Shared Memory Parallelism

- Introduction
  - Why shared memory parallelism is important
  - Shared memory architectures
  - POXIS threads vs OpenMP
  - OpenMP history
  - First steps into OpenMP
- Data parallel programs
  - How to divide the work?
- Data scope
- Synchronization
  - memory races
  - locks, critical and atomic operations

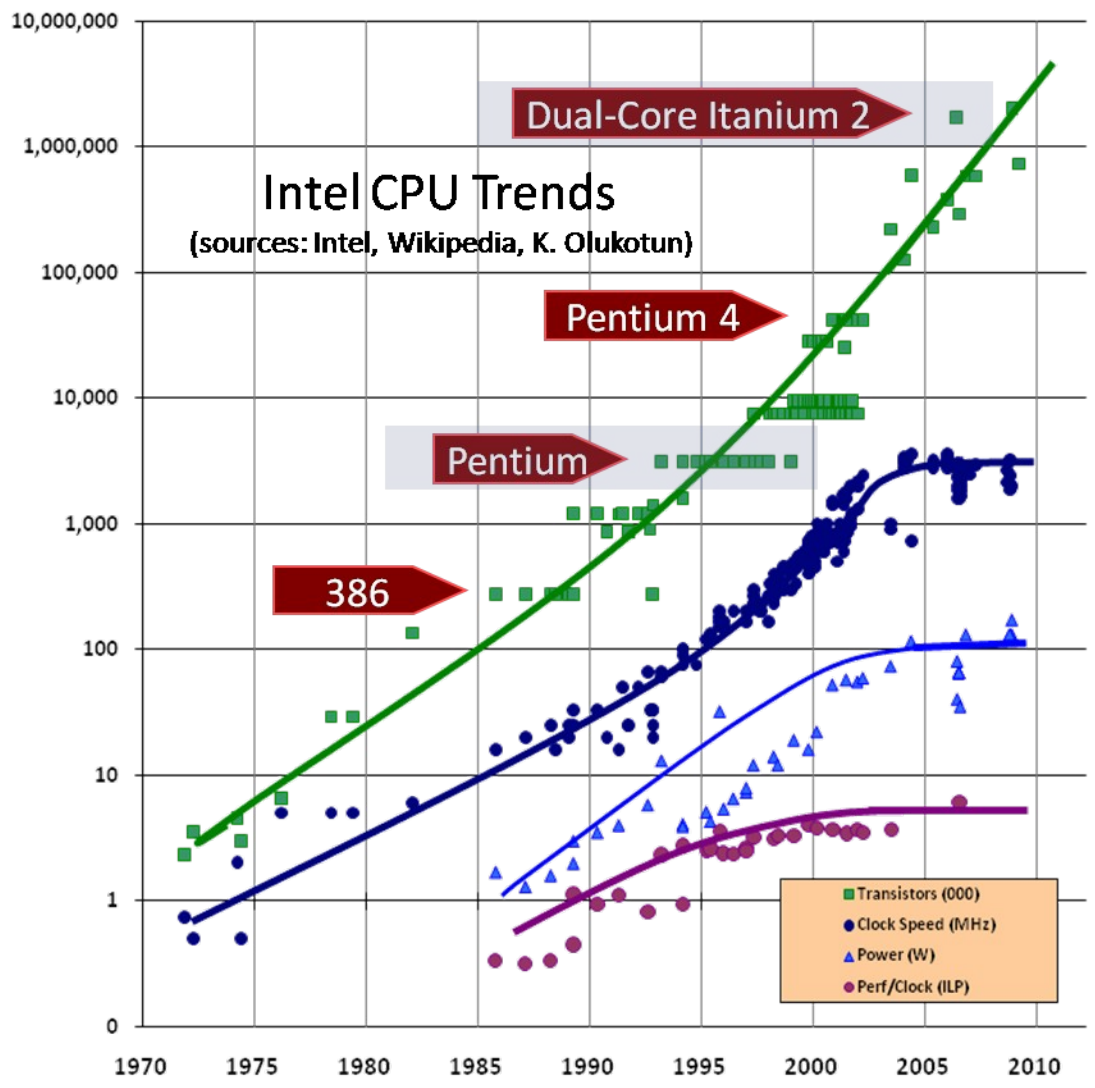
Shared Memory Parallelism

- Directive scoping
- False sharing
- Task parallel programs
  - OpenMP sections
  - OpenMP tasks
- More synchronization
  - flushing, nowait, barriers, locks
- OpenMP functions
  - and the associated environment variables
- Nested parallel programs

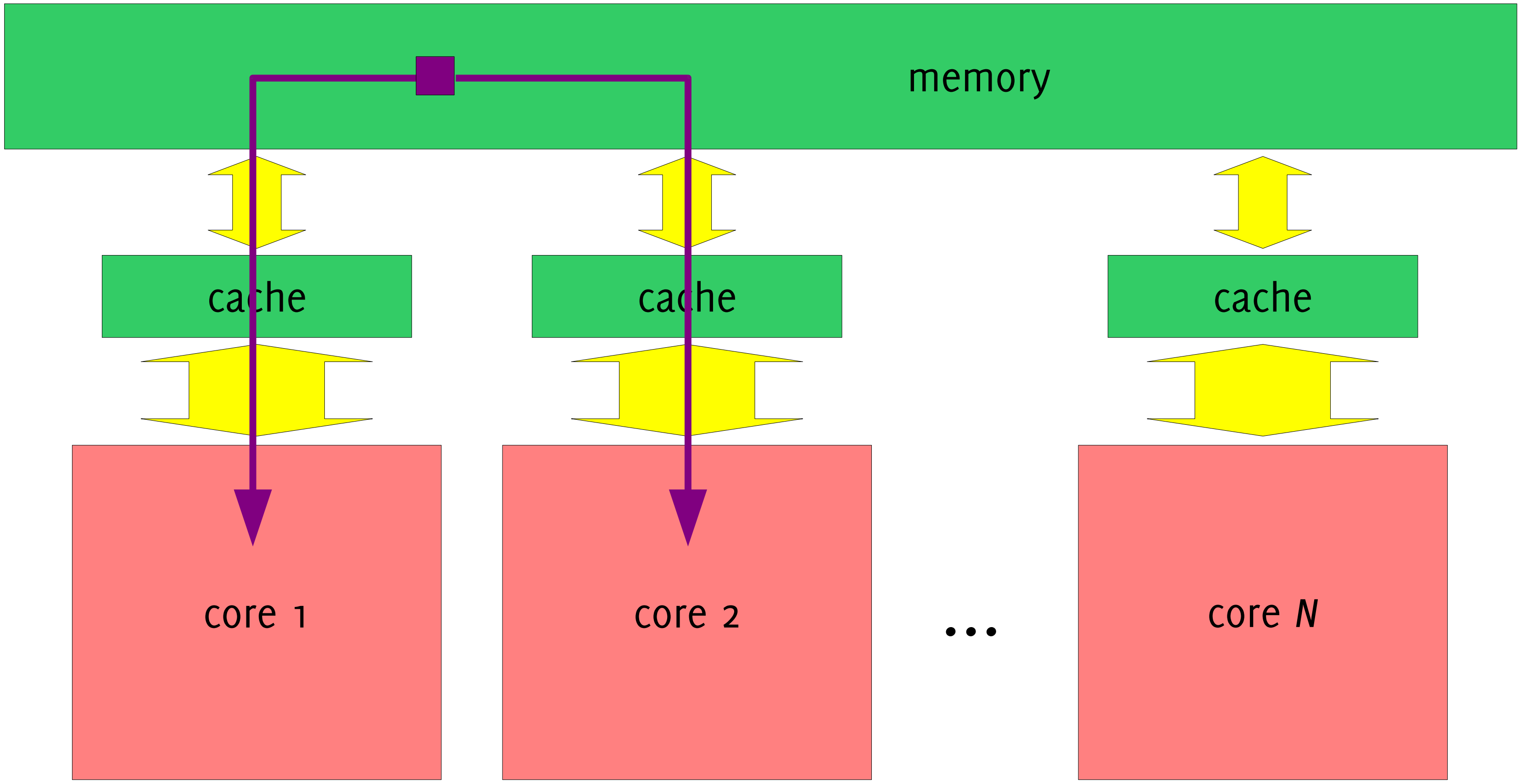
Why?

Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Dr. Dobb's Journal, 30(3), March 2005. (graph updated August 2009)

- Single-threaded applications haven't seen a performance improvement since 2002.
- Chip makers have turned towards multi-core architectures to keep boosting CPU performance.
- Thus: You need to write multi-threaded applications to make best use of your CPU.

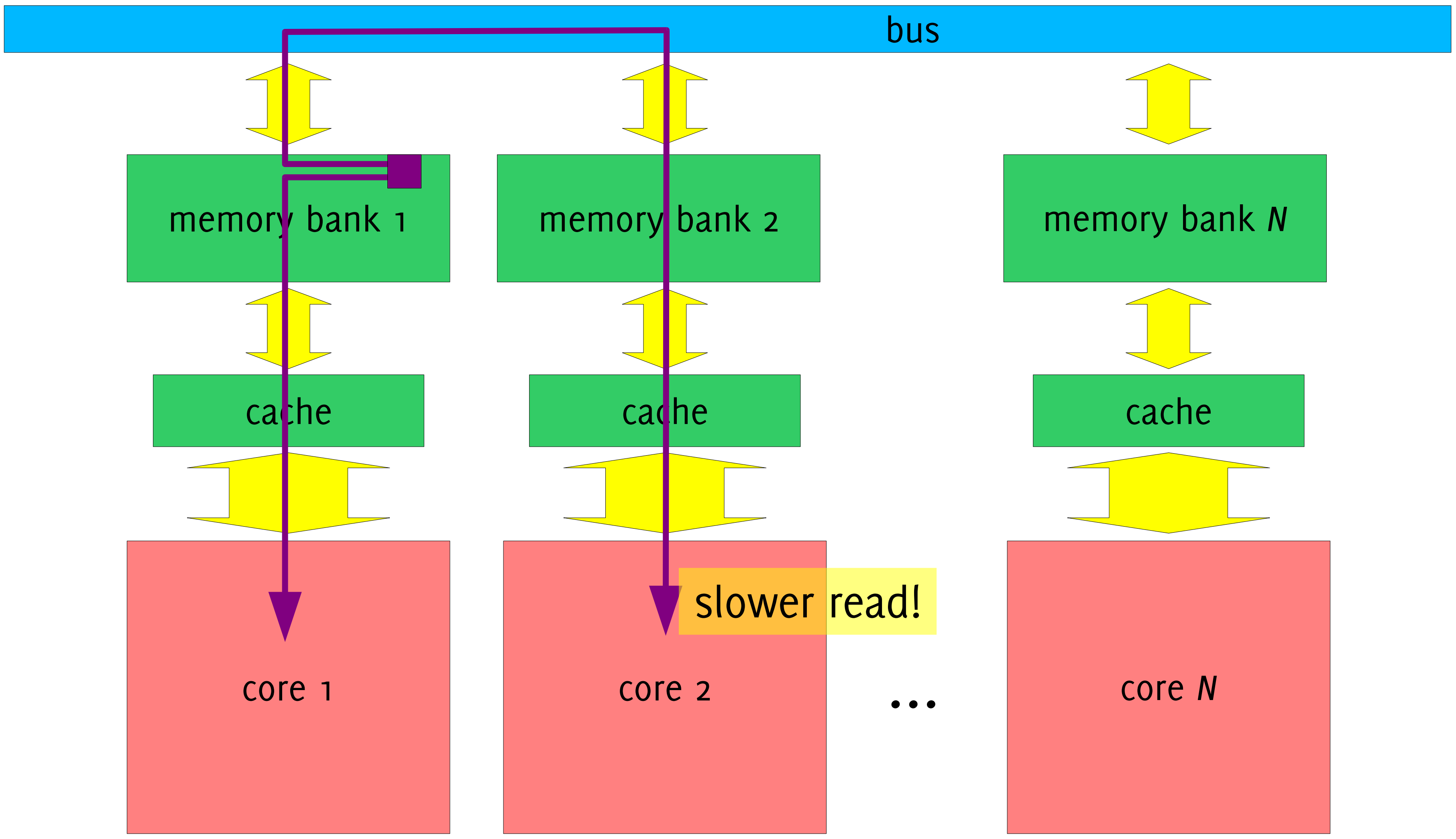


Shared Memory Architectures



Often true for multi-core, single-CPU systems

Shared Memory Architectures



Often true for multi-CPU systems

POSIX Threads vs OpenMP

- POSIX threads are more low-level:
  - Threads need to be created and managed explicitly by the programmer when using POSIX threads.
  - OpenMP directives instruct compiler to generate and manage threads. The directives basically form an abstraction layer on top of the threading mechanism that the compiler chooses to use.
- They are implemented at different levels:
  - POSIX threads need support only from the OS.
  - OpenMP needs support only from the compiler.
- POSIX is an old UNIX standard, but all modern OSes support POSIX (though on Windows you need special addons).
- OpenMP is an established standard, many compilers support it (including GCC and MS Visual Studio).
- OpenMP is (in principle) more portable.

POSIX Threads Example

```
k = N/nthreads
```

decide how to split the task

```
for i = 0 → nthreads
    params[i].start = i*k
    params[i].length = k
```

prepare data for each of the threads

```
...

for i = 0 → nthreads
    pthread_create(&t[i], 0, process_data_func,
                                (void*)&params[i])
```

This function loops over *k* data elements

start the threads

```
for i = 0 → nthreads
    pthread_join(t[i])
```

wait for them to be done

(Pseudo-code, don't try to compile!)

