A Guide To OpenMP

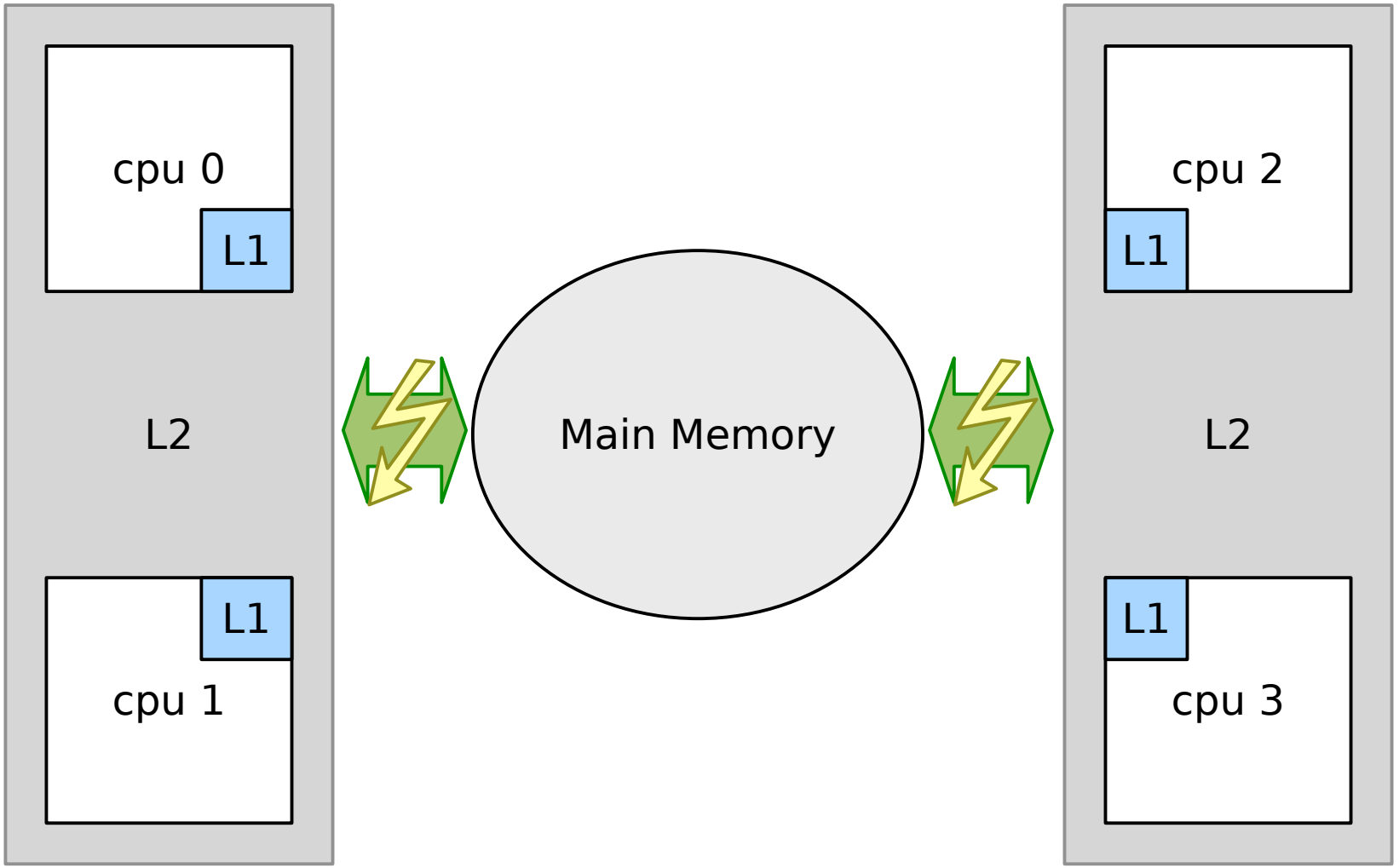
Presented By:
Dr. Barbara Chapman

Created By:
Brett Estrade

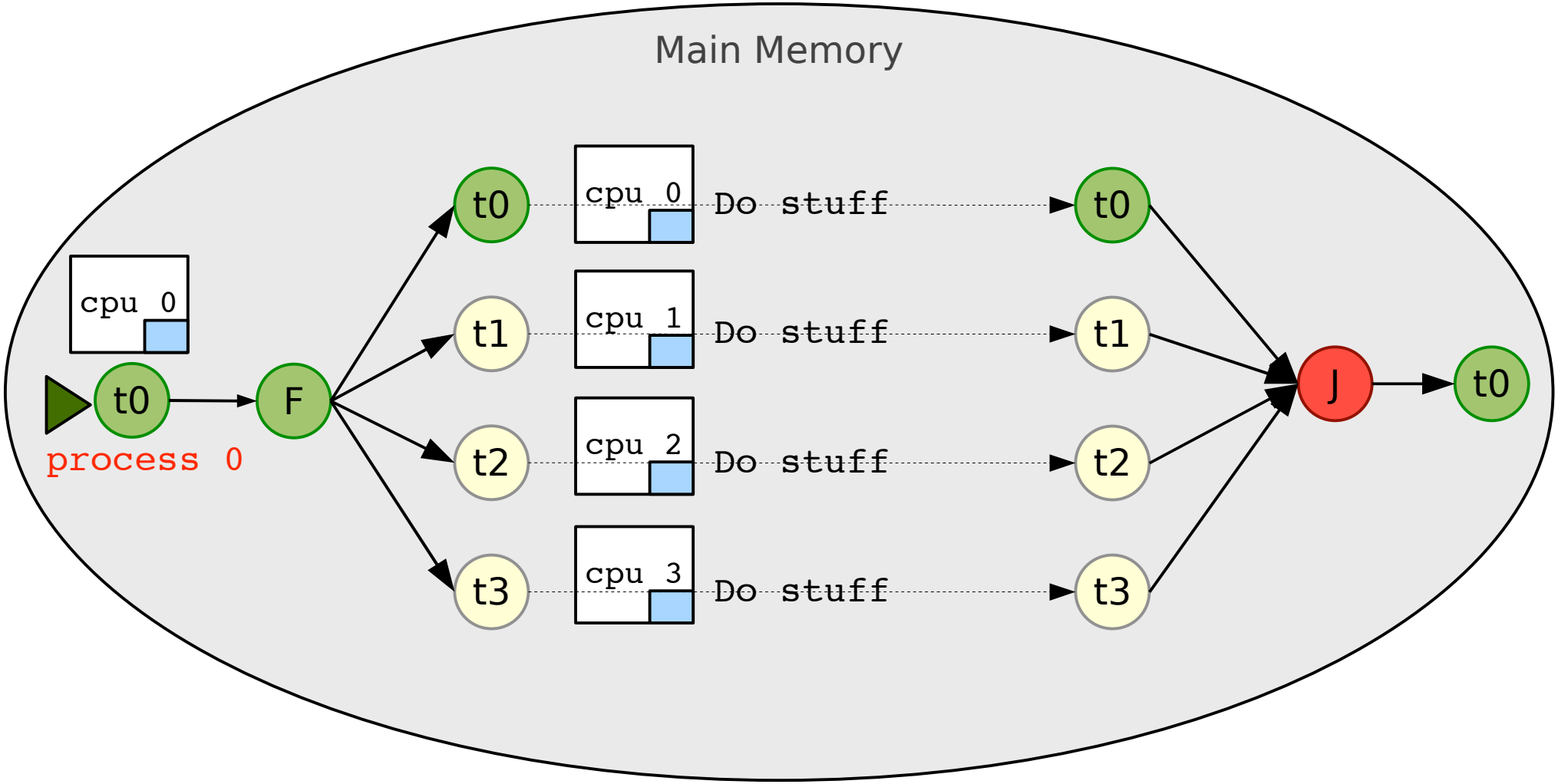
HPCTools Group
University of Houston
Department of Computer Science

http://www.cs.uh.edu/~hpctools

An Example Shared Memory System & Cache Hierarchy



OpenMP's Execution Model – *Fork & Join*

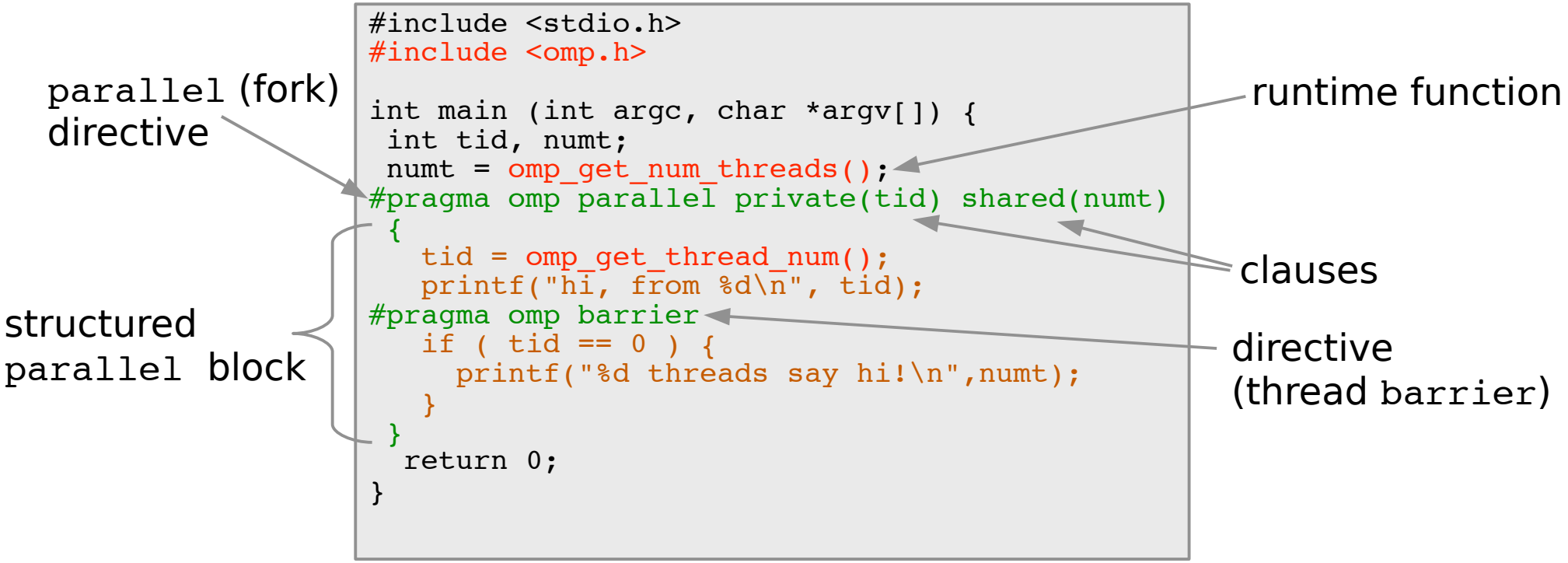


Thread 0 is on CPU 0; at the fork, 3 new threads are created and are distributed to the remaining 3 CPUs.

