Outline

- Processors and registers
- Memory hierarchies
  - Temporal and spatial locality
  - Cache timing
  - Use of microbenchmarks to characterize performance
- Case study: Matrix multiplication
- Optimizations in Practice

Outline

- Optimizations in Practice
- Parallel Programming with Shared Memory

Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions (flags)
- Hard to do by hand

Note on Matrix Storage

- A matrix is a 2-D array of elements, but memory addresses are “1-D”
- Conventions for matrix layout
  - by column, or “column major” (Fortran default); A(i,j) at A+i+j*n
  - by row, or “row major” (C default) A(i,j) at A+i*n+j
  - recursive (later)
- Intense interaction with cache line size

Tips on Tuning

“We should forget bout small efficiences, say about 97% of the time: premature optimization is the root of all evil.”– C.A.R. Hoare (quoted by Donald Knuth)

- Tradeoff: speed vs readability, debuggability, maintainability...
- Only optimize when needful
- Go for low-hanging fruit first: data layouts, libraries, compiler flags
- Concentrate on the bottleneck
- Concentrate on inner loops
- Get correctness (and a test framework) first

Tip #1: Tools & Libraries

- We have gcc. The Intel compilers are better.
- Fortran compilers often do better than C compilers (less pointer aliasing)
- Intel VTune, cachegrind, and Shark can provide useful profiling information (including information about cache misses)
- Libraries build on someone else's hard work

Tip #2: Compiler flags

- -O3: Aggressive optimization
- -march=core2: Tune for specific architecture
- -ftree-vectorize: Automatic use of SSE (supposedly)
- -funroll-loops: Loop unrolling
- -ffast-math: Unsafe floating point optimizations

Tip #3: Memory layout

- Arrange data for unit stride access
- Arrange algorithm for unit stride access
- Tile for multiple levels of cache
- Tile for registers (loop unrolling + “register” variables)

Tip #4: Use small data structures

- Smaller data types are faster
- Bit arrays vs int arrays for flags?
- Minimize indexing data — store data in blocks
- Some advantages to mixed precision calculation (float for large data structure, double for local calculation)
- Sometimes recomputing is faster than saving

Tip #5: Inline judiciously

- Function call overhead is often minor...
- ... but structure matters to optimizer!
- C++ has inline keyword to indicate inlined functions

Loop Unrolling

- Expose instruction-level parallelism and reduce control overhead

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
   signal += 3;
   s0 = signal[0];
   res[0] = f0*s1 + f1*s2 + f2*s0;
   s1 = signal[1];
   res[1] = f0*s2 + f1*s0 + f2*s1;
   s2 = signal[2];
   res[2] = f0*s0 + f1*s1 + f2*s2;
   res += 3;
} while( … );
```

Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies
- With some compilers, you can declare a and b unaliased.
  - Done via “restrict pointers,” compiler flag, or pragma
  - In Fortran, can use function calls (arguments assumed unaliased, maybe).

```
a[i] = b[i] + c;
a[i+1] = b[i+1] * d; // a[i] and b[i+1] don't have a data dependency

float f1 = b[i];
float f2 = b[i+1];

a[i] = f1 + c;
a[i+1] = f2 * d;
```

Exploit Multiple Registers

- Reduce memory bandwidth by pre-loading into local variables

```
while( … ) {
   *res++ = filter[0]*signal[0]
            + filter[1]*signal[1]
            + filter[2]*signal[2];
   signal++;
}

float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( … ) {
   *res++ = f0*signal[0]
           + f1*signal[1]
           + f2*signal[2];
   signal++;
}
```

Expose Independent Operations

- Hide instruction latency
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

```
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```

A Brief History of Parallel Languages

- When vector machines were king
  - Parallel “languages” were loop annotations
  - Performance was fragile, but good user support
- When SIMD machines were king
  - Data parallel languages popular and successful (*Lisp, C*, …)
  - Irregular data (sparse matix-vector multiply OK), but irregular computation (divide and conquer, adaptive meshes, etc.) less clear
- When shared memory multiprocessors (SMPs) were king
  - Shared memory models, e.g., Posix Threads, OpenMP, are popular
- When clusters took over
  - Message Passing (MPI) became dominant
- With the addition of accelerators
  - OpenACC, CUDA were added
- In Cloud Computing
  - Sawzall, Hadoop, SPARK, ...