OpenMP Example

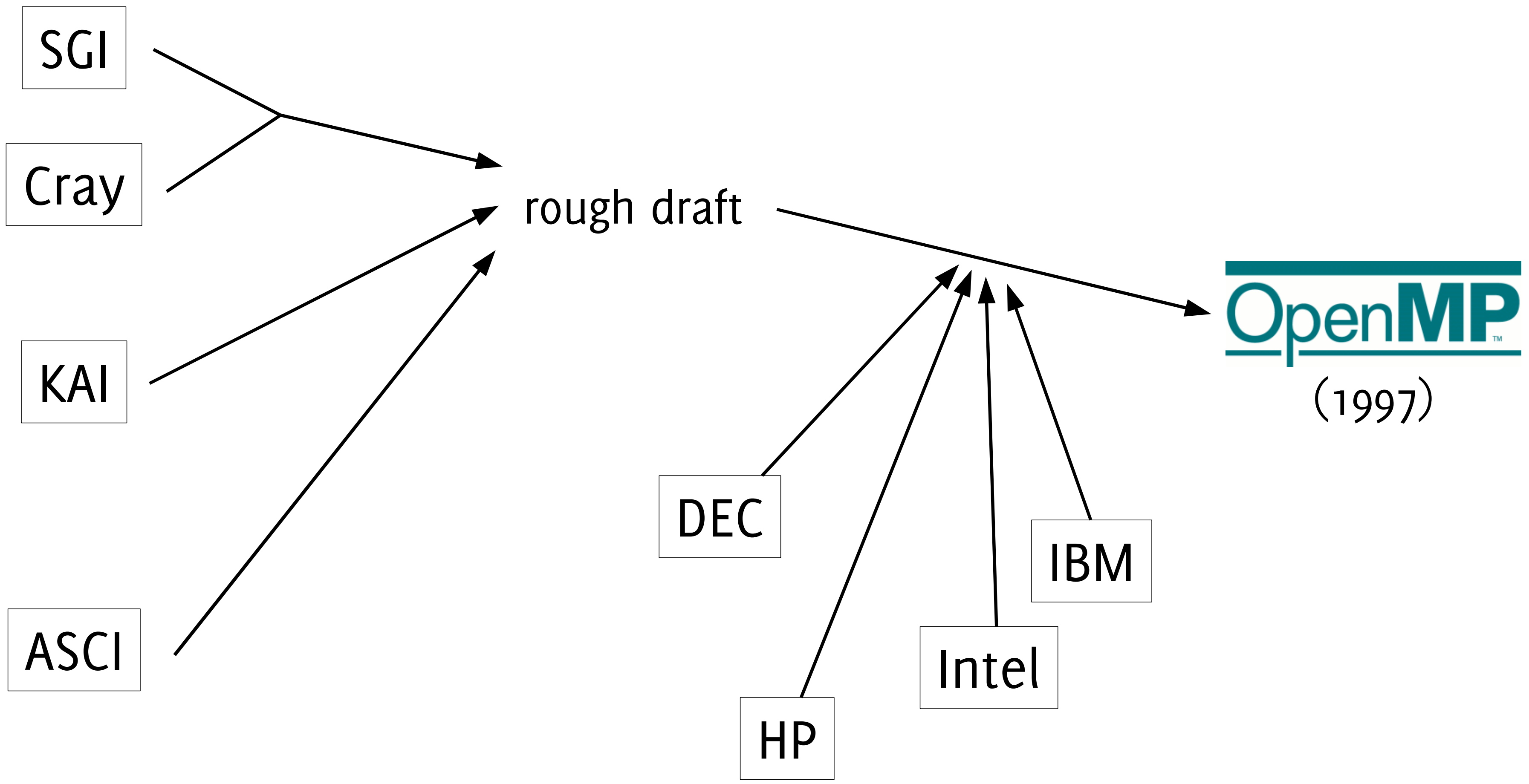
Instructs the compiler to generate code that splits the loop execution over the available threads. At the end of the FOR loop, all threads will be done computing.

```
#pragma omp parallel for
for i = 0 → N
   process_data(i)
```

This function processes one data element (doesn't need to be a function even!)

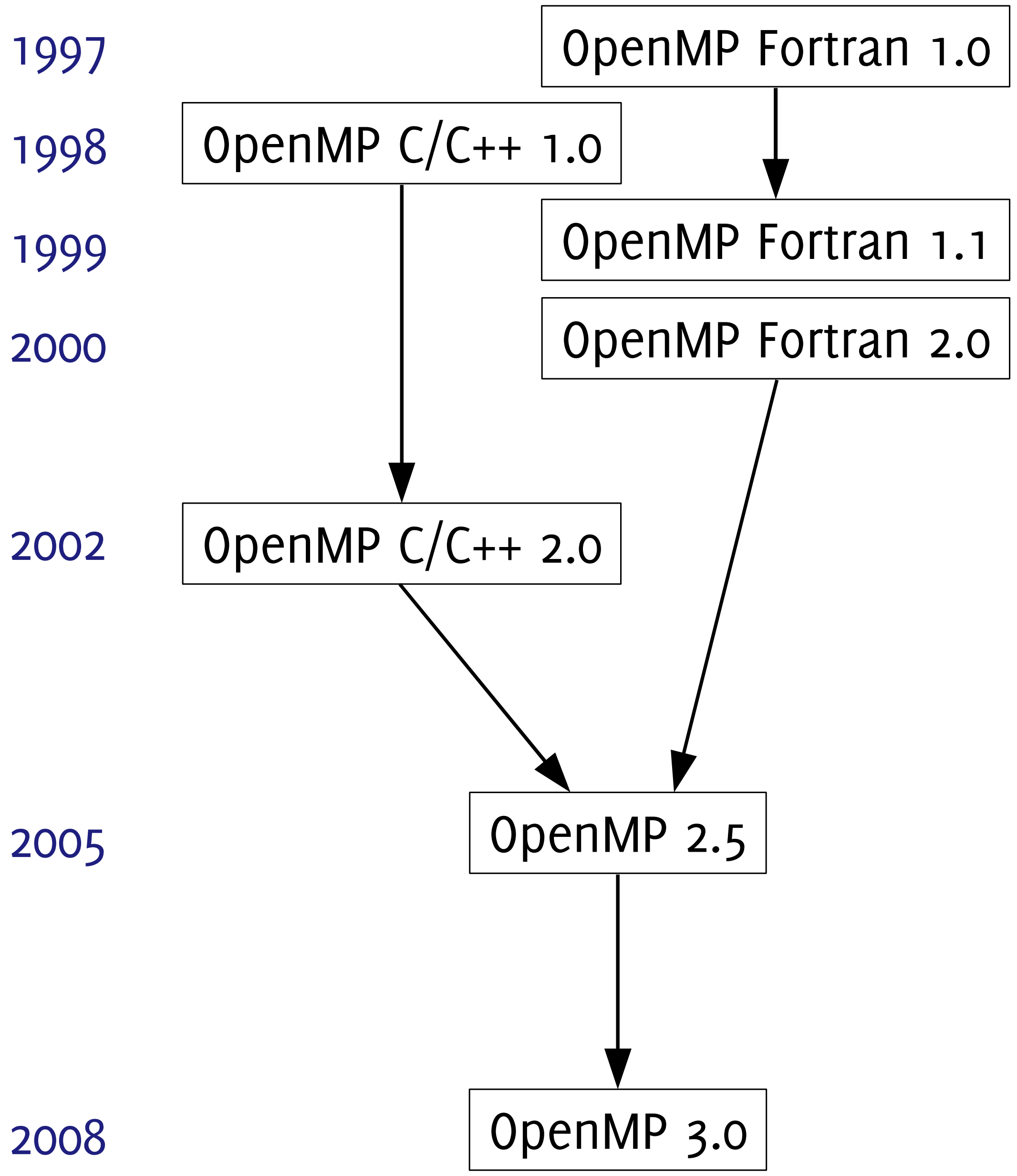
(Pseudo-code, don't try to compile!)