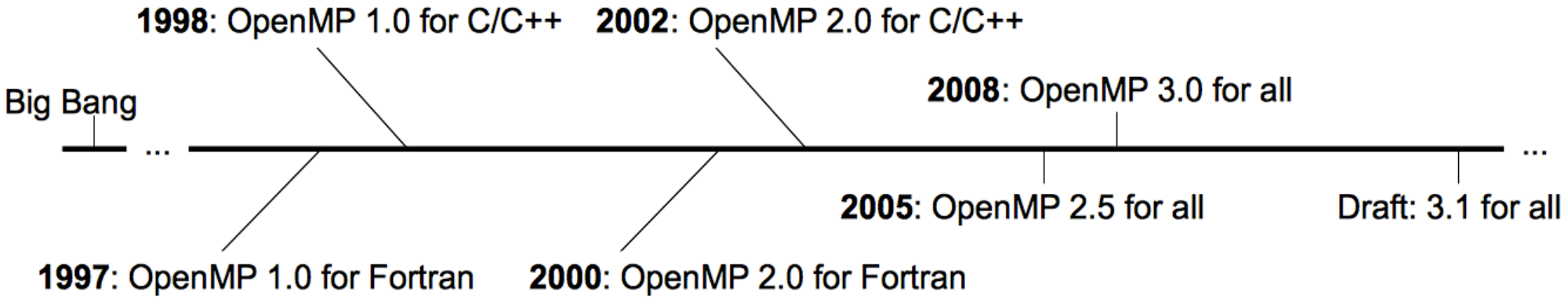
What is OpenMP?

- An industry standard for shared memory parallel programming
  - OpenMP Architecture Review Board
  - AMD, Intel, IBM, HP, Microsoft, Sun/Oracle, Fujitsu, NEC, Texas Instruments, PGI, CAPS; LLNL, ORNL, ANL, NASA, cOMPunity,..
- A set of directives for describing parallelism in an application code
- A user-level API and runtime environment
- A widely supported standard set of parallel programming pragmas with bindings for Fortran, C, & C++
- A *community* of active users & researchers

The Anatomy of an OpenMP Program



The timeline of the OpenMP Standard Specification



- It's **portable**, supported by most C/C++ & Fortran compilers
- Often, much of sequential code can be left untouched
- The development cycle is a friendly one
  - Can be introduced **iteratively** into existing code
  - Correctness can be verified along the way
  - Likewise, performance benefits can be gauged
- Optimizing memory access in the serial program will benefit the threaded version (e.g., false sharing, etc)
- It can be fun to use (immediate gratification)

What Does OpenMP Provide?

- An abstraction above low level thread libraries
- Directives, hidden inside of structured comments
- A *runtime* library that manages execution dynamically
- Additional control via environment variables & a *runtime* API
- Expectations of behavior & sensible defaults
- A promise of *interface* portability;

What Compilers Support OpenMP?

Vendor	Languages	Supported Specification
IBM	C/C++(10.1),Fortran(13.1)	Full 3.0 support
Sun/Oracle	C/C++,Fortran(12.1)	Full 3.0 support
Intel	C/C++,Fortran(11.0)	Full 3.0 support
Portland Group	C/C++,Fortran	Full 3.0 support
Absoft	Fortran(11.0)	Full 2.5 support
Lahey/Fujitsu	C/C++,Fortran(6.2)	Full 2.0 support
PathScale	C/C++,Fortran	Full 2.5 support (based on Open64)
HP	C/C++,Fortran	Full 2.5 support
Cray	C/C++,Fortran	Full 3.0 on Cray XT Series Linux
GNU	C/C++,Fortran	Working towards full 3.0
Microsoft	C/C++,Fortran	Full 2.0

Compiling and Executing Examples

- IBM XL Suite:
  - xlc_r, xlf90, etc

bash

```
% xlc_r -qsmp=omp test.c -o test.x    # compile it
% OMP_NUM_THREADS=4 ./test.x          # execute it
```

- OpenUH:
  - uhcc, uhf90, etc

bash

```
% uhcc -mp test.c -o test.x           # compile it
% OMP_NUM_THREADS=4 ./test.x          # execute it
```

- Contained inside *structured comments*

  C/C++:

```
#pragma omp <directive> <clauses>
```

  Fortran:

```
!$OMP <directive> <clauses>
```

- OpenMP compliant compilers find and parse directives
- Non-compliant *should* safely ignore them as comments
- A *construct* is a directive that affects the enclosing code
- Imperative (standalone) directives exist
- *Clauses* control the behavior of directives
- Order of clauses has no bearing on effect

Summary of OpenMP's Directives

- Forking Threads
  - `parallel`
- Distributing Work
  - `for` (C/C++)
  - `DO` (Fortran)
  - `sections/section`
  - `WORKSHARE` (Fortran)
- Singling Out Threads
  - `single`
  - `Master`
- Mutual Exclusion
  - `critical`
  - `atomic`
- Synchronization
  - `barrier`
  - `flush`
  - `ordered`
  - `taskwait`
- Asynchronous Tasking
  - `task`
- Data Environment
  - `shared`
  - `private`
  - `threadprivate`
  - `reduction`

Useful OpenMP Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_STACKSIZE
- OMP_NESTED
- OMP_THREAD_LIMIT
- OMP_MAX_ACTIVE_LEVELS
- OMP_WAIT_POLICY

Execution environment routines; e.g.,

- `omp_{set,get}_num_threads`
- `omp_{set,get}_dynamic`
- Each envar has a corresponding get/set

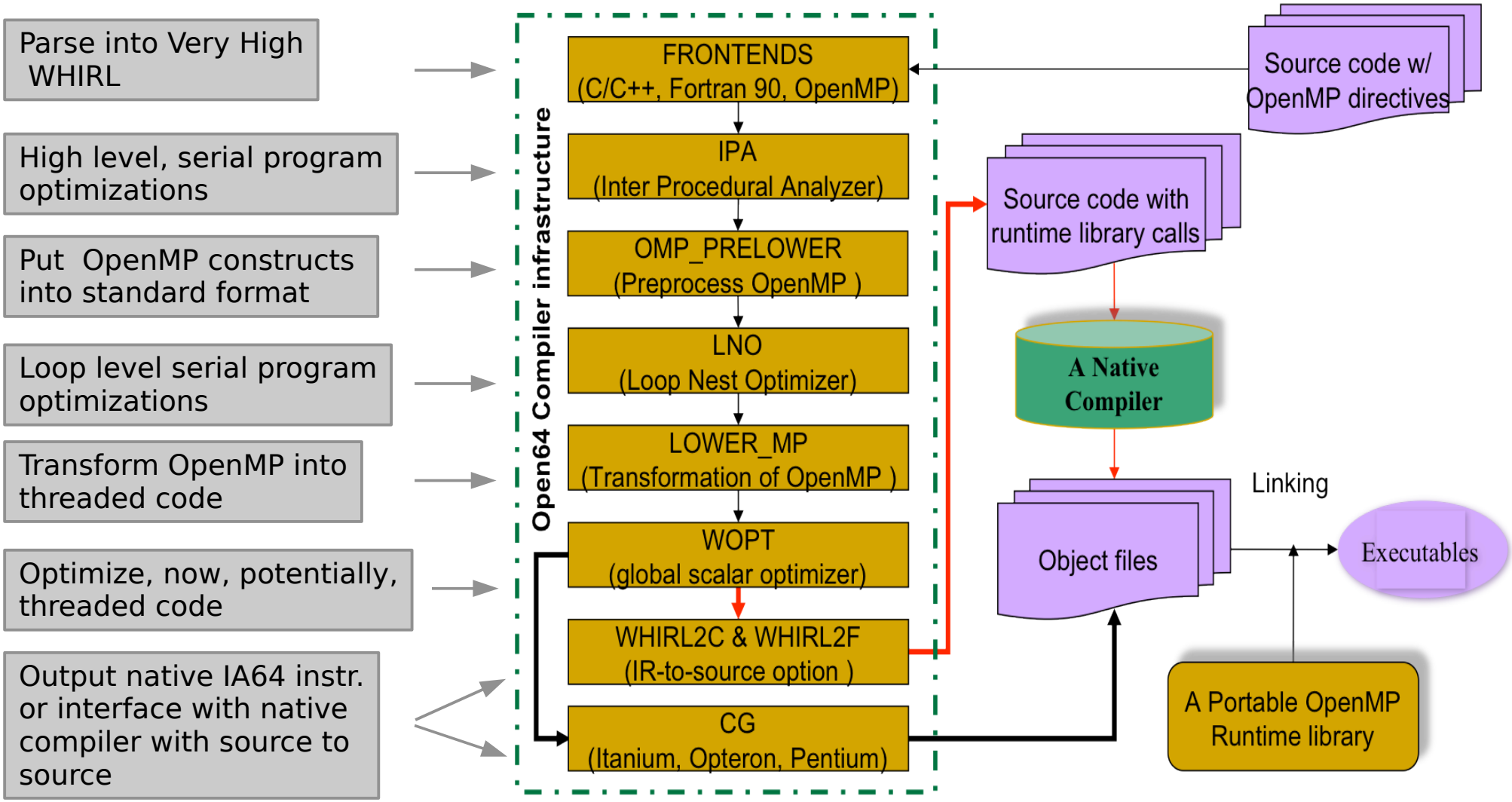
Locking routines (*generalized mutual exclusion*); e.g.,

- `omp_{init,set,test,unset,destroy}_lock`
- `omp_{...}_nest_lock`

Timing routines; e.g.,

- `omp_get_wtime`
- `omp_get_wtick`

How Is an OpenMP Program Compiled? Here's How OpenUH does it.



