int j = 0; j < num_neigh[i]; j++)
                neigh_ind = neigh[i][j];
                Fx[i] += forceX(i, neigh_ind);
                FY[i] += forceY(i, neigh_ind);
             }
        }
}
```

Particles may be unevenly distributed … i.e. different particles have different numbers of neighbors.

Evenly spreading out loop iterations may fail to balance the load among threads

We need a way to tell the compiler how to best distribute the load.

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 … for OpenMP 3.0).

Optimizing loop parallel programs

Short range force computation for a particle system using the cut-off method

```
#include <omp.h>
#pragma omp parallel
{
// define neighborhood as the num_neigh particles
// within "cutoff" of each particle "i".
#pragma omp for schedule(dynamic, 10)
        for( int i = 0; i < n; i++ )
        {
             Fx[i]=0.0;   Fy[i]=0.0;
             for (int j = 0; j < num_neigh[i]; j++)
                  neigh_ind = neigh[i][j];
                  Fx[i] += forceX(i, neigh_ind);
                  FY[i] += forceY(i, neigh_ind);
              }
        }
}
```

Divide range of n into chunks of size 10.

Each thread computes a chunk then goes back to get its next chunk of 10 iterations.

Dynamically balances the load between threads.

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

Least work at runtime : scheduling done at compile-time

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

Summary of OpenMP's key constructs

- The only way to create threads is with the parallel construct:

  #pragma omp parallel

- All thread execute the instructions in a parallel construct.
- Split work between threads by:
  - SPMD: use thread ID to control execution
  - Worksharing constructs to split loops (simple loops only)

    #pragma omp for

- Combined parallel/workshare as a shorthand