OpenMP History



Currently also: AMD, Fujitsu, NEC, The Portland Group, Oracle, Microsoft, Texas Instruments and CAPS-Entreprise
+ academic and governmental research organizations

OpenMP History



First Steps into OpenMP

```
#include “images.h”

int main (void) {
   uint8 *image;
   int size[2];
   int i, j;
   image_read(“test.tif”, &image, &size);

   for (j=0; j<size[1]; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2; /* image[i][j] /= 2 */
      }
   }
   free(image);
}
```

First Steps into OpenMP

```
#include “images.h”
#include <omp.h>
int main (void) {
   uint8 *image;
   int size[2];
   int i, j;
   image_read(“test.tif”, &image, &size);
   #pragma omp parallel for
   for (j=0; j<size[1]; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2; /* image[i][j] /= 2 */
      }
   }
   free(image);
}
```

Compile using “gcc test.c” ⇒ sequential program
Compile using “gcc -fopenmp test.c” ⇒ parallel program

PARALLEL Directive

- Starts a parallel portion of the code:
  - Creates *N* worker threads (the team)
  - Each thread executes the code in the following block
- Each thread has a thread ID (0 to *N*-1)
  - The master thread has ID = 0, and is a member of the team
- by default, all variables are shared among the workers (shared memory paradigm!)

```
/* Some serial code */
#pragma omp parallel
{
/* This code is executed by all threads */
}
/* Some more serial code */
```

Example

```
#include <omp.h>

main ()  {
   int nthreads, tid;

   #pragma omp parallel private(tid)
   {

      tid = omp_get_thread_num();
      printf("Hello World from thread = %d\n", tid);

      if (tid == 0) {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }

   }

   printf("Good Bye World\n");
}
```

Controlling the Number of Threads

- Different methods, in order of importance:
  - IF clause
  - NUM_THREADS clause
  - `omp_set_num_threads()` library function
  - OMP_NUM_THREADS environment variable
  - Default: implementation dependent

IF Clause to PARALLEL Directive

- Determines whether the code is executed in parallel or serially
- Useful, for example, if gain from parallelization does not offset the penalty of thread creation
  - For example: if each thread only processes 10 pixels, it might be faster to process 4 x 10 pixels in serial.

```
/* Some serial code */
#pragma omp parallel if(n>1000)
{
/* This code is executed by all threads */
}
/* Some more serial code */
```

NUM_THREADS Clause to PARALLEL

- Requests a certain number of threads
- Overrules the value set through the `omp_set_num_threads()` library function, which overrules the value set through the environment variable
- In most cases, you'll use the default number of threads, the user can then choose to change that by setting an environment variable before launching your program

```
/* Some serial code */
#pragma omp parallel num_threads(2)
{
/* This code is executed by all threads */
}
/* Some more serial code */
```

Work-Sharing Constructs

- Allows for different ways of sharing work among the team
- Do not launch new threads
- FOR directive: data-parallel
- SECTIONS directive: task-parallel
- SINGLE directive: sequential portion within a parallel block
  - also: MASTER directive, similar to SINGLE

```
#pragma omp parallel
{
/* This code is executed by all threads */
#pragma omp single
   printf("Only printed once.\n");
/* This code is executed by all threads */
}
```

Data Parallelism

FOR Directive

- Divides the loop iterations over the workers
- Must be inside a PARALLEL block
- Loop cannot be a `while` loop, or suchlike (e.g. `break`)
- You cannot control which thread will execute which iteration
- But you can control the scheduling:
  - SCHEDULE clause

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

Combining PARALLEL and FOR

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

$=$

```
#pragma omp parallel for
   for (ii=0; ii<N; ii++)
      a[ii] = ...;
```

```
#pragma omp parallel
{
#pragma omp for
   for (ii=0; ii<N; ii++)
      a[ii] = ...;

#pragma omp for
   for (ii=0; ii<N; ii++)
      b[ii] = ...;
}
```

$\neq$

```
#pragma omp parallel for
   for (ii=0; ii<N; ii++)
      a[ii] = ...;

#pragma omp parallel for
   for (ii=0; ii<N; ii++)
      b[ii] = ...;
```

SCHEDULE Clause to FOR Directive

- 3 scheduling modes:
  - `static`: divides iterations into blocks of *chunksize* elements, statically assigned to threads default chunksize = N/nthreads
  - `dynamic`: same, but dynamically assigned to threads – useful if the work at each iteration can vary default chunksize = 1
  - `guided`: dynamic scheduling, but with decreasing block size
- Other options:
  - `runtime`: use whatever the OMP_SCHEDULE environment variable says
  - `auto`: the compiler or runtime system decides what to do

```
#pragma omp for schedule(type, chunksize)
```

Scheduling examples

```
#pragma omp parallel for \
      num_threads(10) schedule(static)
for (ii=0; ii<1000; ii++) {
   a[ii] = b[ii] + c[ii];
}
```

thread 0: 0-99
thread 1: 100-199
thread 2: 200-299
...

```
#pragma omp parallel for \
      num_threads(10) schedule(static,1)
for (ii=0; ii<1000; ii++) {
   a[ii] = b[ii] + c[ii];
}
```

thread 0: 0,10,20,...
thread 1: 1,11,21,...
thread 2: 2,12,22,...
...

```
#pragma omp parallel for \
      num_threads(10) schedule(guided)
for (ii=0; ii<1000; ii++) {
   a[ii] = b[ii] + c[ii];
}
```

thread 0: 0-99
thread 1: 100-189
thread 2: 190-270
...

Back to our first OpenMP program

```
#include “images.h”
#include <omp.h>
int main (void) {
   uint8 *image;
   int size[2];
   int i, j;
   image_read(“test.tif”, &image, &size);
   #pragma omp parallel for
   for (j=0; j<size[1]; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2;
      }
   }
   free(image);
}
```

How is this task split?

Division of Labour

- Given *N* threads:
  - index *j* is divided into *N* chunks of *size*[1]/*N*

```
/* thread ID = 0 */
for (j=0; j<size[1]/N; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2;
      }
}
```

```
/* thread ID = 1 */
for (j=size[1]/N; j<2*size[1]/N; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2;
      }
}
```

What if size[1] is small?

```
#include “images.h”
#include <omp.h>
int main (void) {
   uint8 *image;
   int size[2];
   int i, j;
   image_read(“test.tif”, &image, &size);
   #pragma omp parallel for collapse(2)
   for (j=0; j<size[1]; j++) {
      for (i=0; i<size[0]; i++) {
         *(image+i+j*size[0]) /= 2;
      }
   }
   free(image);
}
```

How is this task split now?

COLLAPSE Clause to FOR Directive

- *K* for loops are collapsed into a single loop, which is then parallelized
- Given *N* threads:
  - *i and j* are divided into *N* chunks of (size[0]**size*[1])/*N*

```
for (ij=0; ij<size[0]*size[1]; ij++) {
         *(image+ij) /= 2;
}
```

Another Example of Labour Division

```
#include “images.h”
#include <omp.h>
int main (void) {
   uint8 *image, *tmp;
   int size[2];
   const int N = 16;
   float granulometry[N];
   int i;
   image_read(“test.tif”, &image, &size);
   #pragma omp parallel for private(tmp)
   for (i=0; i<N; i++) {
      image_closing(image,&tmp,2*(i+2));
      granulometry[i] = image_sum(tmp);
   }
   free(image);
}
```

Another Example of Labour Division

- 16 iterations, different amount of work:

$$\left.\begin{array}{l}
-\; i = 0{:}\ k^*(2^*(i+2))^2 = k^*16 \\
-\; i = 1{:}\ k^*(2^*(i+2))^2 = k^*36 \\
-\; i = 2{:}\ k^*(2^*(i+2))^2 = k^*64 \\
-\; i = 3{:}\ k^*(2^*(i+2))^2 = k^*100
\end{array}\right\} k^*216$$

$$\left.\begin{array}{l}
-\; i = 4{:}\ k^*(2^*(i+2))^2 = k^*144 \\
-\; i = 5{:}\ k^*(2^*(i+2))^2 = k^*196 \\
-\; i = 6{:}\ k^*(2^*(i+2))^2 = k^*256 \\
-\; i = 7{:}\ k^*(2^*(i+2))^2 = k^*324
\end{array}\right\} k^*920$$

$$\left.\begin{array}{l}
-\; i = 8{:}\ k^*(2^*(i+2))^2 = k^*400 \\
-\; i = 9{:}\ k^*(2^*(i+2))^2 = k^*484 \\
-\; i = 10{:}\ k^*(2^*(i+2))^2 = k^*576 \\
-\; i = 11{:}\ k^*(2^*(i+2))^2 = k^*676
\end{array}\right\} k^*2136$$

$$\left.\begin{array}{l}
-\; i = 12{:}\ k^*(2^*(i+2))^2 = k^*784 \\
-\; i = 13{:}\ k^*(2^*(i+2))^2 = k^*900 \\
-\; i = 14{:}\ k^*(2^*(i+2))^2 = k^*1024 \\
-\; i = 15{:}\ k^*(2^*(i+2))^2 = k^*1156
\end{array}\right\} k^*3864 \xleftarrow{\quad}$$

~18 times as much work as tread #0!