Liao, et. al.: http://www2.cs.uh.edu/~copper/openuh.pdf

HPCTools

What Does the Transformed Code Look like?

- Intermediate code,"W2C" - WHIRL to C
  - `uhcc -mp -gnu3 -CLIST:emit_nested_pu simple.c`
  - http://www2.cs.uh.edu/~estrabd/OpenMP/simple/

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int my_id;
#pragma omp parallel default(none) private(my_id)
  {
    my_id = omp_get_thread_num();
    printf("hello from %d\n",my_id);
  }
  return 0;
}
```

The original `main()`

```
static void __omprg_main_1(__ompv_gtid_a, __ompv_slink_a)
  _INT32 __ompv_gtid_a;
  _UINT64 __ompv_slink_a;
{

  register _INT32 _w2c___comma;
  _UINT64 _temp___slink_sym0;
  _INT32 __ompv_temp_gtid;
  _INT32 __mplocal_my_id;

  /*Begin_of_nested_Program_Unit(s)*/

  _temp___slink_sym0 = __ompv_slink_a;
  __ompv_temp_gtid = __ompv_gtid_a;
  _w2c___comma = omp_get_thread_num();
  __mplocal_my_id = _w2c___comma;
  printf("hello from %d\n", __mplocal_my_id);
  return;
} /* __omprg_main_1 */
```

parallel region in `main` is outlined to `__omprg_main_1()`

- The "*runtime*" manages the multi-threaded execution:
  - It's used by the resulting executable OpenMP program
  - It's what spawns threads (e.g., calls pthreads)
  - It's what manages shared & private memory
  - It's what distributes (shares) work among threads
  - It's what synchronizes threads & tasks
  - It's what reduces variables and keeps `lastprivate`
  - It's what is influenced by envars & the user level API
- Doxygen docs of OpenUH's OpenMP RTL, `libopenmp`
  - http://www2.cs.uh.edu/~estrabd/OpenUH/r593/html-libopenmp/
- The Doxygen call graph for __omp_fork in libopenmp/threads.c
  - `__omp_fork(...) call graph`

The new main()


```
extern _INT32 main() {
  register _INT32 _w2c___ompv_ok_to_fork;
  register _UINT64 _w2c_reg3;
  register _INT32 _w2c___comma;
  _INT32 my_id;
  _INT32 __ompv_gtid_s1;

  /*Begin_of_nested_PU(s)*/

  _w2c___ompv_ok_to_fork = 1;
  if(_w2c___ompv_ok_to_fork)
  {
    _w2c___ompv_ok_to_fork = __ompc_can_fork();
  }
  if(_w2c___ompv_ok_to_fork)
  {
    __ompc_fork(0, &__omprg_main_1, _w2c_reg3);
  }
  else
  {
    __ompv_gtid_s1 = __ompc_get_local_thread_num();
    __ompc_serialized_parallel();
    _w2c___comma = omp_get_thread_num();
    my_id = _w2c___comma;
    printf("hello from %d\n", my_id);
    __ompc_end_serialized_parallel();
  }
  return 0;
} /* main */
```

calls RTL fork and passes function pointer to outlined main()

__omprg_main_1's frame pointer

serial version

Nobody wants to code like this, so let the compiler and runtime do most all this tedious work!

Programming with OpenMP 3.0

The `parallel` Construct

- Where the “fork” occurs (e.g., `__ompc_fork(...)`)
- Encloses all other OpenMP constructs & directives
- This construct accepts the following clauses: `if, num_threads, private, firstprivate, shared, default, copyin, reduction`
- Can call functions that contain “orphan” constructs
  - Statically outside of parallel, but dynamically inside during runtime
- Can be nested

A Simple OpenMP Example

C/C++

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

int main (int argc, char *argv[]) {
 int tid, numt;
 numt = omp_get_num_threads();
#pragma omp parallel private(tid) shared(numt)
 {
   tid = omp_get_thread_num();
   printf("hi, from %d\n", tid);
#pragma omp barrier
   if ( tid == 0 ) {
     printf("%d threads say hi!\n",numt);
   }
 }
  return 0;
}
```

get number of threads

fork

get thread id

wait for all threads

join (implicit barrier, all wait)

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

Note, thread order not guaranteed!

The Fortran Version

C/C++

```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
 int tid, numt;
 numt = omp_get_num_threads();
#pragma omp parallel private(tid) shared(numt)
 {
   tid = omp_get_thread_num();
   printf("hi, from %d\n", tid);
#pragma omp barrier
   if ( tid == 0 ) {
     printf("%d threads say hi!\n",numt);
   }
 }
  return 0;
}
```

F90

```
      program hello90
      use omp_lib
      integer:: tid, numt
      numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
      tid = omp_get_thread_num()
      write (*,*) 'hi, from', tid
!$omp barrier
      if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
      end if
!$omp end parallel
      end program
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

Note, thread order not guaranteed!

Now, Just the Parallelized Code

C/C++

```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
 int tid, numt;
 numt = omp_get_num_threads();
#pragma omp parallel private(tid) shared(numt)
 {
   tid = omp_get_thread_num();
   printf("hi, from %d\n", tid);
#pragma omp barrier
   if ( tid == 0 ) {
     printf("%d threads say hi!\n",numt);
   }
 }
  return 0;
```

F90

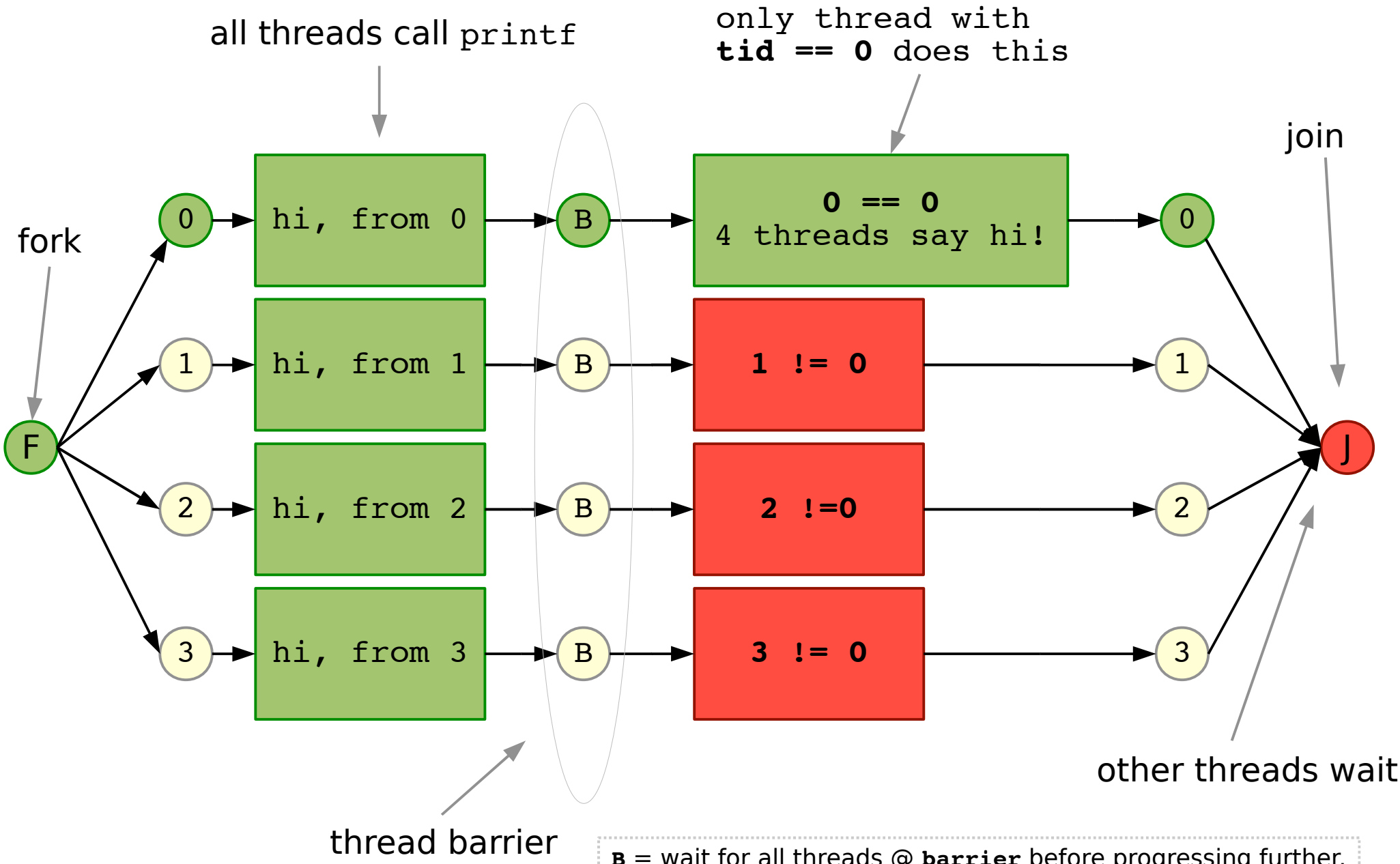
```f90
      program hello90
      use omp_lib
      integer:: tid, numt
      numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
      tid = omp_get_thread_num()
      write (*,*) 'hi, from', tid
!$omp barrier
      if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
      end if
!$omp end parallel
      end program
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

Note, thread order not guaranteed!