  #pragma omp parallel for

- High level synchronization is safest

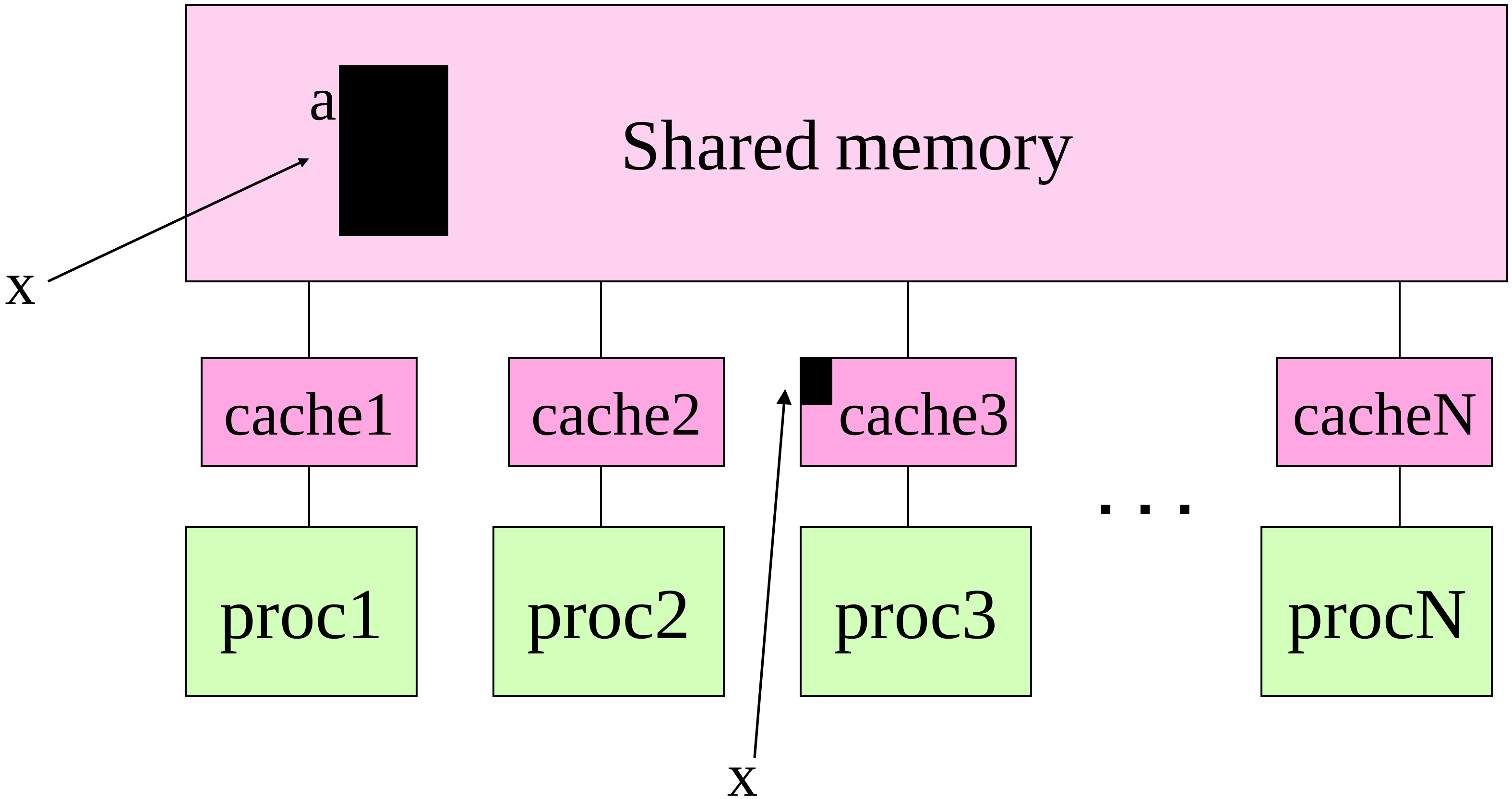
  #pragma critical

  #pragma barrier

Outline

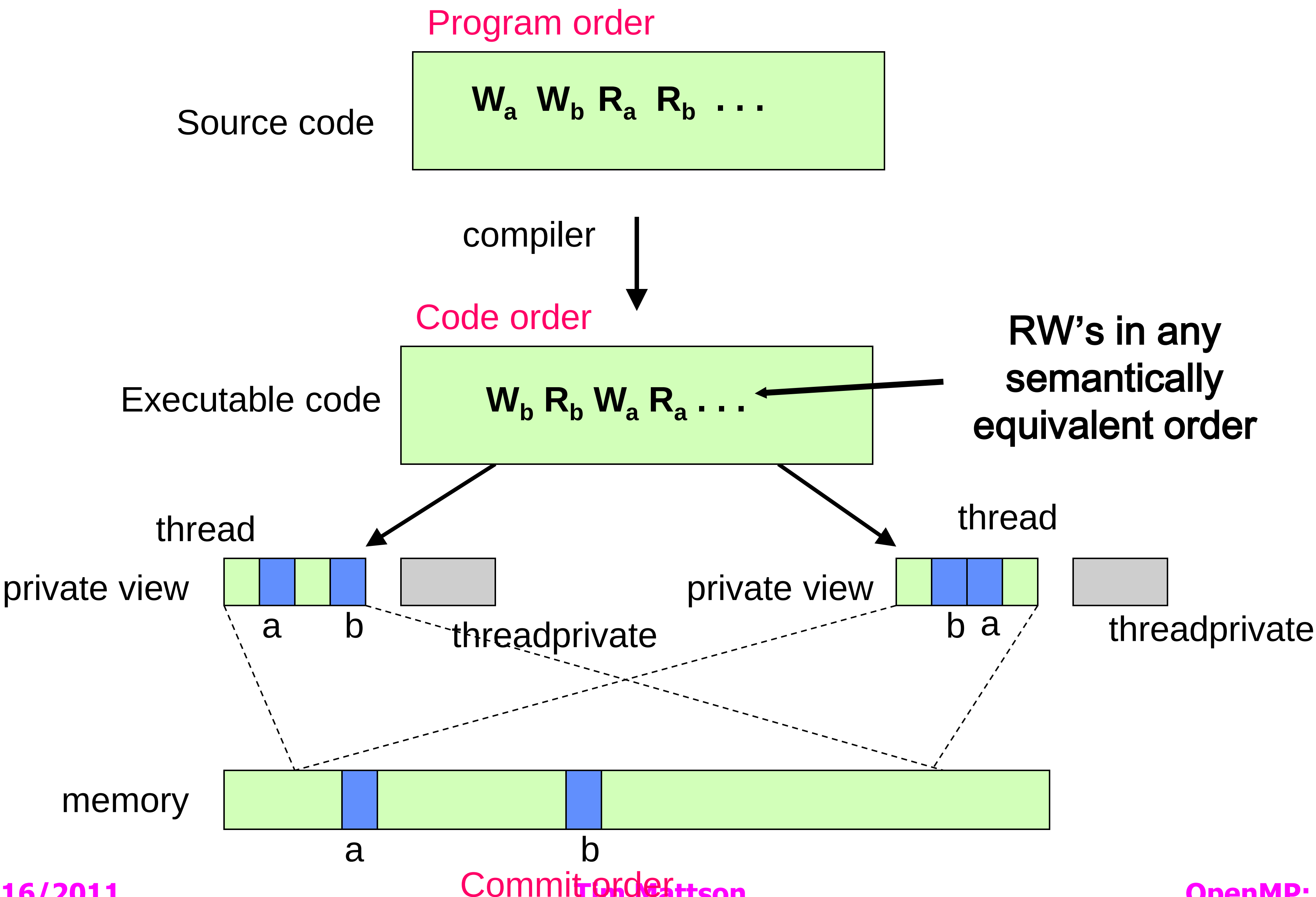
- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- ⇨ OpenMP memory model ← beware the flush
- OpenMP 3.0
- The NUMA crisis and OpenMP

Shared Memory Architecture



There is a single address space (shared memory) but due to the caches, a processor may hold a value for "x" that is different from the one in shared memory.

OpenMP Memory Model: Basic Terms



OpenMP: Forcing a consistent view of memory

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory for all thread-visible variables (the "flush set").

  #pragma omp flush

- For the variables in the flush set:
  - » All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - » All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - » Variables in registers or write buffers must be updated in memory.
- Compilers reorder instructions to better exploit the functional units and keep the machine busy
  - A compiler CANNOT do the following:
    - » Reorder read/writes of variables in a flush set relative to a flush.
    - » Reorder flush constructs when flush sets overlap
  - A compiler CAN do the following:
    - » Reorder instructions NOT involving variables in the flush set relative to the flush.
    - » Reorder flush constructs that don't have overlapping flush sets.

> OpenMP applies flushes automatically at the "right" places (barriers, end of workshare constructs, etc). You usually don't need to worry about flushes explicitly.

Pair-wise synchronization 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

This use of flush exposes the details of OpenMP's relaxed memory model … a risky practice for experienced shared memory programmers only.

Producer/consumer and flush