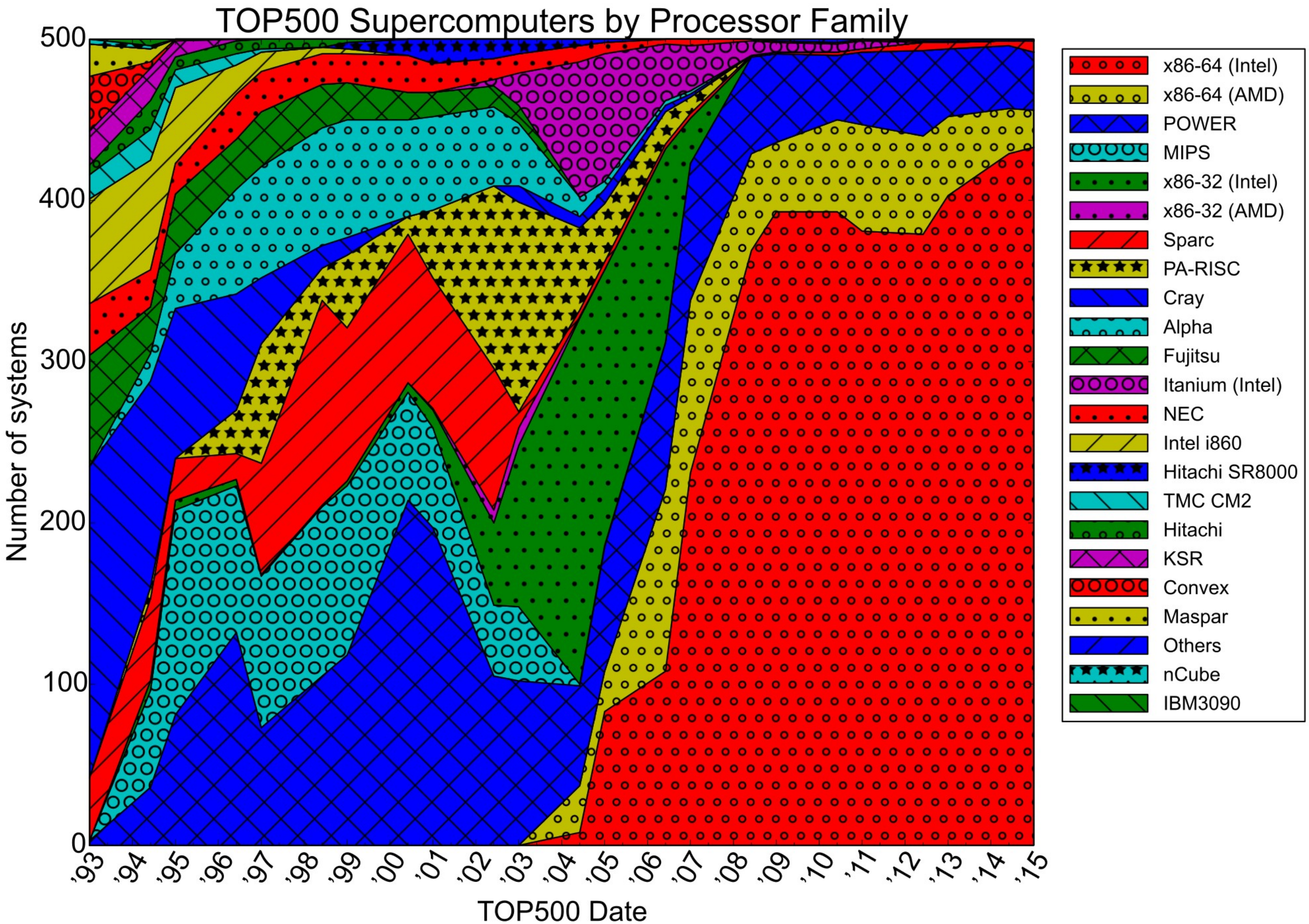
Outline (via OpenMP)

- Shared memory parallelism with threads
- What and why OpenMP?
- Parallel programming with OpenMP
  - Introduction to OpenMP
  - Creating parallelism
  - Parallel Loops
  - Synchronizing
  - Data sharing
- Beneath the hood
  - Shared memory hardware
- Summary