Trace of The Execution



B = wait for all threads @ **barrier** before progressing further.

- The "`if`" clause contains a conditional expression.
- If TRUE, forking occurs, else it doesn't

```
int n = some_func();
#pragma omp parallel if(n>5)
  {
    … do stuff in parallel
  }
```

- The "`num_threads`" clause is another way to control the number of threads active in a `parallel` contruct

```
int n = some_func();
#pragma omp parallel num_threads(n)
  {
    … do stuff in parallel
  }
```

- `default([shared]|none|private)`
- `shared(list,)` - supported by `parallel` construct only
- `private(list,)`
- `firstprivate(list,)`
- `lastprivate(list,)` - supported by loop & `sections` constructs only
- `reduction(<op>:list,)`
- `copyprivate(list,)` - supported by `single` construct only
- `threadprivate` – a standalone directive, **not** a clause

```
#pragma omp threadprivate(list,)
```

```
!$omp threadprivate(list,)
```

- `copyin(list,)` - supported by `parallel` construct only

private & shared in that Simple OpenMP Example

C/C++

```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
 int tid, numt;
 numt = omp_get_num_threads();
#pragma omp parallel private(tid) shared(numt)
 {
   tid = omp_get_thread_num();
   printf("hi, from %d\n", tid);
#pragma omp barrier
   if ( tid == 0 ) {
     printf("%d threads say hi!\n",numt);
   }
 }
  return 0;
}
```

F90

```f90
      program hello90
      use omp_lib
      integer:: tid, numt
      numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
      tid = omp_get_thread_num()
      write (*,*) 'hi, from', tid
!$omp barrier
      if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
      end if
!$omp end parallel
      end program
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

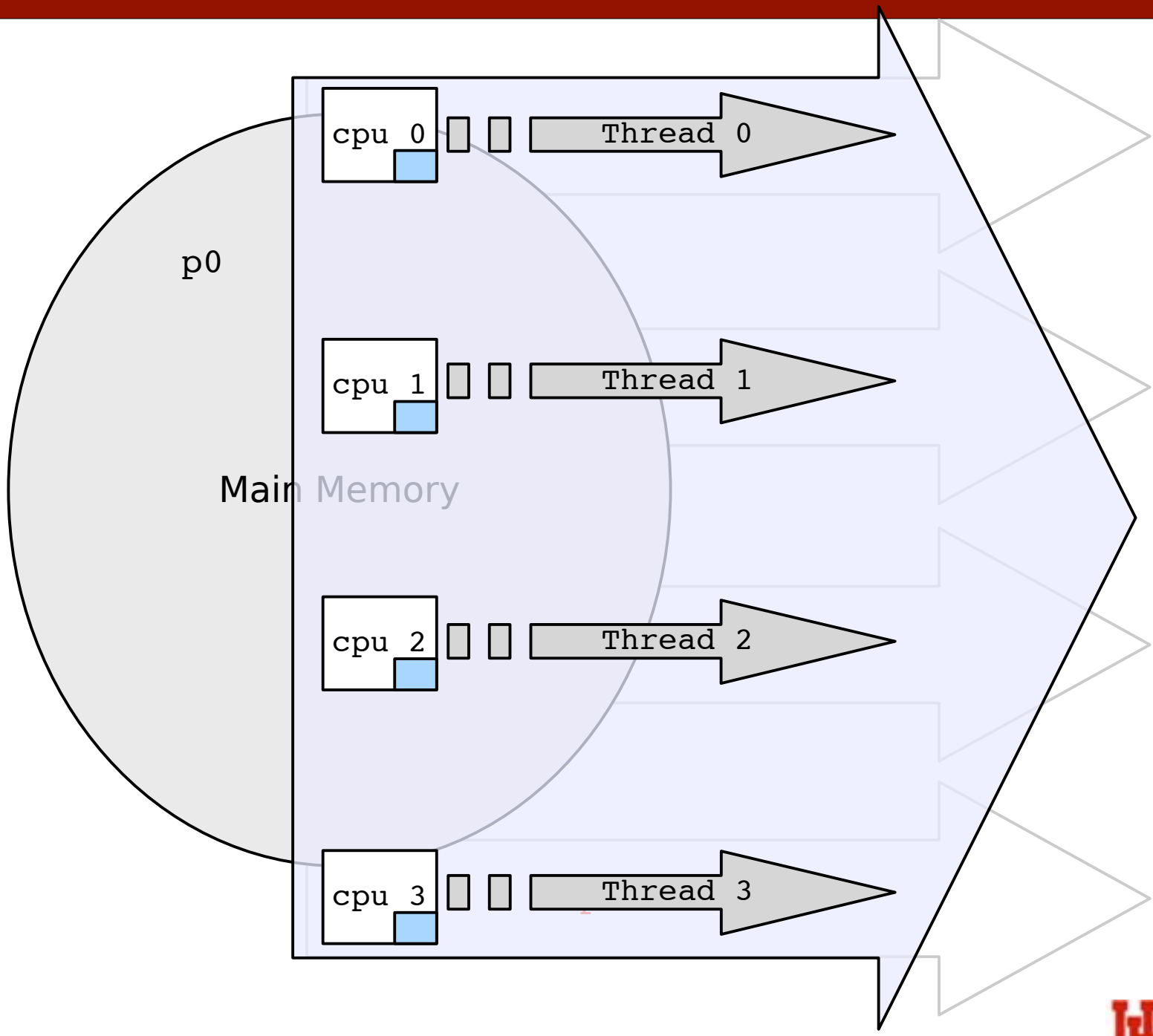
Note, thread order not guaranteed!

- OpenMP uses a “relaxed consistency” model
  - Threads may temporarily “see” different values for the same shared variable
  - Cores may have out of date values in their cache
- Most constructs imply a “`flush`” of each thread's cache
- Treated as a memory “fence” by compilers when it comes to reordering operations
- OpenMP provides an explicit flush directive

```
#pragma flush (list,)
```

```
!$OMP FLUSH(list,)
```

Multiple Threads May Have Copies of Shared Variables in Cache



An explicit `barrier` in that Simple OpenMP Example

C/C++

```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
 int tid, numt;
 numt = omp_get_num_threads();
#pragma omp parallel private(tid) shared(numt)
 {
   tid = omp_get_thread_num();
   printf("hi, from %d\n", tid);
#pragma omp barrier
   if ( tid == 0 ) {
     printf("%d threads say hi!\n",numt);
   }
 }
  return 0;
}
```

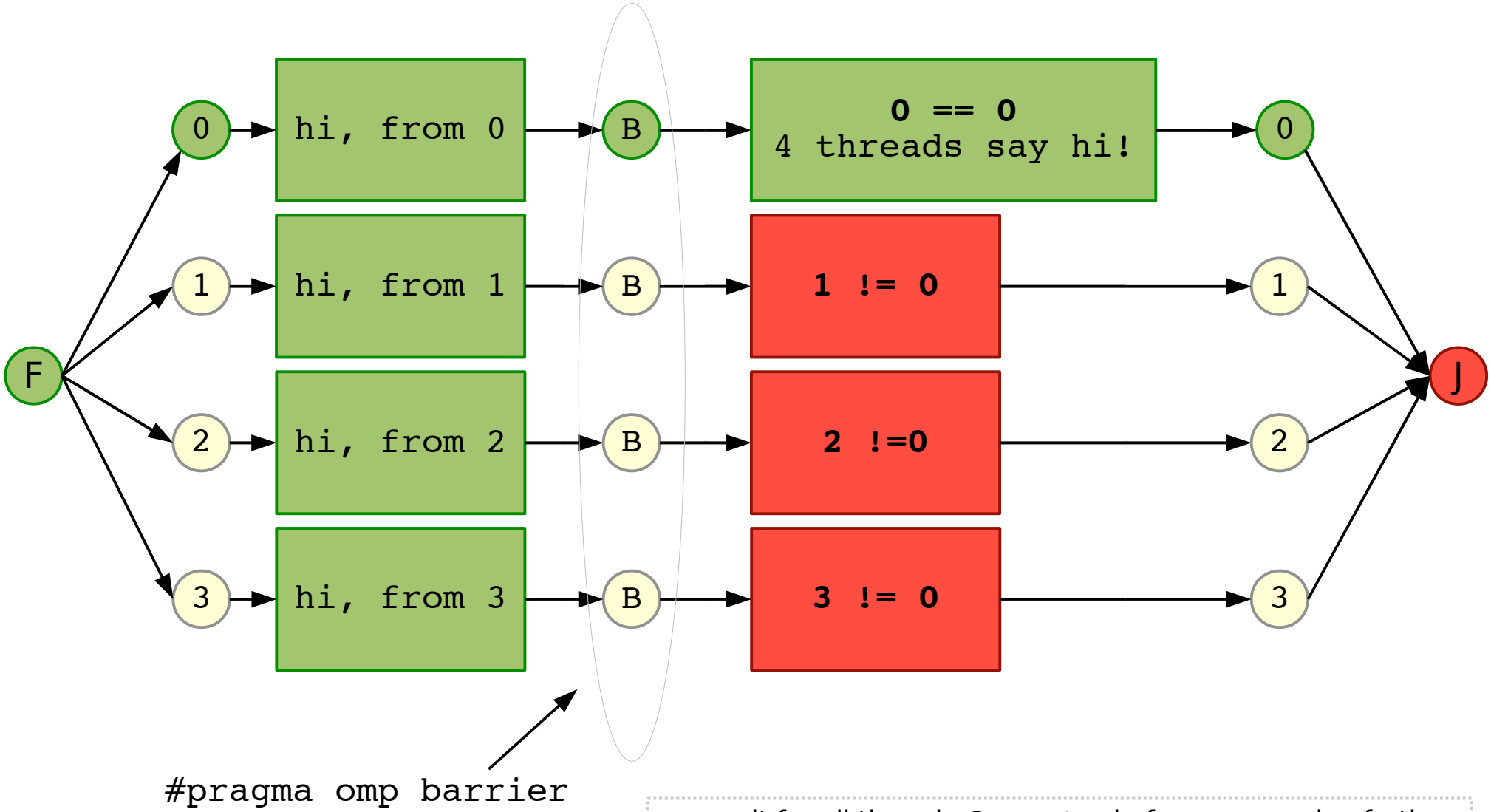
F90