```
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
     }
     #pragma omp section
     {
       #pragma omp flush
       while (flag != 1){
           #pragma omp flush
       }
       #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

The rest of OpenMP 2.5

- Create threads
  - parallel
- Share work among a set of threads
  - for
  - single
  - Sections
- Synchronize to remove race conditions
  - Critical
  - Atomic
  - Barrier
  - locks
  - flush
- Manage data environment
  - Private
  - shared
  - threadprivate
  - firstprivate
  - Lastprivate
  - Reduction
- Interact with runtime
  - change numbers of threads
  - Discover thread properties
  - modify environment

Outline

- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- OpenMP memory model ← beware the flush
- ⇒ OpenMP 3.0
- The NUMA crisis and OpenMP

OpenMP 3.0: completed May 2008

- Task expression
  - Task Queues
  - Loop collapse
- Resource management
  - Stack size control
  - Thread wait policy
  - Improved Nesting support
  - Multiple Internal control variables
- Scheduling
  - STATIC schedule
  - Schedule kinds
- Clean up:
  - Constructors/destructors
  - Memory model
  - Unsigned int in a for-loop
  - Storage reuse

Of all these changes, the most significant by far is the addition of task queues

Tasks beyond loops

- OpenMP is fundamentally based on tasks … i.e. the constructs in OpenMP define sets of tasks executed by teams of threads.
- OpenMP 2.5 provides only a few ways to define tasks:
  - The code redundantly executed inside parallel regions (SPMD programs).
  - Iterations from "simple loops" split between threads.
  - Section construct
  - Single construct (with a no wait if you want concurrency)
- OpenMP 3.0 adds explicit tasks with deferred execution (task queues) … thereby dramatically expanding the scope of algorithms that can be handled by OpenMP

Explicit Tasks in OpenMP 3.0

- OpenMP 2.5 can not handle the very common case of a pointer chasing loop:

```
nodeptr list, p;
for (p=list; p!=NULL; p=p->next)
        process(p->data);
```