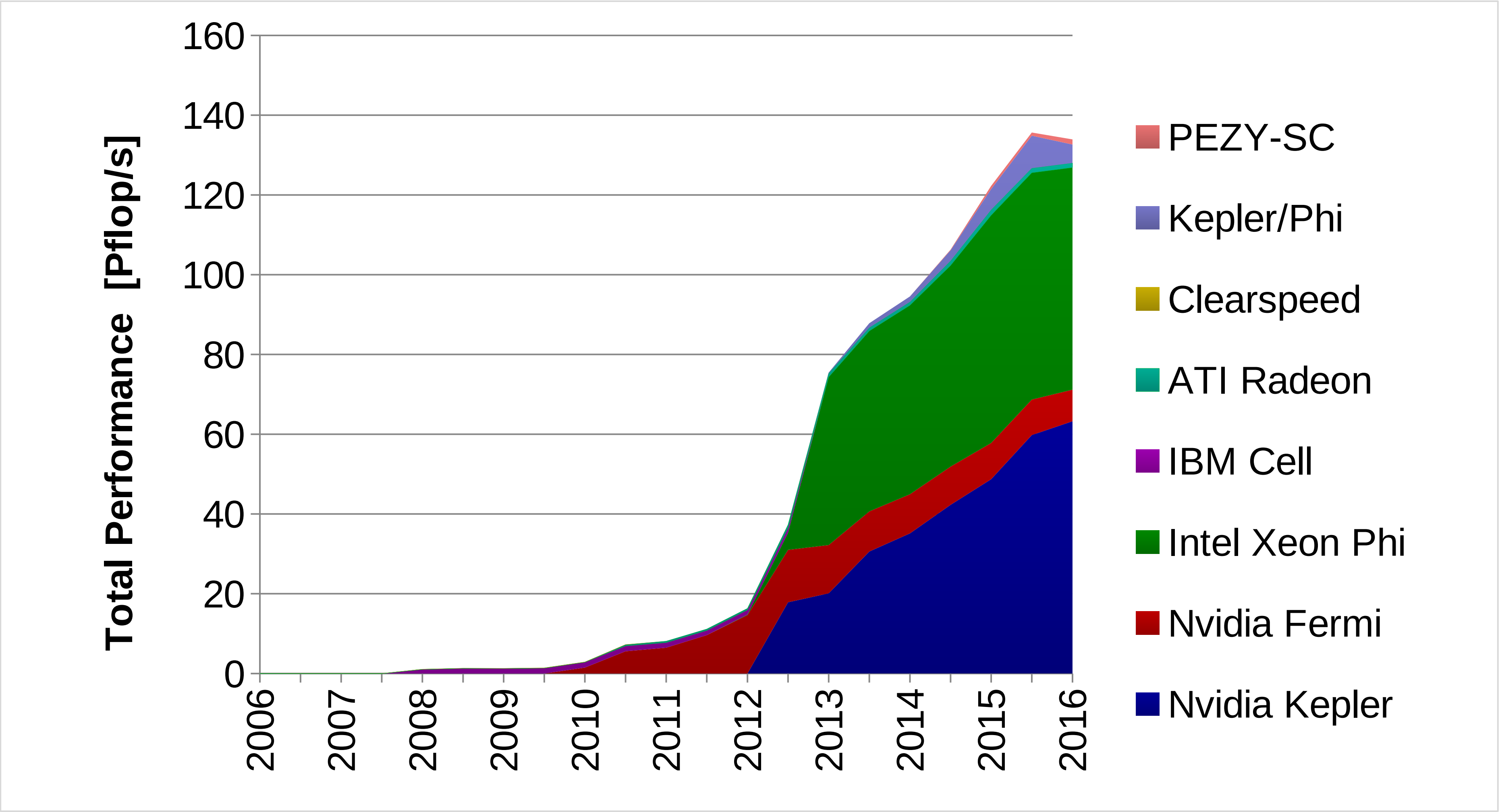
Shared Memory

- A Program is a collection of *threads* of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

Parallel Architectures over time



Performance of Accelerators



Overview of POSIX Threads

- POSIX: Portable Operating System Interface
  - 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
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because *shared memory is implicit*; 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 attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attributes: minimum stack size, priority
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

“Simple” Threading Example

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

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);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Loop Level Parallelism

Many scientific application have parallelism in loops

- With threads:

```
    … my_stuff [n][n];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      … pthread_create (update_cell[i][j], …,
                        my_stuff[i][j]);
```