Another Example of Labour Division

```
#include “images.h”
#include <omp.h>
int main (void) {
   uint8 *image, *tmp;
   int size[2];
   const int N = 16;
   float granulometry[N];
   int i;
   image_read(“test.tif”, &image, &size);
   #pragma omp parallel for private(tmp) schedule(dynamic,1)
   for (i=0; i<N; i++) {
      image_closing(image,&tmp,2*(i+2));
      granulometry[i] = image_sum(tmp);
   }
   free(image);
}
```

Maybe also reverse loop direction?

What is a Memory Race?

- When multiple threads read and write to the same variable, there is a memory race
- This can be a difficult bug to find!
- There are many tools in OpenMP to avoid memory races

```
int n;
#pragma omp parallel
{
   n = omp_get_thread_num();
   printf("Thread ID = %d\n", n);
}
printf("The variable n now has the value %d\n", n);
```

What is a Memory Race?

- When multiple threads read and write to the same variable, there is a memory race
- This can be a difficult bug to find!
- There are many tools in OpenMP to avoid memory races

is now undefined!!!

```
int n;
#pragma omp parallel private(n)
{
   n = omp_get_thread_num();
   printf(“Thread ID = %d\n”, n);
}
printf(“The variable n now has the value %d\n”, n);
```

Data Scoping

- Most variables are shared by default
- Private variables are:
  - loop index variables
  - any variable declared inside a function called within the parallel section
- Data scope attribute clauses change the variable scope:
  - DEFAULT: changes the default for all variables
    - `default(private), default(shared), default(none)`
  - SHARED: the variable is shared among threads
    - `shared(var1,var2,...)`
  - PRIVATE: the variable is private to the thread
    - `private(var1,var2,...)`
  - FIRSTPRIVATE: same as PRIVATE, but copies value at start
  - LASTPRIVATE: same as PRIVATE, but copies value at end

LASTPRIVATE Clause to PARALLEL

- Copies the *sequentially last* value of the variable back to the original variable object of the enclosing construct
  - That is, the team member that performs the last iteration of a *for* loop copies it's value over.

is now defined

```
int n;
#pragma omp parallel lastprivate(n)
{
   n = omp_get_thread_num();
   printf("Thread ID = %d\n", n);
}
printf("The variable n now has the value %d\n", n);
```

FIRSTPRIVATE Clause to PARALLEL

- Copies the current value of a variable to the private copies of each thread

```
int n = 0;
#pragma omp parallel private(n)
{
   n += omp_get_thread_num();
   printf(“Thread ID = %d\n”, n);
}
```

Error!
n is uninitialised!

```
int n = 0;
#pragma omp parallel firsprivate(n)
{
   n += omp_get_thread_num();
   printf(“Thread ID = %d\n”, n);
}
```

Synchronization

- When multiple threads *need* to write to the same variable or memory address, synchronization between threads is needed

```
double a[N];
double norm = 0;

... (initialise a)

#pragma omp parallel for
for (ii=0; ii<N; ii++) {
   norm += a[ii]*a[ii];
}

norm = sqrt(norm);
printf(“norm = %f\n”, norm);
```

Error!
data race

Solution 1: Locks

```
double a[N];
double norm = 0;
double tmp;
omp_lock_t lock;

... (initialise a)

omp_init_lock(&lock);

#pragma omp parallel for private(tmp)
for (ii=0; ii<N; ii++) {
   tmp = a[ii]*a[ii];
   omp_set_lock(&lock);
   norm += tmp;
   omp_unset_lock(&lock);
}

omp_destroy_lock(&lock);

norm = sqrt(norm);
printf(“norm = %f\n”, norm);
```

Solution 2: CRITICAL Directive

```
double a[N];
double norm = 0;
double tmp;

... (initialise a)

#pragma omp parallel for private(tmp)
for (ii=0; ii<N; ii++) {
   tmp = a[ii]*a[ii];
   #pragma omp critical name
      norm += tmp;
}

norm = sqrt(norm);
printf(“norm = %f\n”, norm);
```

Solution 3: ATOMIC Directive

```
double a[N];
double norm = 0;
double tmp;

... (initialise a)

#pragma omp parallel for private(tmp)
for (ii=0; ii<N; ii++) {
   tmp = a[ii]*a[ii];
   #pragma omp atomic
      norm += tmp;
}

norm = sqrt(norm);
printf(“norm = %f\n”, norm);
```

Solution 4: REDUCTION Clause

(private and initialised to 0)

```
double a[N];
double norm;

... (initialise a)

#pragma omp parallel for reduction(+:norm)
for (ii=0; ii<N; ii++) {
   norm += a[ii]*a[ii];
}

norm = sqrt(norm);
printf("norm = %f\n", norm);
```

REDUCTION Clause Implementation

```
double a[N];
double norm = 0;
double tmp;

... (initialise a)

#pragma omp parallel private(tmp)
{
   tmp = 0;
   #pragma omp for
   for (ii=0; ii<N; ii++) {
      tmp += a[ii]*a[ii];
   }
   #pragma omp atomic
      norm += tmp;
}

norm = sqrt(norm);
printf(“norm = %f\n”, norm);
```

But:
Execution order could affect numerical result!

Directive Scoping

- The FOR, SECTIONS and SINLGE directive must occur within a PARALLEL directive
- But: this doesn’t necessarily have to happen within the same function (or even file!)

```
void main(void) {
   dothework();
   #pragma omp parallel
   {
      dothework();
   }
}
```