- OpenMP 3.0 covers this case with explicit tasks:

```
nodeptr list, p;
#pragma omp single
{
 for (p=list; p!=NULL; p=p->next)
     #pragma omp task firstprivate(p)
          process(p->data);
}
```

One thread goes through the loop and creates a set of tasks

Captures value of p for each task

tasks go on a queue to be executed by an available thread

Task Expression: The new OpenMP 3.0 Task directive

- Explicit tasks are created with the task construct

  #pragma omp task [<clause>] …

  <structured block>

- A task is executed by a thread, called the **task-thread**, which may be any thread in the encountering thread's team.
- A **task barrier** … is a point where preceding tasks must complete before threads continue
- To prevent deadlock, we define "**thread switching points**" where a thread may pause and execute other tasks.
  - This happens most commonly at barriers or other natural places where a break in the action makes sense.

Tasks with synchronization

```
struct node {
    struct node *left;
    struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
    #pragma omp task // p is firstprivate by default
        postorder_traverse(p->left);
    if (p->right)
    #pragma omp task // p is firstprivate by default
        postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

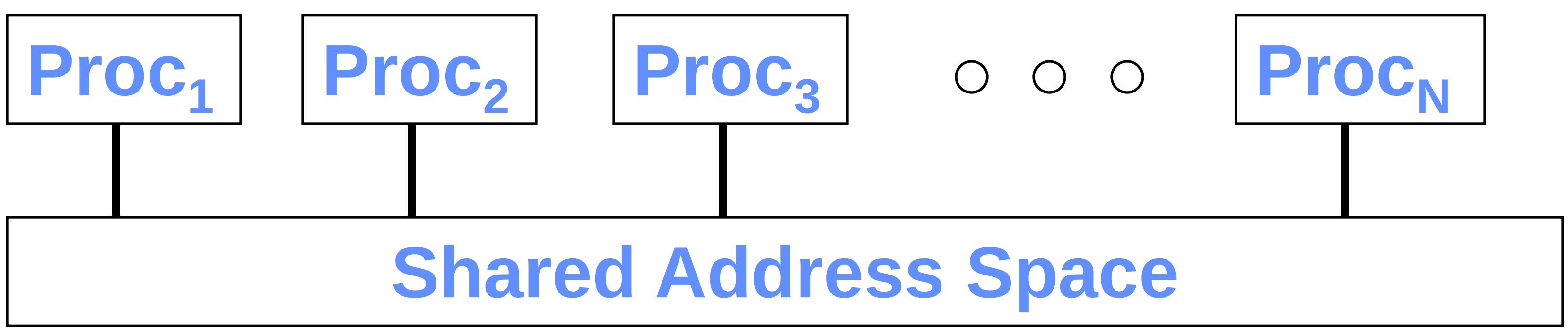
Do not proceed until prior tasks in scope have completed

Outline

- OpenMP: History and high level overview
- Software development with OpenMP
- OpenMP 2.5: other essential constructs
  - Synchronization
  - Runtime library
  - Loop scheduling
- OpenMP memory model ← beware the flush