```
      program hello90
      use omp_lib
      integer:: tid, numt
      numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
      tid = omp_get_thread_num()
      write (*,*) 'hi, from', tid
!$omp barrier
      if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
      end if
!$omp end parallel
      end program
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
<barrier>
4 threads say hi!
```

Trace of The Execution



B = wait for all threads @ **barrier** before progressing further.

A reduction Example

C/C++

```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
 int t, i;
 i = 0;
#pragma omp parallel private(t) reduction(+,i)
 {
   t = omp_get_thread_num();
   i = t + 1;
   printf("hi, from %d\n", t);
#pragma omp barrier
   if ( t == 0 ) {
     int numt = omp_get_num_threads();
     printf("%d threads say hi!\n",numt);
   }
 }
  printf("i is reduced to %d\n",i);
  return 0;
}
```

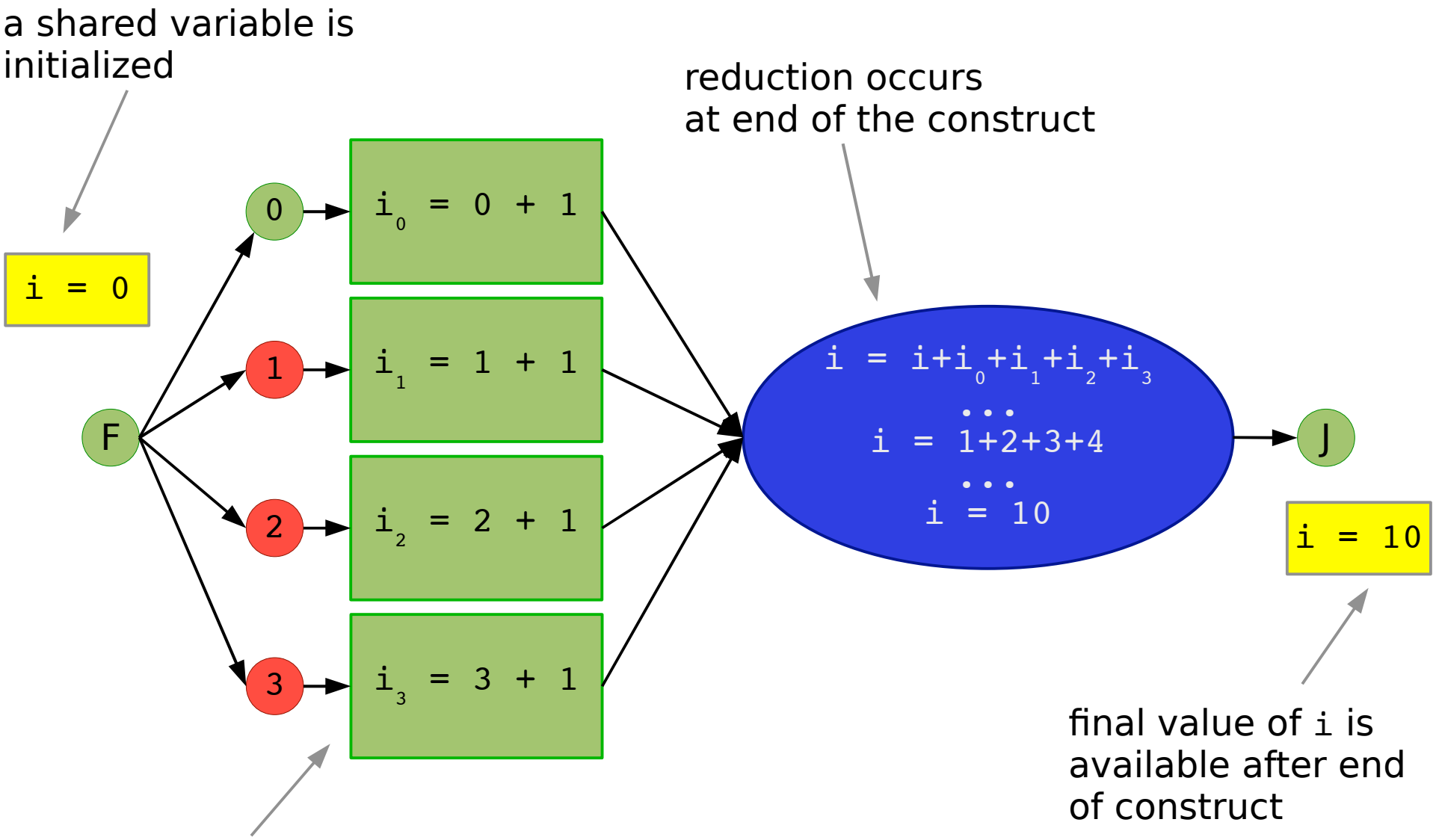
F90

```
      program hello90
      use omp_lib
      integer:: t, i, numt
      i = 0
!$omp parallel private(t) reduction(+:i)
      t = omp_get_thread_num()
      i = t + 1;
      write (*,*) 'hi, from', t
!$omp barrier
      if ( t == 0 ) then
        numt = omp_get_num_threads()
        write (*,*) numt,'threads say hi!'
      end if
!$omp end parallel
      write (*,*) 'i is reduced to ', i
      end program
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
i is reduced to 10
```

Trace of a variable reduction



Valid Operations for the `reduction` Clause

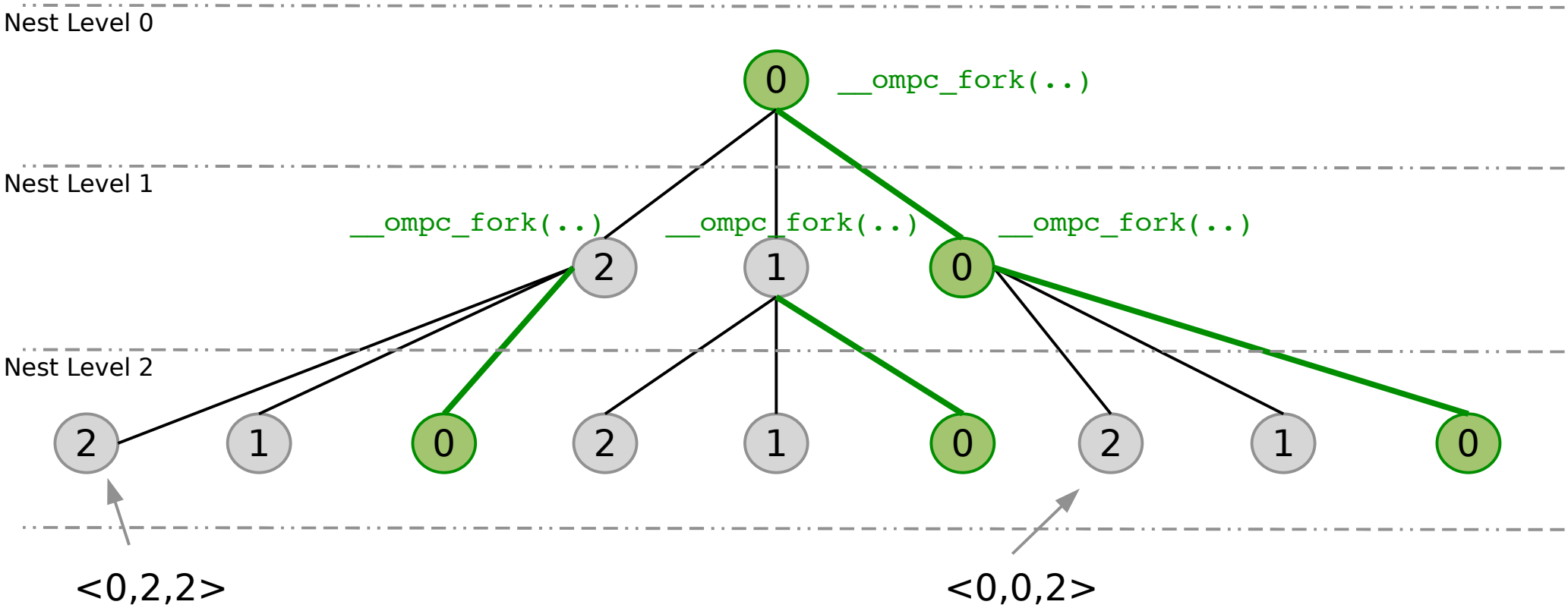
- Reduction operations in C/C++:
  - Arithmetic: + - *
  - Bitwise: & ^ |
  - Logical: && ||
- Reduction operations in Fortran
  - Equivalent arithmetic, bitwise, and logical operations
  - `min`, `max`
- User defined reductions (UDR) is an area of current research
- Note: initialized value matters!

Nested `parallel` Constructs

- Can be nested, but specification makes it optional
  - `OMP_NESTED={true,false}`
  - `OMP_MAX_ACTIVE_LEVELS={1,2,..}`
  - `omp_{get,set}_nested()`
  - `omp_get_level()`
  - `omp_get_ancestor_thread_num(level)`
- Each encountering thread becomes the master of the newly forked team
- Each subteam is numbered 0 through N-1
- Useful, but still incurs `parallel` overheads

The Uniqueness of Thread Numbers in Nesting

Thread numbers are not unique; paths to each thread are.

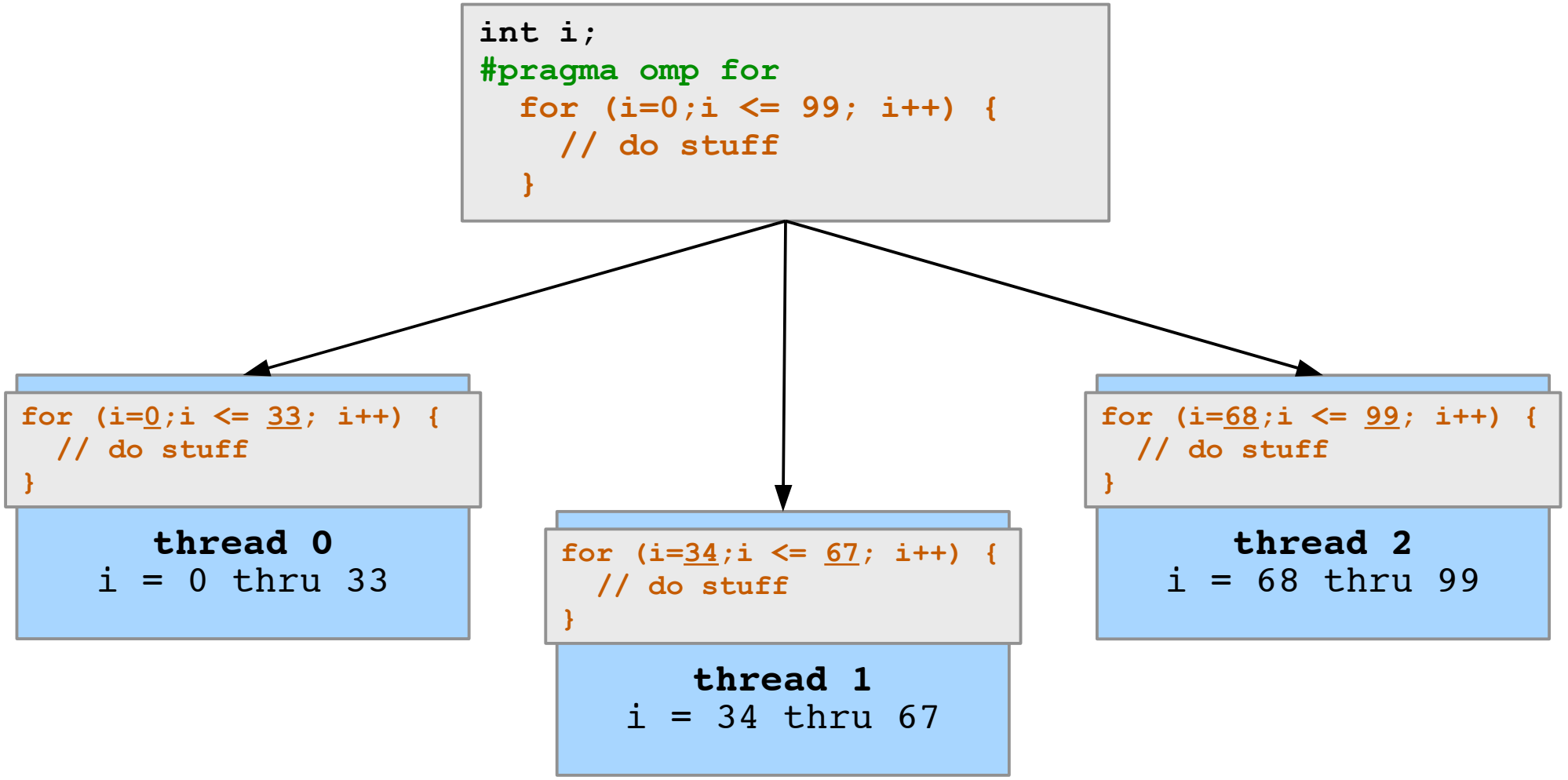


Because of the tree structure, each thread can be uniquely identified by its full path from the root of its sub-tree;

This path tuple can be calculated in **O***(level)* using `omp_get_level` and `omp_get_ancestor_thread_num` in combination.

- Threads must share in the work of program
- OpenMP provides “work sharing” constructs
- Specifies the work to be distributed and how
- These constructs include:
  - loops (`for, DO`)
  - `sections`
  - `WORKSHARE` (Fortran only)
  - `single, master`

OpenMP Parallelizes Loops by Distributing Iterations to Each Thread



Parallelizing Loops - C/C++