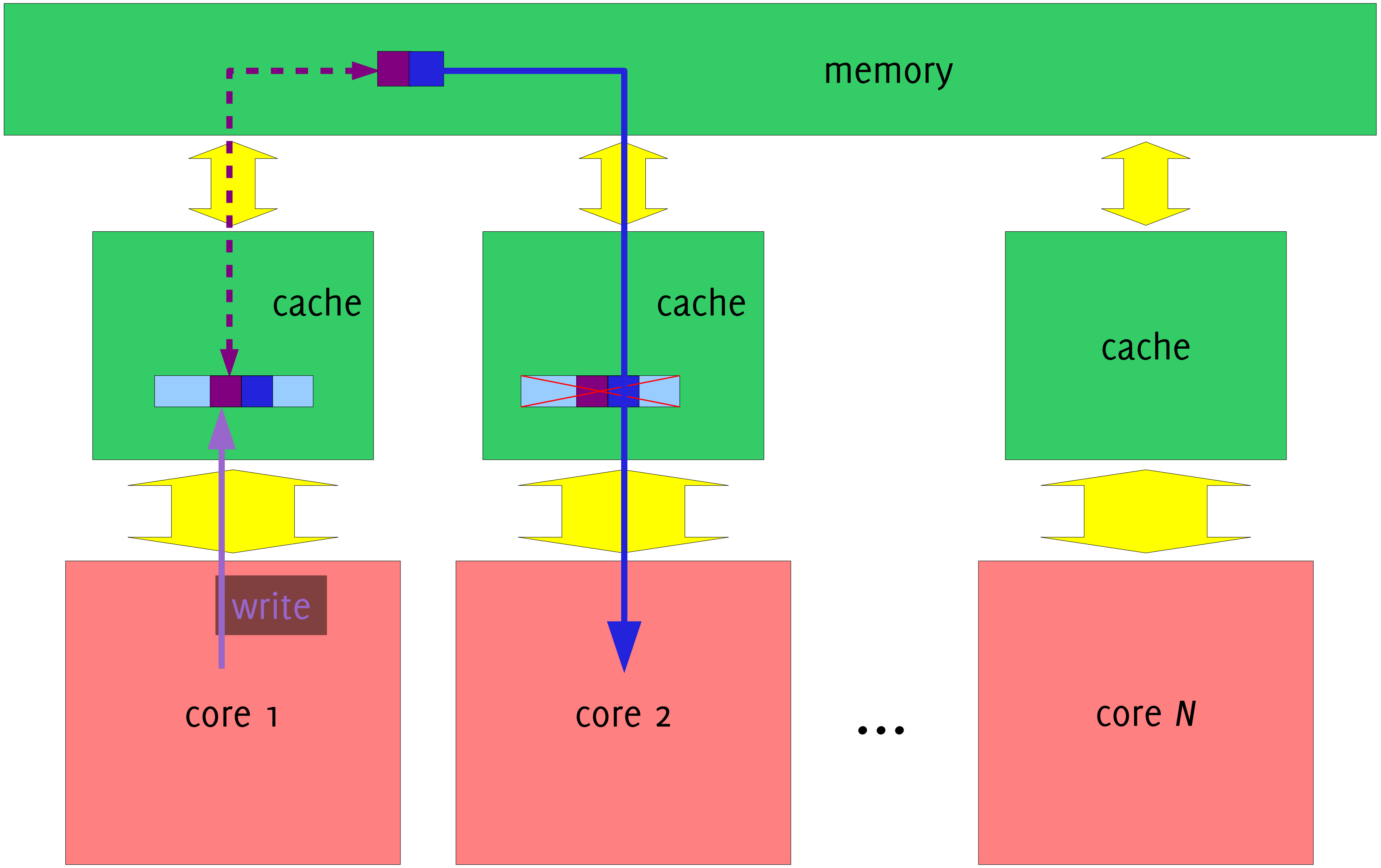
```
void dothework(void) {
   #pragma omp for
      for (ii=0; ii<N; ii++) {
         ...
      }
}
```

False Sharing

- Independent data elements might be less independent than you think!
  - Memory addresses are grouped into *cache lines*
  - If one element of the cache line is changed, the whole line is invalidated

```
float data[N], total = 0;
int ii;
#pragma omp parallel num_threads(N)
{
   int n = omp_get_thread_num();
   data[n] = 0;
   while(moretodo(n))
      data[n] += calculate_something(n);
}
for (ii=0; ii<N; ii++)
   total += data[n];
```

False Sharing



False Sharing

- Avoid false sharing by using private variables, for example

```
float data, total = 0;

#pragma omp parallel num_threads(N) private(data)
{
   int n = omp_get_thread_num();
   data = 0;
   while(moretodo(n))
      data += calculate_something(n);
   #pragma omp critical
      total += data;
}
```

False Sharing

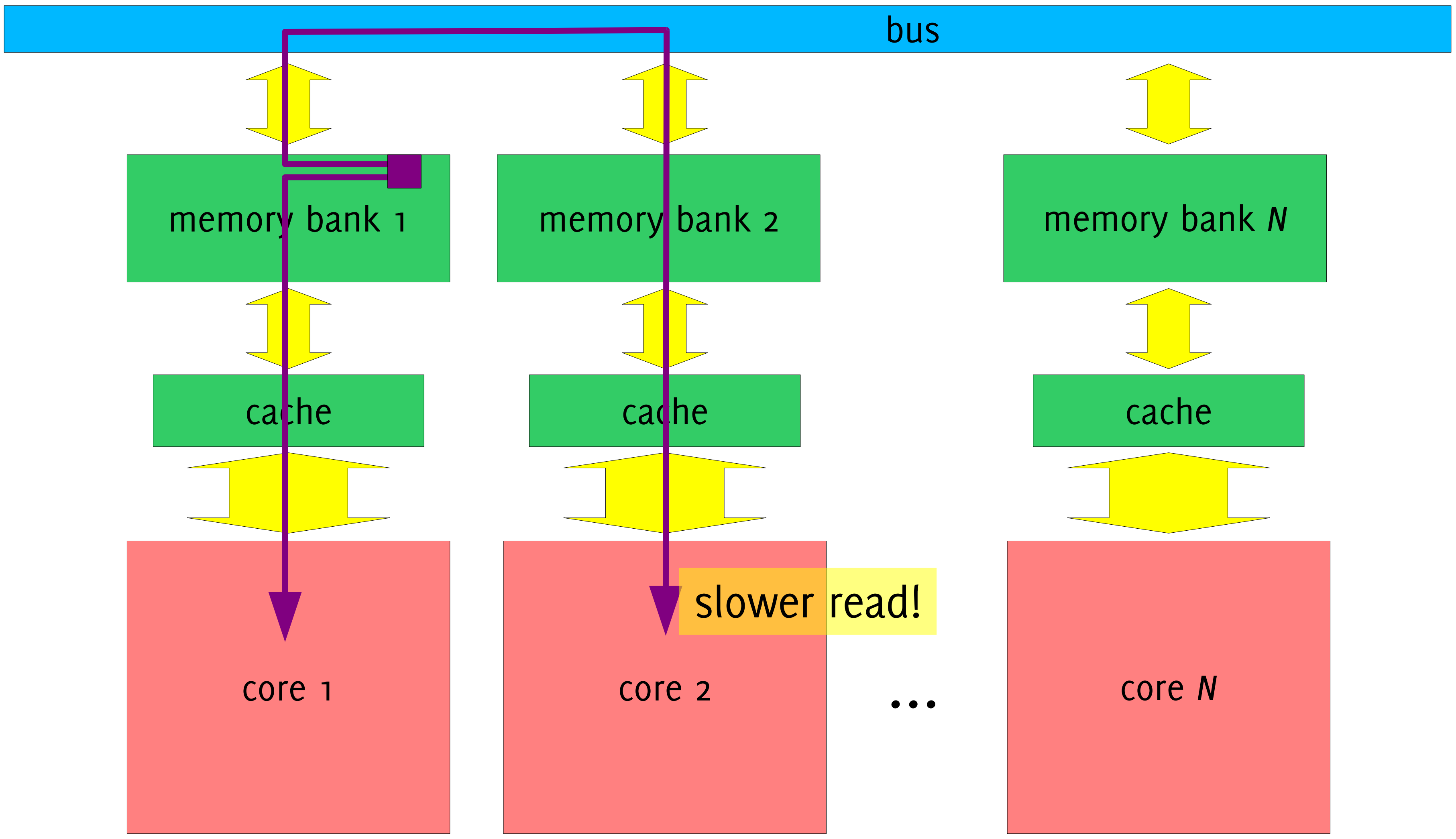
```
#pragma omp parallel for
   for (j=0; j<N; j++) {
      for (i=0; i<M; i++) {
         image[ i + j*M ] /= 2;
      }
   }
```

Each thread accesses a contiguous region in memory:
not much false sharing.

```
#pragma omp parallel for
   for (i=0; i<M; i++) {
      for (j=0; j<N; j++) {
         image[ i + j*M ] /= 2;
      }
   }
```

Each thread accesses alternating memory elements:
a lot of false sharing!