- OpenMP 3.0
- ⇨ The NUMA crisis and OpenMP

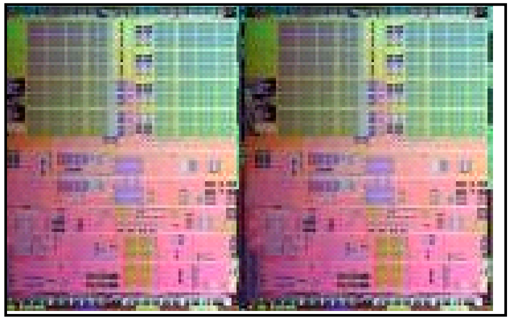
OpenMP Computational model

- OpenMP was created with a particular abstract machine or ***computational model*** in mind:
  - » Multiple processing elements.
  - » A shared address space with "equal-time" access for each processor.
  - » Multiple light weight processes (threads) managed outside of OpenMP (the OS or some other "third party").



How realistic is this model?

- Some of the old supercomputer mainframes followed this model,

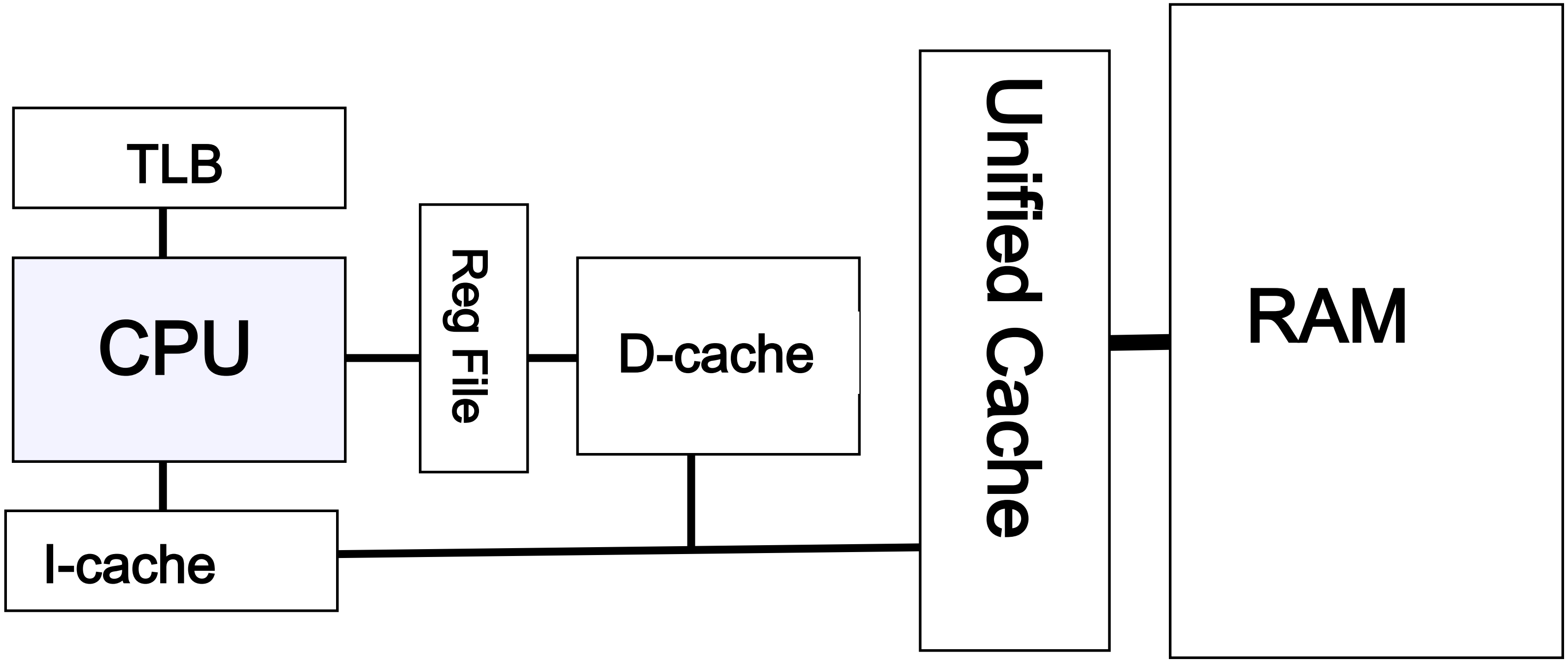


A CPU with lots of cache …

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

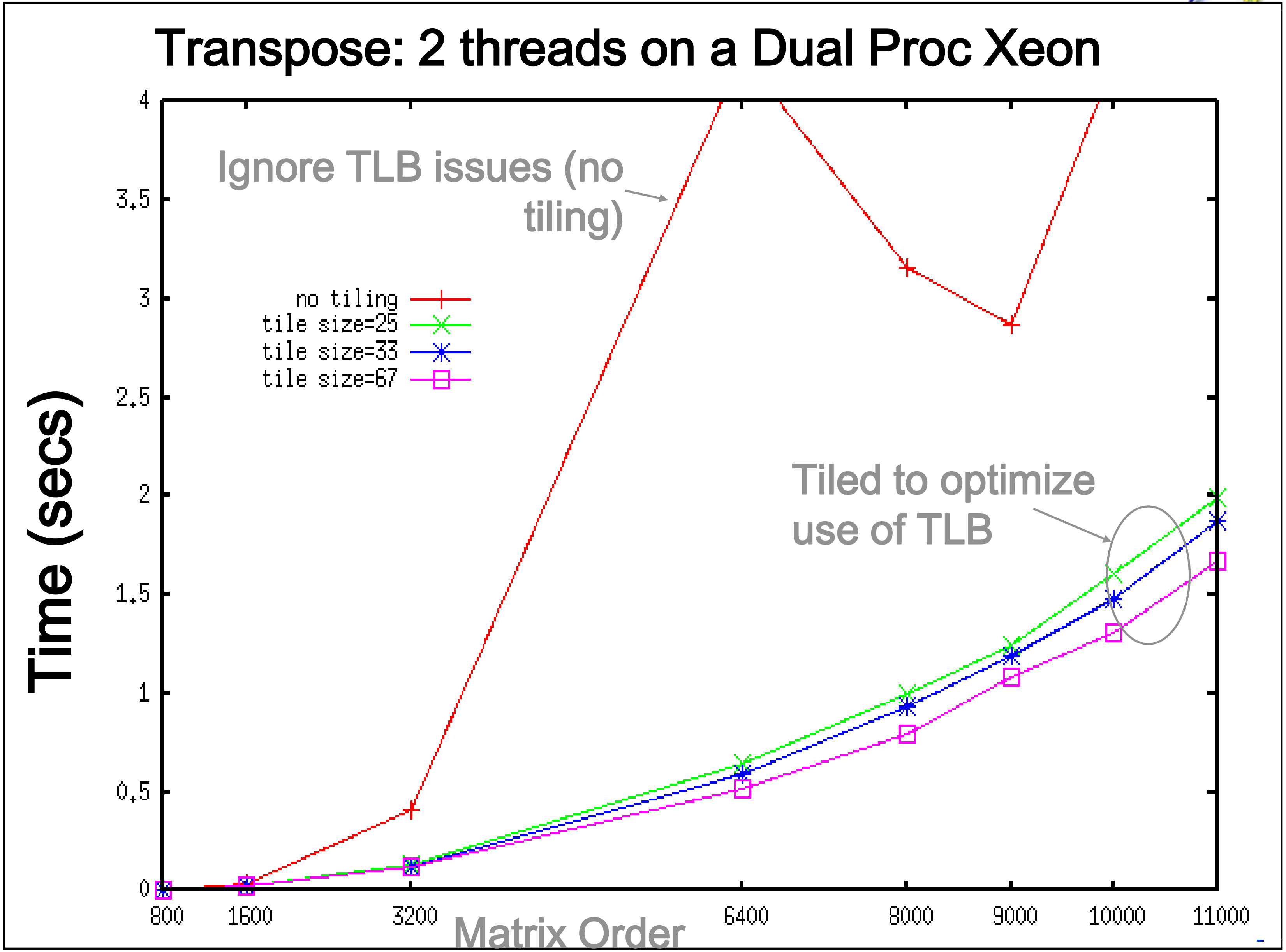
Memory Hierarchies

- A typical microprocessor memory hierarchy