```
#include <stdio.h>
#include <omp.h>
#define N 100

int main(void)
{
 float a[N], b[N], c[N];
 int i;
 omp_set_dynamic(0);            // ensures use of all available threads
 omp_set_num_threads(20);       // sets number of all available threads to 20
/* Initialize arrays a and b. */
 for (i = 0; i < N; i++)
   {
     a[i] = i * 1.0;
     b[i] = i * 2.0;
   }
/* Compute values of array c in parallel. */

#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp for [nowait]
   for (i = 0; i < N; i++)
     c[i] = a[i] + b[i];
   }
 printf ("%f\n", c[10]);
}
```

http://developers.sun.com/solaris/articles/studio_openmp.html

Parallelizing Loops - C/C++


```
{
     a[i] = i * 1.0;
     b[i] = i * 2.0;
   }
/* Compute values o#include <stdio.h>
#include <omp.h>
#define N 100

int main(void)
{
 float a[N], b[N], c[N];
 int i;
 omp_set_dynamic(0);           // ensures use of all available threads
 omp_set_num_threads(20);      // sets number of all available threads to 20
/* Initialize arrays a and b. */
 for (i = 0; i < N; i++)
   f array c in parallel. */

#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp for [nowait]
   for (i = 0; i < N; i++)
     c[i] = a[i] + b[i];
  }
 printf ("%f\n", c[10]);
}
```

"nowait" is optional

http://developers.sun.com/solaris/articles/studio_openmp.html

Parallelizing Loops - Fortran


```
      PROGRAM VECTOR_ADD
      USE OMP_LIB
      PARAMETER (N=100)
      INTEGER N, I
      REAL A(N), B(N), C(N)
      CALL MP_SET_DYNAMIC (.FALSE.)  !ensures use of all available threads
      CALL OMP_SET_NUM_THREADS (20)   !sets number of available threads to 20
! Initialize arrays A and B.
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = I * 2.0
      ENDDO
! Compute values of array C in parallel.
!$OMP PARALLEL SHARED(A, B, C), PRIVATE(I)
!$OMP DO
      DO I = 1, N
        C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO [nowait]
      ! ... some more instructions
!$OMP END PARALLEL
      PRINT *, C(10)
      END
```

http://developers.sun.com/solaris/articles/studio_openmp.html

Parallelizing Loops - Fortran


```
      PROGRAM VECTOR_ADD
      USE OMP_LIB
      PARAMETER (N=100)
      INTEGER N, I
      REAL A(N), B(N), C(N)
      CALL MP_SET_DYNAMIC (.FALSE.)  !ensures use of all available threads
      CALL OMP_SET_NUM_THREADS (20)   !sets number of available threads to 20
! Initialize arrays A and B.
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = I * 2.0
      ENDDO
! Compute values of array C in parallel.
!$OMP PARALLEL SHARED(A, B, C), PRIVATE(I)
!$OMP DO
      DO I = 1, N
        C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO [nowait]
      ! ... some more instructions
!$OMP END PARALLEL
      PRINT *, C(10)
      END
```

"nowait" is optional & added to the closing directive

http://developers.sun.com/solaris/articles/studio_openmp.html

- Scheduling refers to how iterations are assigned to a particular thread;
- There are 5 types:
  - *static* – each thread is able to calculate its chunk
  - *dynamic* – first come, first serve managed by runtime
  - *guided* – decreasing chunk sizes, increasing work
  - *auto* – determined automatically by compiler or runtime
  - *runtime* – defined by `OMP_SCHEDULE` or `omp_set_schedule`
- Limitations
  - only one schedule type may be used for a given loop
  - the chunk size applies to *all* threads

Parallel Loop Scheduling - Example

Fortran

```
!$OMP PARALLEL SHARED(A, B, C) PRIVATE(I)
!$OMP DO SCHEDULE (DYNAMIC,4)
      DO I = 1, N
        C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO [nowait]
!$OMP END PARALLEL
```

schedule chunk size

C/C++

```
#pragma omp parallel shared(a, b, c) private(i)
 {
#pragma omp for schedule (guided,4) [nowait]
   for (i = 0; i < N; i++)
     c[i] = a[i] + b[i];
 }
```

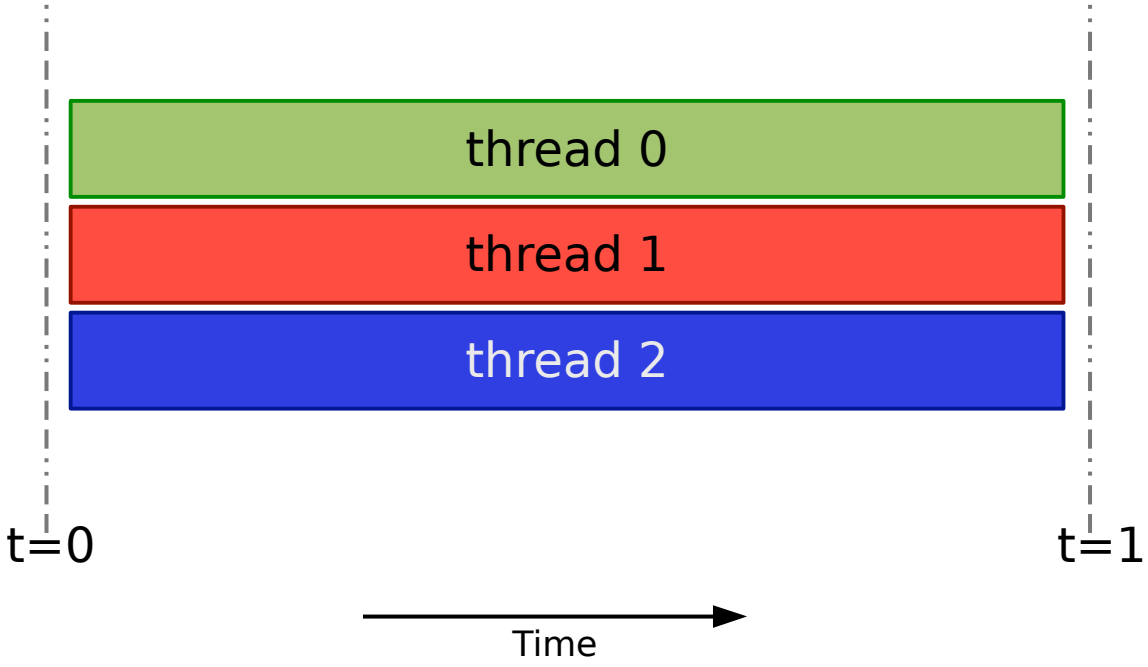
A section Construct Example


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

int square(int n){
  return n*n;
}
int main(void){
 int x, y, z, xs, ys, zs;
 omp_set_dynamic(0);
 omp_set_num_threads(3);
 x = 2; y = 3; z = 5;

#pragma omp parallel shared(xs,ys,zs) firstprivate (x, y, z)
 {
#pragma omp sections
   {
#pragma omp section
     { xs = square(x);
       printf ("id = %d, xs = %d\n", omp_get_thread_num(), xs);
     }
#pragma omp section
     { ys = square(y);
       printf ("id = %d, ys = %d\n", omp_get_thread_num(), ys);
     }
#pragma omp section
     { zs = square(z);
       printf ("id = %d, zs = %d\n", omp_get_thread_num(), zs);
     }
   }
 }
  return 0;
}
```