But overhead of thread creation is nontrivial

- update_cell should have a significant amount of work
- 1/p-th of total work if possible

Data Race Example

Thread 1

for i = 0, n/2-1

s = s + f(A[i])

Thread 2

for i = n/2, n-1

s = s + f(A[i])

- Problem is a race condition on variable (static int) s
- A *race condition* or *data race* occurs when:
  - two processors (or two threads (or more!)) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

threads are working mostly independently

need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

Locks only affect processors using them:

If a thread accesses the data without doing the acquire/release, locks by others will not help

Java, C++, and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks (In Java: synchronized (l) {…})

Can’t forgot to say “release”

Semaphores (a signaling mechanism) generalize locks to allow k threads simultaneous access; good for limited resources.

Unlike in a mutex, a semaphore can be decremented by another process (a mutex can only be unlocked by its owner)

Mutexes in POSIX Threads

- To create a mutex:

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

- To use it:

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

- To deallocate a mutex

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

- Multiple mutexes may be held, but can lead to problems:

thread1	*thread2*
lock(a)	lock(b)
lock(b)	lock(a)

- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Overhead of thread creation is high (1-loop iteration probably too much)
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative
  - Helps with some of these, but doesn’t make them disappear

What is OpenMP?

- OpenMP = Open specification for Multi-Processing
  - openmp.org – Talks, examples, forums, etc.
  - Specification controlled by the Architecture Review Board (ARB)
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
  - A nonprofit organization that controls the OpenMP Spec
  - Latest spec: OpenMP 5.0 (Nov. 2018)
- High-level API for programming in C/C++ and Fortran
  - Preprocessor (compiler) directives (~ 80%)

    #pragma omp construct [clause [clause …]]
  - Library Calls (~ 19%)

    #include <omp.h>
  - Environment Variables (~ 1%)

    all caps

A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Requires compiler support (C, C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than P concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will **not**:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

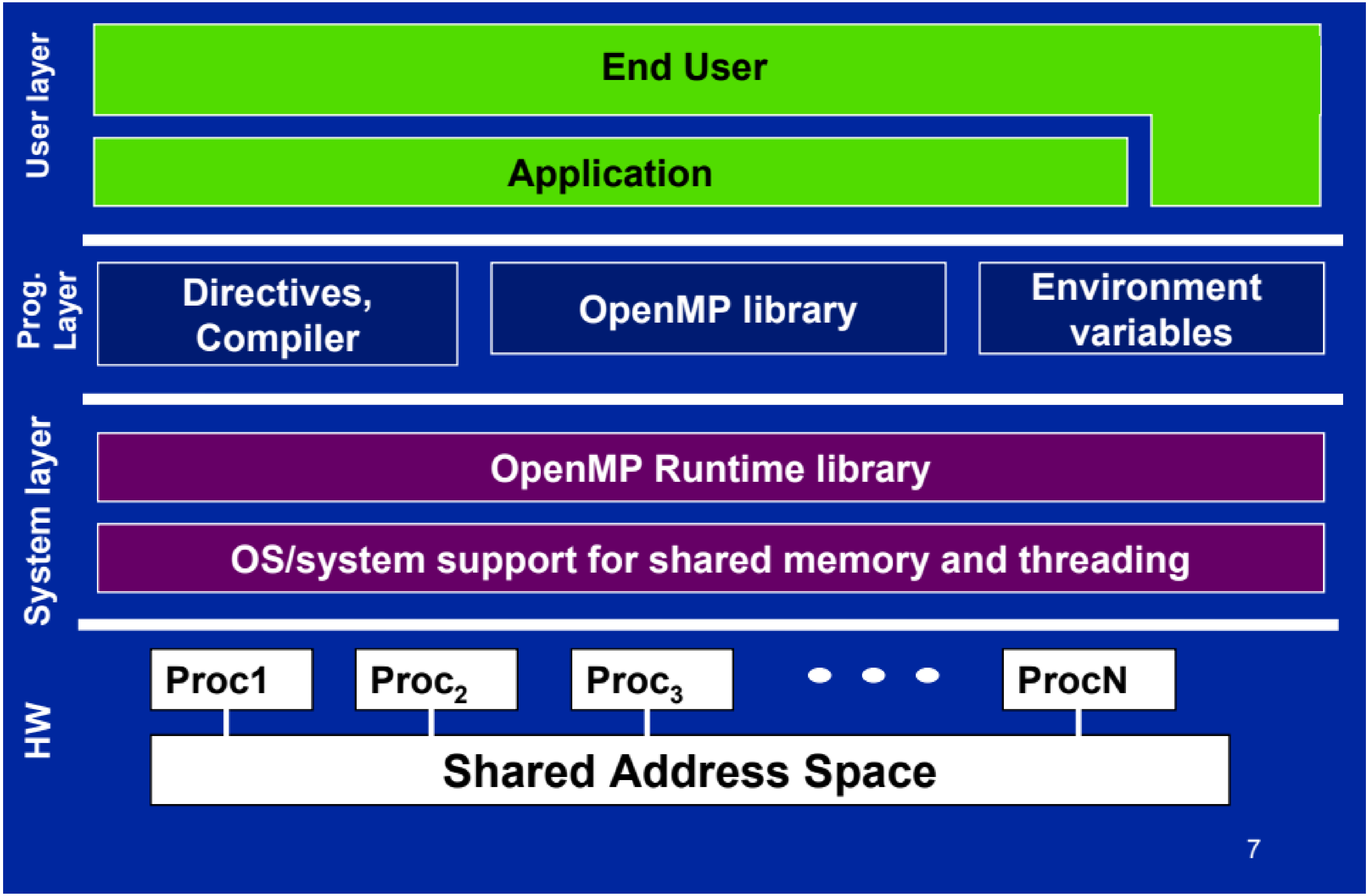
The OpenMP Common Core:
Most OpenMP programs use these 9+10 items