- Instruction cache and data cache pull data from a unified cache that maps onto RAM.
- TLB implements virtual memory and brings in pages to support large memory foot prints.

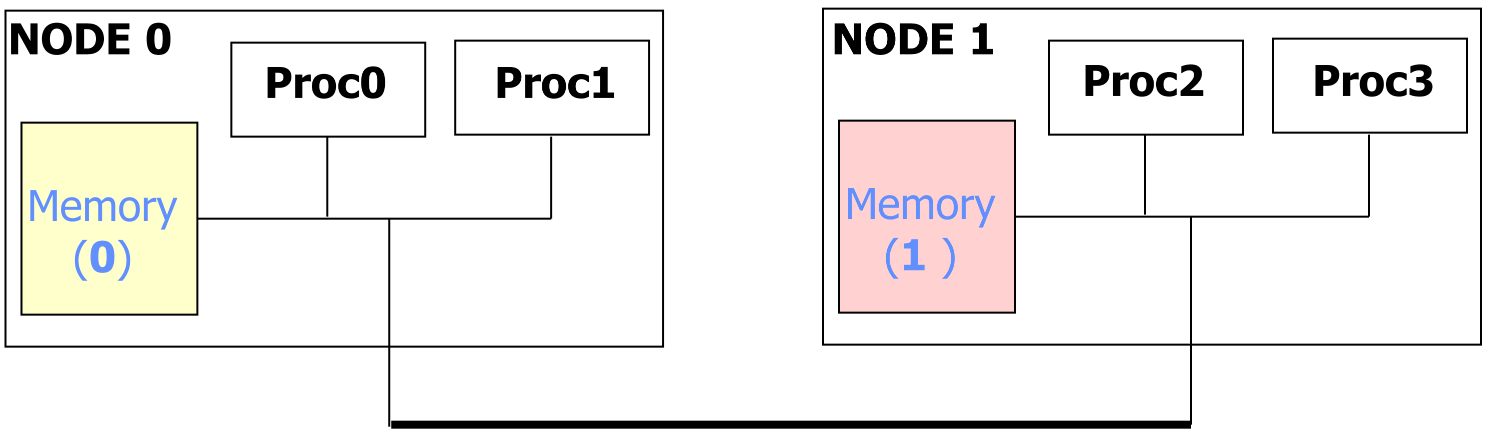
Do you need to worry about the TLB?



Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel

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.

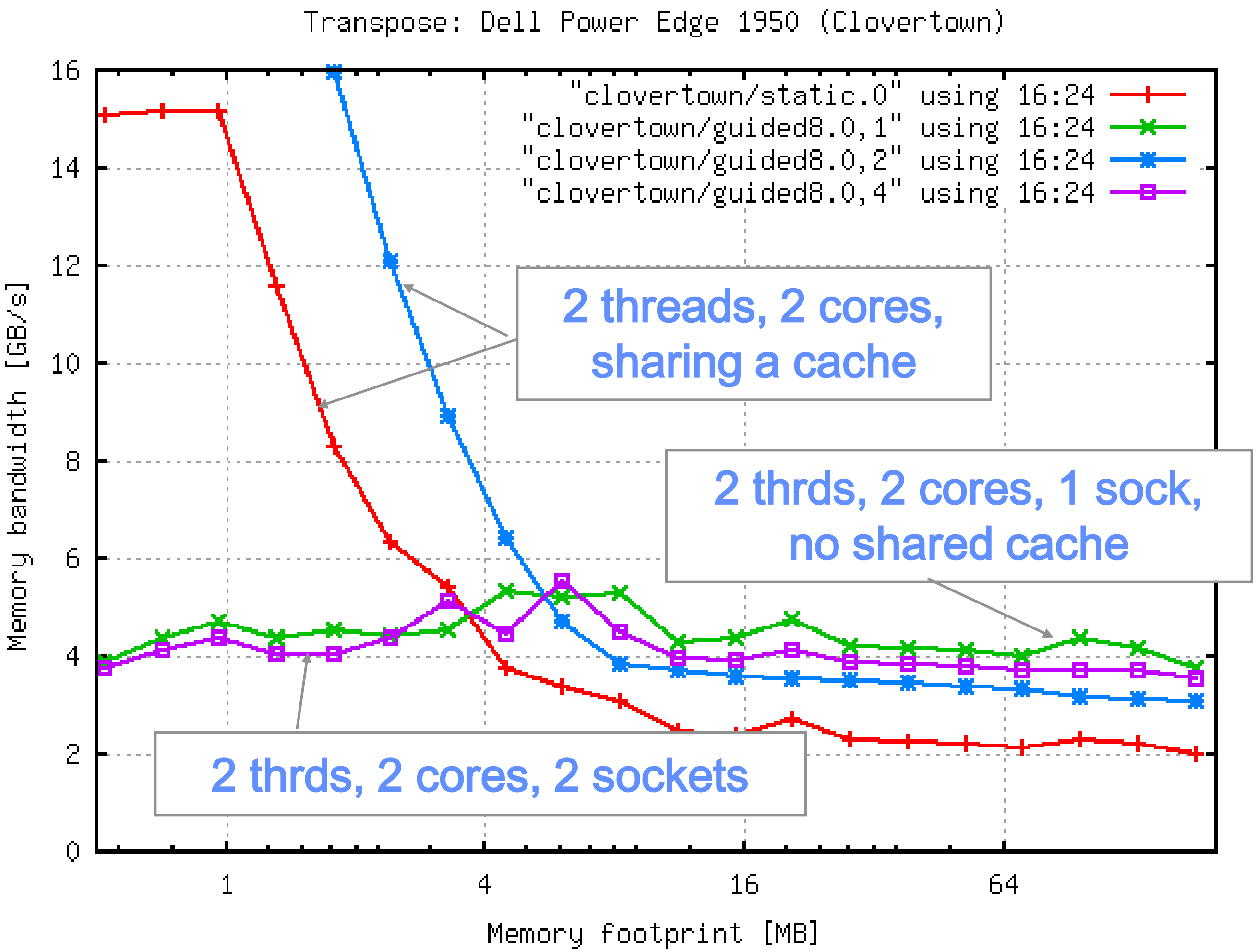
Consider a cluster, and it gets much worse!

Itanium 2 latencies

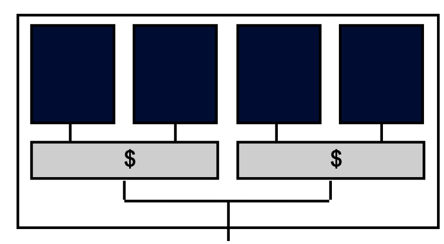
latency to L1:	1 - 2 cycles
latency to L2:	5 - 7 cycles
latency to L3:	12 - 21 cycles
latency to memory:	180 – 225 cycles
Gigabit Ethernet - latency to remote node:	~45000 cycles (30uS)
Infiniband* - latency to remote node:	~7500 cycles (5uS)

Source: Intel Cluster OpenMP workshop

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