http://developers.sun.com/solaris/articles/studio_openmp.html

A section Construct Example

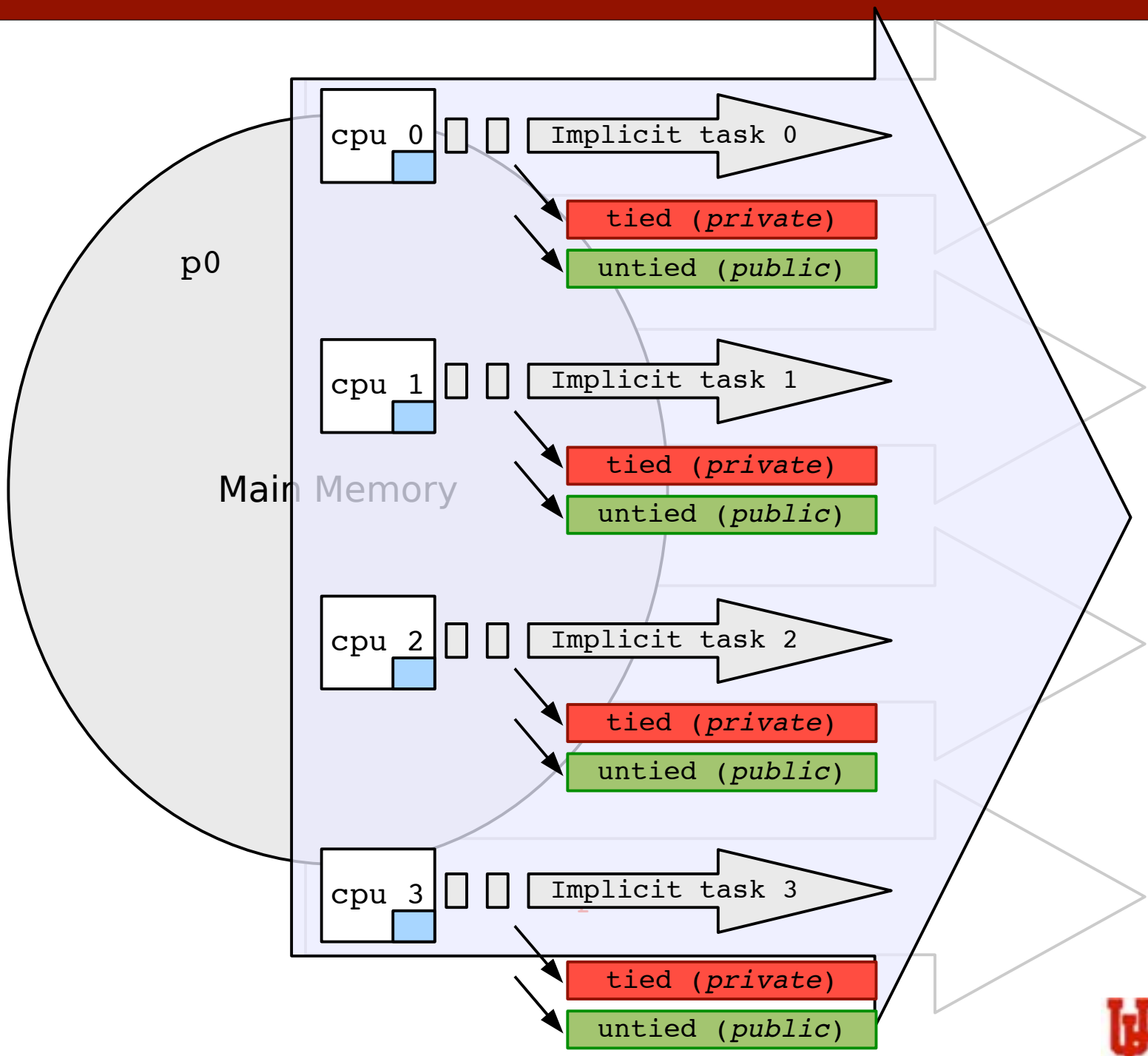

```
#pragma omp sections
  {
#pragma omp section
      { xs = square(x);
        printf ("id = %d, xs = %d\n", omp_get_thread_num(), xs);
      }
#pragma omp section
      { ys = square(y);
        printf ("id = %d, ys = %d\n", omp_get_thread_num(), ys);
      }
#pragma omp section
      { zs = square(z);
        printf ("id = %d, zs = %d\n", omp_get_thread_num(), zs);
      }
  }
```



The `task` Construct

- Tasks were added in 3.0 to handle dynamic and unstructured applications
  - Recursion
  - Tree & graph traversals
- OpenMP's execution model based on threads was redefined
- A thread is considered to be an *implicit* task
- The `task` construct defines singular tasks explicitly
- Less overhead than nested `parallel` regions

Each Thread May Have Both a `tied` & `untied` queue



- Clauses supported are: `if, default, private, firstprivate shared, tied/untied`
- By default, all variables are `firstprivate`
- Tasks can be nested syntactically, but are still asynchronous
- The taskwait directive causes a task to wait until all its children have completed

A `task` Construct C/C++ Example

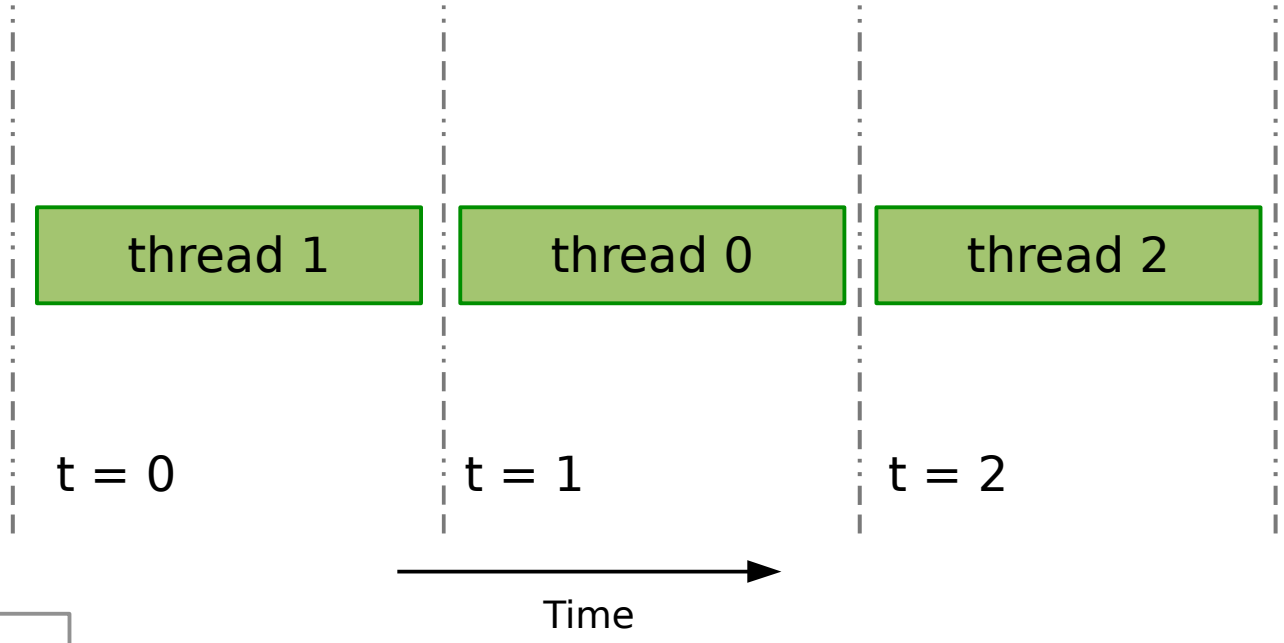

```
struct node {
  struct node *left;
  struct node *right;
};

extern void process(struct node *);

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

Mutual Exclusion: `critical`, `atomic`, and `omp_lock_t`

- Some code must be executed by one thread at a time
- Effectively serializes the threads
- Also called critical sections
- OpenMP provides 3 ways to achieve mutual exclusion
  - The `critical` construct encloses a critical section
  - The `atomic` construct enclose updates to shared variables
  - A low level, general purpose locking mechanism

A `critical` Construct Example