How to Distribute Data in Memory?



Often true for multi-CPU systems

How to Distribute Data in Memory?

- OpenMP cannot dictate where data is to be placed
- Both Windows and Linux use the “first touch” principle to place data

```
float *data = malloc(N*sizeof(float));
memset(data,0,N*sizeof(float));

#pragma omp parallel
{
   #pragma omp for
      for (ii=0; ii<N; ii++) {
         data[ii] = dosomething(data[ii]);
      }
}
```

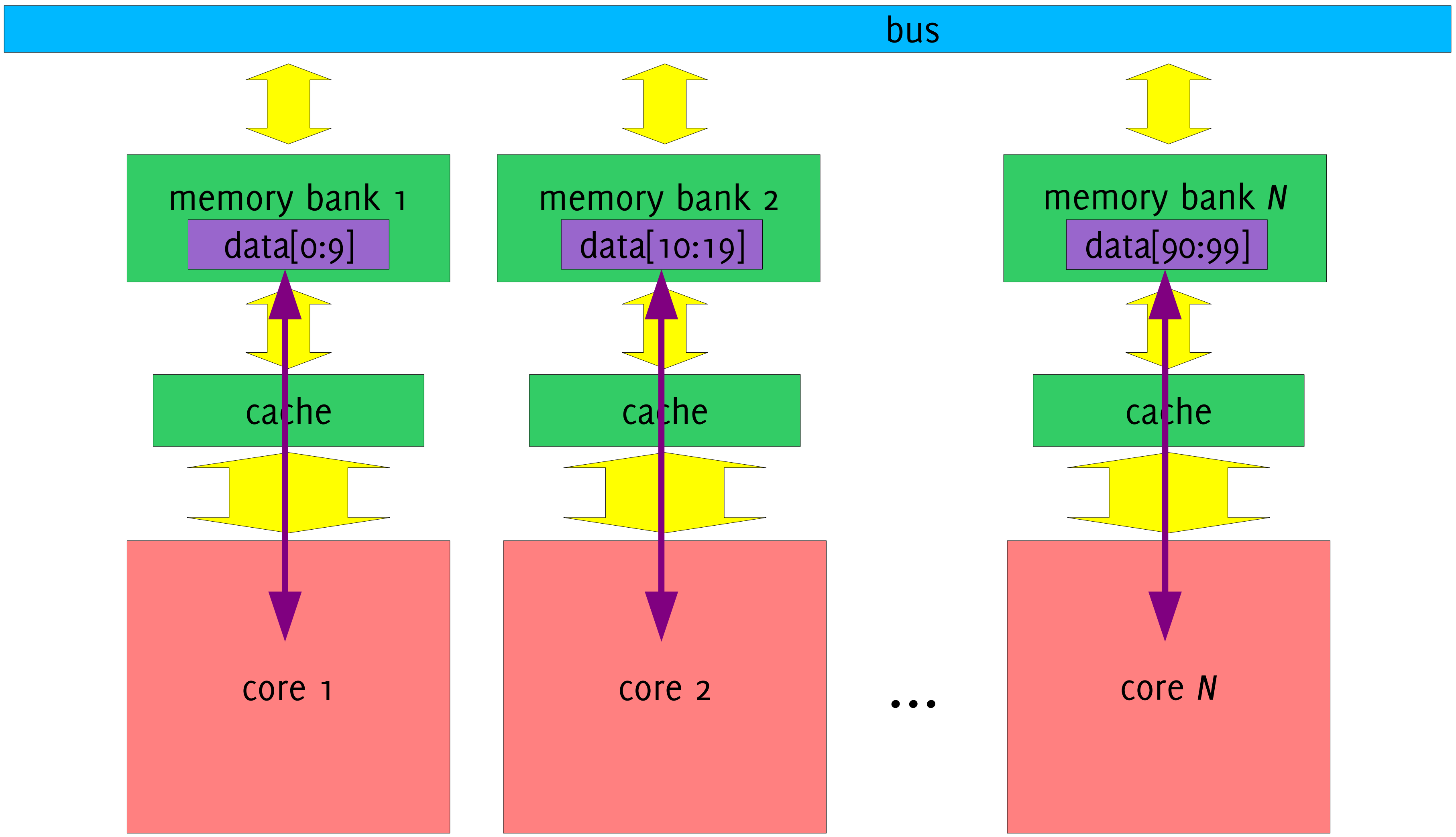
How to Distribute Data in Memory?

```
float *data = malloc(N*sizeof(float));

#pragma omp parallel
{
   #pragma omp for schedule(static)
      for (ii=0; ii<N; ii++)
         data[ii] = 0;

   #pragma omp for schedule(static)
      for (ii=0; ii<N; ii++) {
         data[ii] = dosomething(data[ii]);
      }
}
```

How to Distribute Data in Memory?



Often true for multi-CPU systems

Task Parallelism

Task Parallelism in OpenMP

- Sometimes it is possible to divide an algorithm into *different* independent tasks
- For example, one image analysis task needs to:
  - correct for uneven illumination
  - segment cells

    $\left.\begin{array}{l}\text{correct for uneven illumination}\\ \text{segment cells}\end{array}\right\}$ independent!
  - measure fluorescence intensity with each cell
- OpenMP provides constructs for this as well:
  - SECTIONS directive
  - TASK construct
- Synchronization might be necessary!
  - MASTER, BARRIER, TASKWAIT, FLUSH

SECTIONS Directive

- The SECTIONS directive is followed by a block
- This block contains a set of SECTION directives
- Each SECTION directive is followed by a block
- Each of these blocks is executed by one thread in the team
- When exiting the SECTIONS block, all SECTION blocks are finished

```
#pragma omp sections
{
   #pragma omp section
      /* task 1 */
   #pragma omp section
      /* task 2 */
   #pragma omp section
      /* task 3 */
}
```

SECTIONS Directive

```
image_read(“test.tif”, &image, &size);

#pragma omp parallel num_threads(2)
{
   #pragma omp sections
   {

      #pragma omp section
         image_correct(image, &corrected);

      #pragma omp section
      {
         image_segment(image, &labs);
         image_label(labs, &labs);
      }

   }
}

image_measure(labs, corrected, &measurements)
```

SECTIONS Directive

- What happens when there's fewer SECTION directives than threads?
- What happens where there's more?

```
#pragma omp sections private(a,b,c)
{
   #pragma omp section
      /* task 1 */
   #pragma omp section
      /* task 2 */
   #pragma omp section
      /* task 3 */
}
```

TASK Construct

- Much, much more flexible way of scheduling tasks
- Tasks can generate new tasks
- New to OpenMP 3.0
  - i.e. requires GCC 4.4 or later

```
#pragma omp task
   /* task 1 */
#pragma omp task
   /* task 2 */
#pragma omp task
   /* task 3 */

#pragma omp taskwait
```

```
#pragma omp task private(a)
{
   a = somefunction(0);
}
```

```
#pragma omp task if(notinahurry)
   /* task */
```

Task Scheduling

- Task scheduling is implementation-dependent
- Tasks are tied to a thread by default, but can be UNTIED
- Tasks can be interrupted at scheduling points:
  - after creating a new task
  - after the last instruction
  - at a TASKWAIT directive
  - at implicit and explicit barriers
  - anywhere in an untied task
- At a scheduling point, a thread can:
  - begin execution of any task bound to the current team
  - resume any suspended task tied to the thread
  - resume any suspended untied task bound to the current team

TASK Examples

```
#pragma omp parallel
{
   #pragma omp task
      traverse(p);
}
```

p is firstprivate by default

```
void traverse(struct node *p) {
   if (p->left)
      #pragma omp task
         traverse(p->left);
   if (p->right)
      #pragma omp task
         traverse(p->right);
   #pragma omp taskwait
   process(p);
}
```

TASK Examples

```
#pragma omp parallel
{
   #pragma omp single
   {
      int i;
      for (i=0; i<LARGE_NUMBER; i++)
         #pragma omp task
            process(item[i]);
   }
}
```

The thread generating the tasks can stop half way the loop to start working on tasks (e.g. if the queue is full). If it happens to pick up some really long task, the other threads might finish all the tasks in the queue and then wait for more tasks to be scheduled.