A single quad-core chip is a NUMA machine!



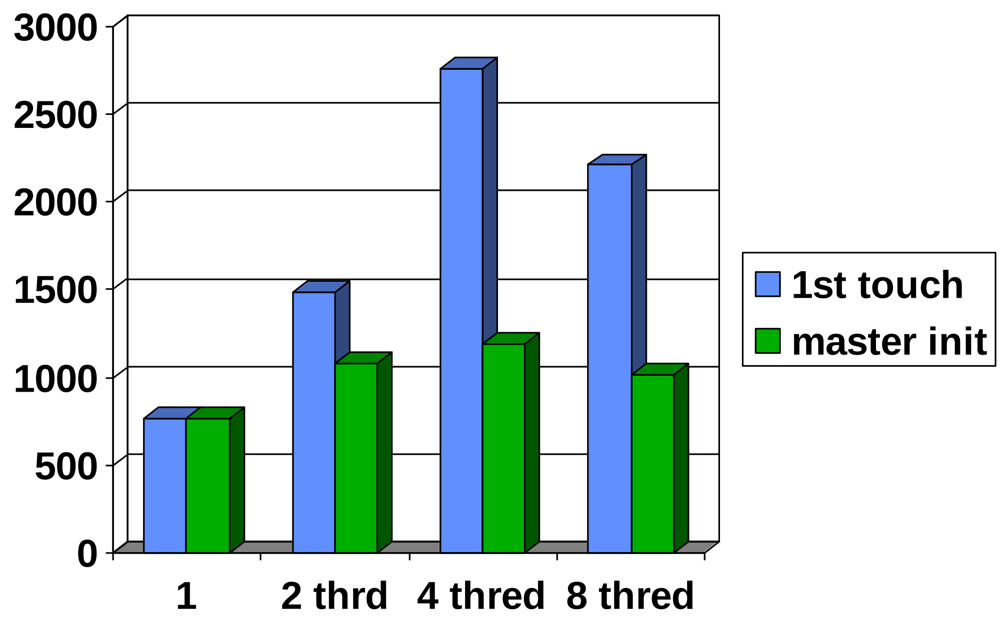
Xeon® 5300 Processor block diagram

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

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

Conclusion

- OpenMP is one of the most commonly used APIs for programming shared memory computers:
  - My friends in the Intel Software group tell me countless ISVs are shipping applications using OpenMP.
- Incremental parallelism with testing at every step is not a luxury … it is a requirement.
- OpenMP makes things about as easy as we can for application programmers.
- OpenMP is useful in hybrid models to help expose additional levels of concurrency
- BUT, OpenMP is in trouble when it comes to NUMA machines … and practically all machines are NUMA so this is a big deal.