```
#pragma omp parallel shared(c) private(a, b, i)
  {
#pragma omp critical

    { for (i = 0; i < N; i++)
          C[ 0] += a[i] + b[i];
      printf("%f\n",c[0]);
    }
  }
```

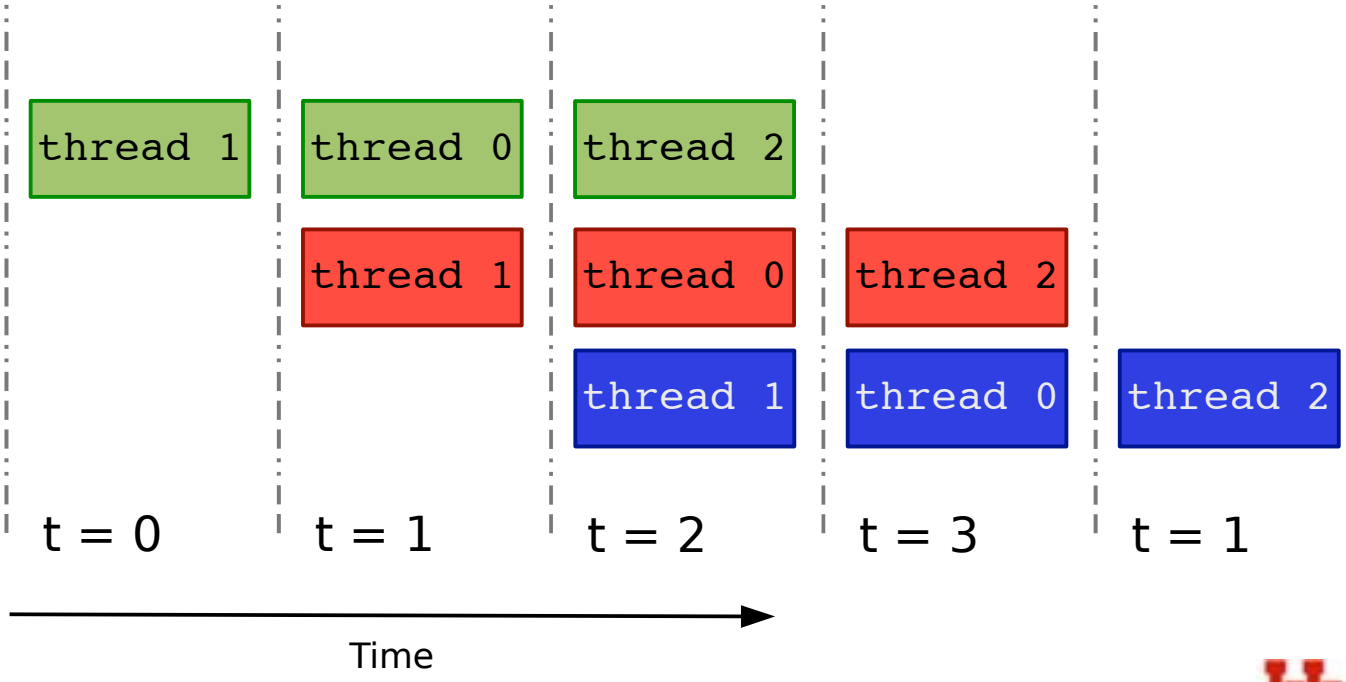


Note:
Encountering thread order not guaranteed!

A Named `critical` Construct Example

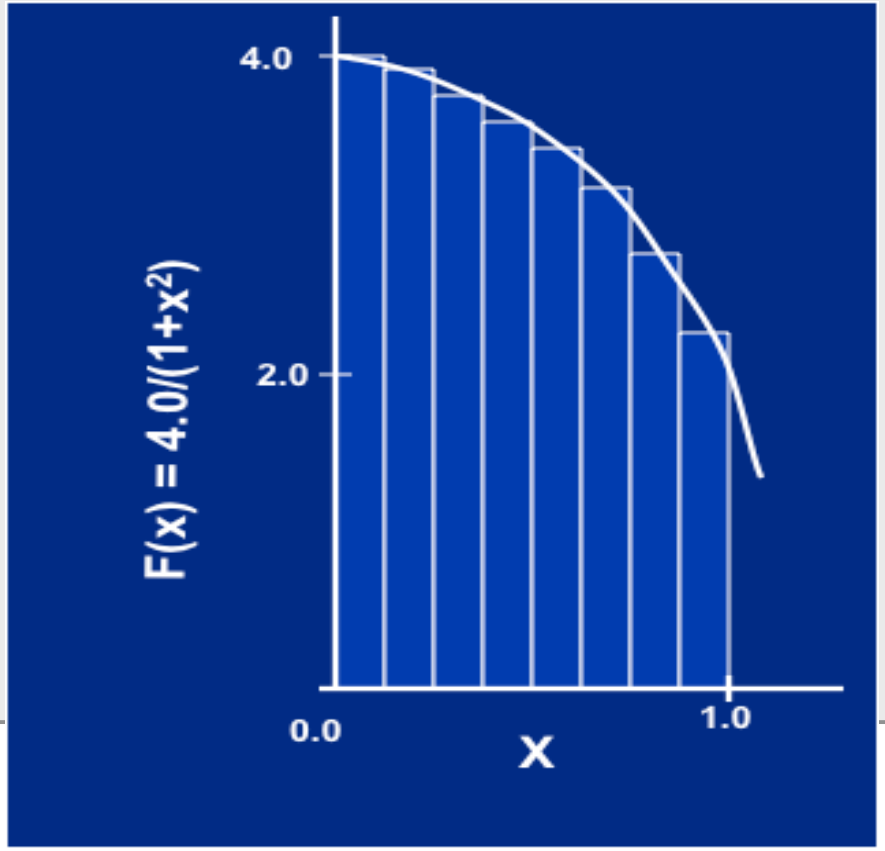

```
#pragma omp critical(a)
     {
       // some code
     }
#pragma omp critical(b)
     {
       // some code
     }
#pragma omp critical(c)
     {
       // some code
     }
```

Note:
Encountering thread order not guaranteed!



A Practical Example – Calculating π


```
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;
}
```



Mathematically, we know:

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

And this can be approximated as a sum of the area of rectangles:

$$\sum_{i=1}^{N} F(x_i)\Delta x \approx \pi$$

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

A Naïve Parallel Program - Calculating π


```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i, id, nthreads;  double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x)
  {
    id = omp_get_thread_num();
#pragma omp single
      nthreads = omp_get_num_threads();
    for (i=id, sum[id]=0.0;i< num_steps; i=i+nthreads){
      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

Can't assume the number of threads requested; use `single` to prevent conflicts.

False sharing of the sum array will affect performance

A Slightly More Efficient Parallel Program - Calculating π


```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i, id, nthreads;  double x, pi, sum;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x, sum)
  {
    id = omp_get_thread_num();
#pragma omp single
      nthreads = 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;
  }
}
```

No array, so no false sharing

Note: this method of combining partial sums doesn't scale very well.

A Simple & Efficient Parallel Program - Calculating π


```
#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 a `critical` construct

i private by default

Note: a parallel program is created without changing any existing code and adding 4 simple lines.

The Final Parallel Program - Calculating π


```
#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 a `critical` construct

i private by default

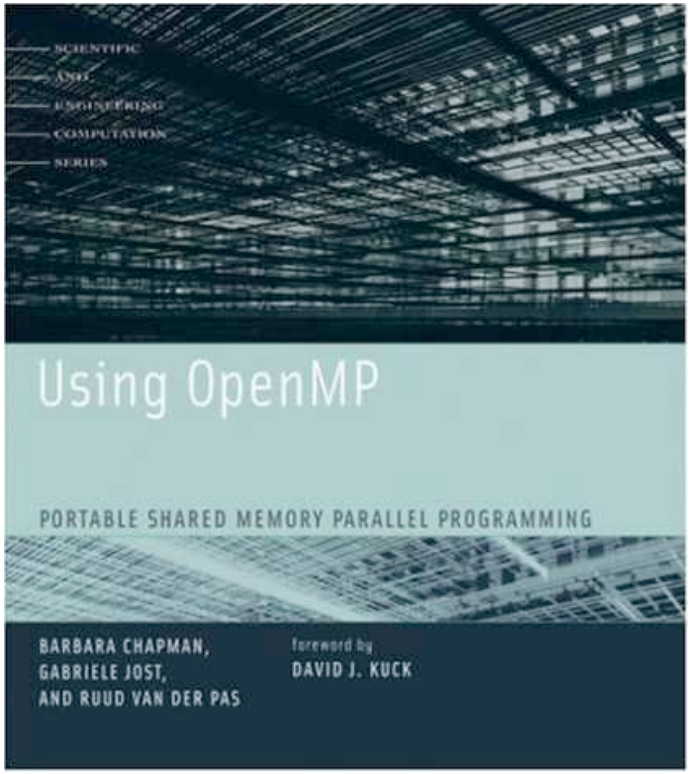
In practice, the number of threads is usually set using the environment variable, `OMP_NUM_THREADS`.

- Minimize `parallel` constructs
- Use *combined* constructs, if it doesn't violate the above
- Minimize shared variables, maximize private
- Minimize barriers, but don't sacrifice safety
- When inserting OpenMP into existing code
  - Use a disciplined, iterative development cycle – test against serial version
  - Use barriers liberally
  - Optimize OpenMP & asynchronize **last**
- When starting from scratch
  - Start with an optimized serial version

- Vendor buy-in and R&D support is as strong as ever
- Must remain relevant
- Active areas of research:
  - Refinement to tasking model (scheduling, etc)
  - User defined reductions (UDRs)
  - Accelerators & heterogeneous environments
  - Error handling
  - Hybrid models
- Scaling issues being addressed:
  - Thousands of threads
  - Data locality
  - More flexible & efficient synchronization

Additional Resources

- http://www.cs.uh.edu/~hpctools
- http://www.compunity.org
- http://www.openmp.org
  - Specification 3.0
  - More resources
- “Using OpenMP”, Chapman, et. al.



Covers through 2.5