#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

The OpenMP Common Core:
Most OpenMP programs use these 9+10 items

int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup: False Sharing and other performance issues
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)

OpenMP "stack"



OpenMP basic syntax

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

	C and C++	Fortran
Compiler directive	#pragma omp construct [clause [clause] …]	!$OMP construct [clause [clause] …]

Example	#pragma omp parallel private(x)
	{
	    ...
	}
		!$OMP PARALLEL
		    ...
		!$OMP END PARALLEL

Function prototype	#include <omp.h>	use OMP_LIB

Hello World (again...)

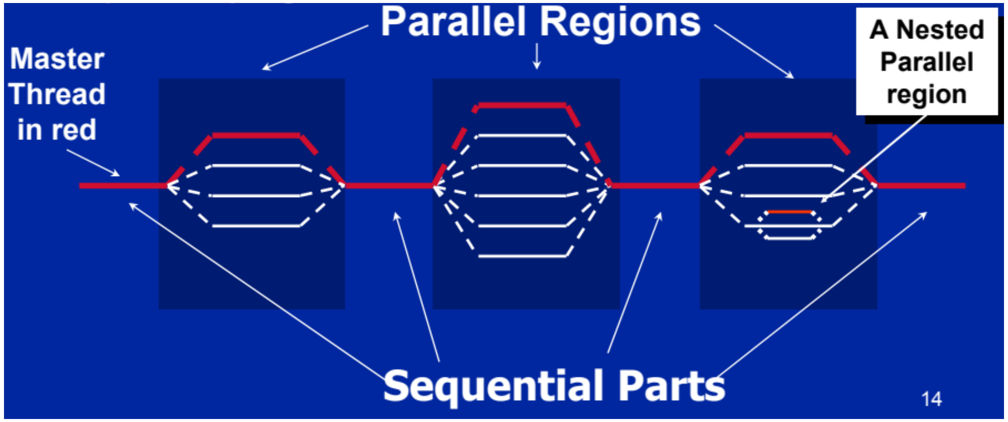
```
#include<stdio.h>
int main()
{
    printf(" hello ");
    printf(" world \n");

}
```

Hello World (again, this time in OpenMP)

```
#include <omp.h>
#include<stdio.h>
int main()
{
  #pragma omp parallel
  {
     printf(“ hello ”);
     printf(“ world \n”);
  }
}
```

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.
- Example, To create a 4 thread Parallel region:

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

- Each thread calls pahrump(ID,A) for ID = 0 to 3
- OpenMP does not have to delete the threads here; it can create a thread pool and reuse it (implementation dependent)
- This is in contrast to pthreads, which is *imperative* (the library will kill all threads by definition during pthread_join)

Thread creation: How many threads did you actually get?