TASK Examples

```
#pragma omp parallel
{
   #pragma omp single
   {
      int i;
      #pragma omp task untied
      {
         for (i=0; i<LARGE_NUMBER; i++)
            #pragma omp task
               process(item[i]);
      }
   }
}
```

More on Synchronization: Barriers

```
#pragma omp parallel
{

   #pragma omp for
      for (ii=0; ii<N; ii++)
         c[ii] = a[ii]*a[ii];

   #pragma omp for
      for (ii=0; ii<N; ii++)
         d[ii] = sqrt(b[ii]);

}
```

threads synchronize here

More on Synchronization: Barriers

```
#pragma omp parallel
{

   #pragma omp for nowait
      for (ii=0; ii<N; ii++)
         c[ii] = a[ii]*a[ii];

   #pragma omp for nowait
      for (ii=0; ii<N; ii++)
         d[ii] = sqrt(b[ii]);

}
```

All 3 work-sharing constructs have an implied barrier at the end:

- FOR
- SECTIONS
- SINGLE

There is (obviously) also an inplied barrier at the end of the PARALLEL construct!

More on Synchronization: Barriers

```
#pragma omp parallel
{

   #pragma omp for nowait schedule(static)
      for (ii=0; ii<N; ii++)
         c[ii] = a[ii]*a[ii];

   #pragma omp for nowait schedule(static)
      for (ii=0; ii<N; ii++)
         d[ii] = sqrt(c[ii]);

}
```

More on Synchronization: Barriers

```
#pragma omp parallel private(id)
{
   id = omp_get_thread_num();
   a[id] = do_some_computing[id];

   #pragma barrier

   #pragma omp for
      for (ii=0; ii<N; ii++)
         b[ii] = do_some_more(a,ii);

}
```

threads synchronize here

More on Synchronization: Locks

- Functions to work with locks:
  - `omp_init_lock() / omp_init_nest_lock()`
  - `omp_destroy_lock() / omp_destroy_nest_lock()`
  - `omp_set_lock() / omp_set_nest_lock()`
  - `omp_unset_lock() / omp_unset_nest_lock()`
  - `omp_test_lock() / omp_test_nest_lock()`
- Normal locks:
  - Can only be set once. If the same thread calls the set function a second time, it will fail.
- Nested locks:
  - Can be set multiple times by the same thread only.
  - Carries a count, so it has to be unset the same number of times before another thread can take the lock.

More on Synchronization: Locks

```
omp_lock_t lck;
int id;

omp_init_lock(&lck);
#pragma omp parallel private(id)
{
   id = omp_get_thread_num();
   omp_set_lock(&lck);
   printf("My thread id is %d.\n", id);
   omp_unset_lock(&lck);
   while (! omp_test_lock(&lck)) {
      skip(id);
   }
   work(id);
   omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Only one thread at the time can do this statement, other threads wait until their turn.

The function skip() is called while the lock is not available. Once it is, work() can be called.

More on Synchronization: Locks

```
typedef struct {
      int a,b;
      omp_nest_lock_t lck;
} pair;

void main(pair *p) {
  #pragma omp parallel sections
  {
    #pragma omp section
      incr_pair(p, a, b);
    #pragma omp section
      incr_b(p, b);
  }
}
```

Always called from within a lock.

```
void incr_a(pair *p, int a) {
  p->a += a;
}

void incr_b(pair *p, int b) {
  omp_set_nest_lock(&p->lck);
  p->b += b;
  omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p,
               int a, int b) {
  omp_set_nest_lock(&p->lck);
  incr_a(p, a);
  incr_b(p, b);
  omp_unset_nest_lock(&p->lck);
}
```

Sometimes called from within a locked region, sometimes not: need to lock, but must be able to lock twice.

OpenMP Functions

- Related to thread count:
  - `omp_set_num_threads(n)`
    - Overrules OMP_NUM_THREADS environment variable
  - `omp_get_num_threads()`
    - How many threads are in the current team?
  - `omp_get_max_threads()`
    - How many threads could potentially be generated in the next PARALLEL section?
  - `omp_get_thread_limit()`
    - How many threads can this program make, in total?
    - Value is set through OMP_THREAD_LIMIT environment variable
  - `omp_get_thread_num()`
    - What is the ID of the current thread?

OpenMP Functions

- Related to PARALLEL Directive:
  - `omp_in_parallel()`
    - Is this code executed in parallel?
- Related to FOR Directive:
  - `omp_set_schedule(kind,chunksize)`
    - Overrules OMP_SCHEDULE environment variable
    - Changes the default scheduling method
  - `omp_get_schedule(&kind,&chunksize)`
    - Returns the scheduling method used in #pragma omp for schedule(runtime)
- Also:
  - `omp_get_num_procs()`
    - How many processors are available to this program?

OpenMP Functions

- Functions to time your code:
  - ANSI C function `clock()` measures processor time used
    - This includes time for all processors!
  - ANSI C function `time()` has a 1s resolution
    - Usually not good enough for timing code.
  - `omp_get_wtime()`
    - Returns the wall time passed (sec) since some point in the past.
    - Could be different for each thread (?)
  - `omp_get_wtick()`
    - Returns the precision of the timer used by `omp_get_wtime()`.

Nested Parallelism

Nested Parallel Programs

- A PARALLEL section within a PARALLEL section
- Nested parallelism is off by default:
  - the inner PARALLEL section gets only 1 thread
- Turn on nested parallelism by:
  - using `omp_set_nested(1)`
  - setting the OMP_NESTED environment variable to "TRUE"
- Probing functions:
  - `omp_get_nested()`
  - `omp_get_level()`
  - `omp_get_ancestor_thread_num(level)`
  - `omp_get_team_size(level)`
  - `omp_get_active_level()`

Nested Parallel Program Example

```
void nesting(int n) {
  int i, j;
  #pragma omp parallel
  {
    #pragma omp for
      for (i=0; i<n; i++) {
        #pragma omp parallel
        {
          #pragma omp for
            for (j=0; j < n; j++)
              work(i, j);
        }
      }
  }
}
```

Nested Parallel Program Example

```
void nesting(int n) {
  int i;
  #pragma omp parallel
  {
    #pragma omp for
      for (i=0; i<n; i++) {
        innerloop(i,n);
      }
  }
}
```

```
void innerloop(int i, int n) {
  int j;
  #pragma omp parallel
  {
    #pragma omp for
      for (j=0; j < n; j++)
        work(i, j);
  }
}
```

The Future of OpenMP

- OpenMP 4.0
  - "Topics under consideration include support for accelerators such as GPUs, major enhancements to the tasking model, mechanisms to support error handling and user defined reductions."
- Next ANSI C Standard (C1X) includes multithreading support
  - _Thread_local storage-class specifier
  - ‹threads.h› header:
    - thread creation/management functions
    - mutex
    - condition variable
    - thread-specific storage functionality
  - _Atomic type qualifier and ‹stdatomic.h›