- You create a team threads in OpenMP with the **parallel** construct.
- You can request a number of threads with **omp_set_num_threads()**
- But is the number of threads requested the number you actually get?
  - NO! An implementation can silently decide to give you a team with fewer threads.
  - Once a team of threads is established … the system will not reduce the size of the team.

A fun example

- Mathematically, we know that: $4.0 \int_0^1 1/(1+x^2)\, dx = \pi$
- 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.

Serial Pi

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
int main ()
{
      int i;
      double x, pi, tdata;
      double sum = 0.0;
      step = 1.0/(double) num_steps;
      tdata = omp_get_wtime();
      for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
      tdata = omp_get_wtime() - tdata;
      printf(“ pi = %f in %f secs\n”,pi, tdata);
}
```

Parallel Pi

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 4
void main ()
{
       int i, nthreads;
       double pi, sum[NUM_THREADS], tdata;
       step = 1.0/(double) num_steps;
       omp_set_num_threads(NUM_THREADS);
       tdata = omp_get_wtime();;
         #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;
      tdata = omp_get_wtime() - tdata;
      printf(" pi = %f in %f secs\n",pi, tdata);;
}
```

Serial vs. Parallel Pi

- Serial: pi = 3.141593 in 0.000585 seconds
- Parallel(2): pi = 3.141593 in 0.000620 secs
- Parallel(4): pi = 3.141593 in 0.003407 secs
- Critical section pi(4): pi = 3.141593 in 0.000320 secs

Why such poor scaling? 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 … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Snoopy bus protocol (reprise)

- Basic idea:
  - Broadcast operations on memory bus
  - Cache controllers “snoop” on all bus transactions
  - Memory writes induce serial order
  - Act to enforce coherence (invalidate, update, etc)
- Problems:
  - Bus bandwidth limits scaling
  - Contending writes are slow
- There are other protocol options (e.g. directory-based).
  - But usually give up on *full* sequential consistency.

Weakening sequential consistency

- Try to reduce to the true cost of sharing
- **volatile** tells compiler when to worry about sharing
- Memory fences (see RISC-V ISA for more) tell when to force consistency
- Synchronization primitives (lock/unlock) include fences

Padding

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
#define PAD 8
double step;
#define NUM_THREADS 2
void main ()
{
       int i, nthreads;
       double pi, tdata, sum[NUM_THREADS][PAD];
       step = 1.0/(double) num_steps;
       omp_set_num_threads(NUM_THREADS);
       tdata = omp_get_wtime();;
         #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]=0.0;i< num_steps; i=i+nthrds) {
              x = (i+0.5)*step;
              sum[id][0] += 4.0/(1.0+x*x);
           }
          }
       for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
      tdata = omp_get_wtime() - tdata;
      printf(" pi = %f in %f secs\n",pi, tdata);;
}
```

Synchronization

- High level synchronization included in the common core (the full OpenMP specification has *many* more):
  - Critical section
  - Barrier

Synchronization: critical section

```
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
          res += consume (B); // Thread wait: One at a time...
      }
}
```

Synchronization: Barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code … it is an executable statement.

```
double Arr[8], Brr[8];
int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{   int id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id==0) numthrds = nthrds;
    Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
    Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

Synchronization: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 4
void main ()
{     int nthreads; 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; // scalar sum per thread
    id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)   nthreads = nthrds;
      for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
          x = (i+0.5)*step;
          sum += 4.0/(1.0+x*x); // no array, no sharing
      }
             #pragma omp critical
             pi += sum * step; // critical section avoids conflicts during updates
  }
}
```

Parallelizing loops

- OpenMP easily parallelizes loops
  - Easiest when: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor and runtime calculate loop bounds for each thread directly from serial source
- The loop **#pragma omp for** construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++){ // I is made private to each thread
      big_ugly_calc(I);
   } // All threads wait here before proceeding
}
```

Parallelizing loops (this time with feeling)

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

- Parallel with for: (could put on the same line)

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

Loop scheduling

- `schedule` clause determines how loop iterations are divided among the thread team; no one best way
- **`static([chunk])`** divides iterations statically between threads (default if no hint)
- Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
- Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
- **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
- Forms a logical work queue, consisting of all loop iterations
- Default **`[chunk]`** is 1
- **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

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);
    }
```

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