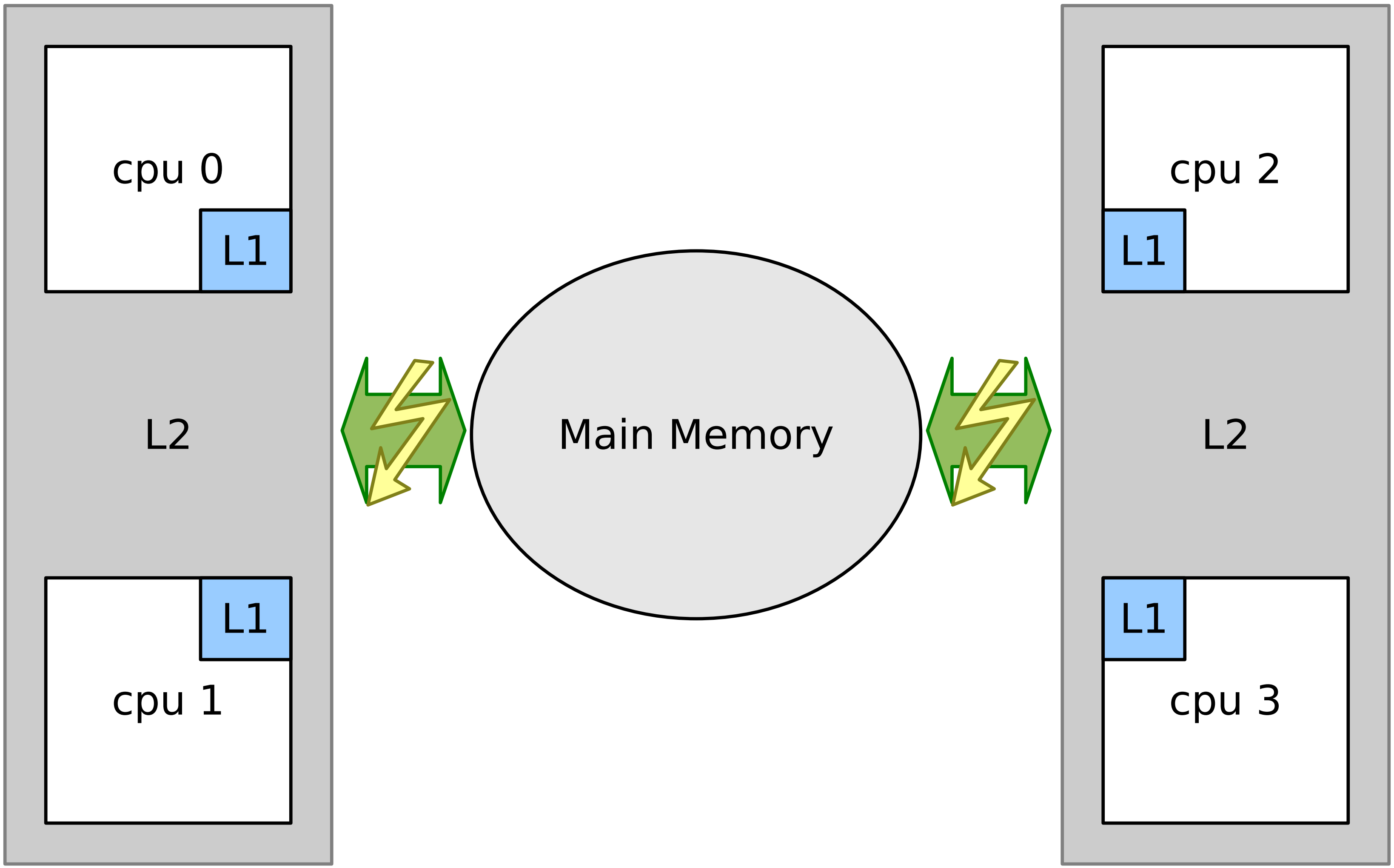
A Guide To OpenMP

Brett Estrade
<estrabd@cs.uh.edu>

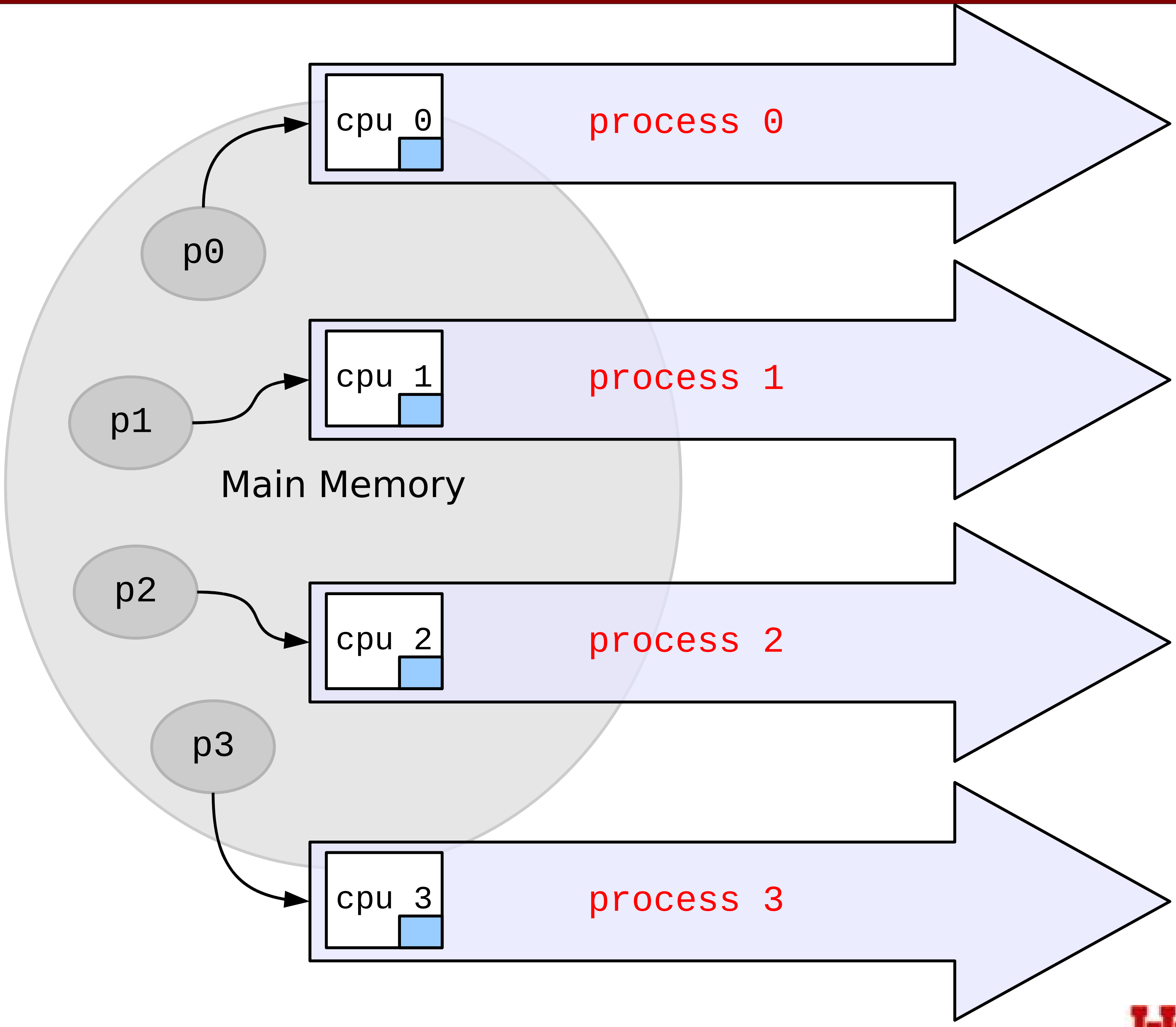
HPCTools Group
University of Houston
Department of Computer Science

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

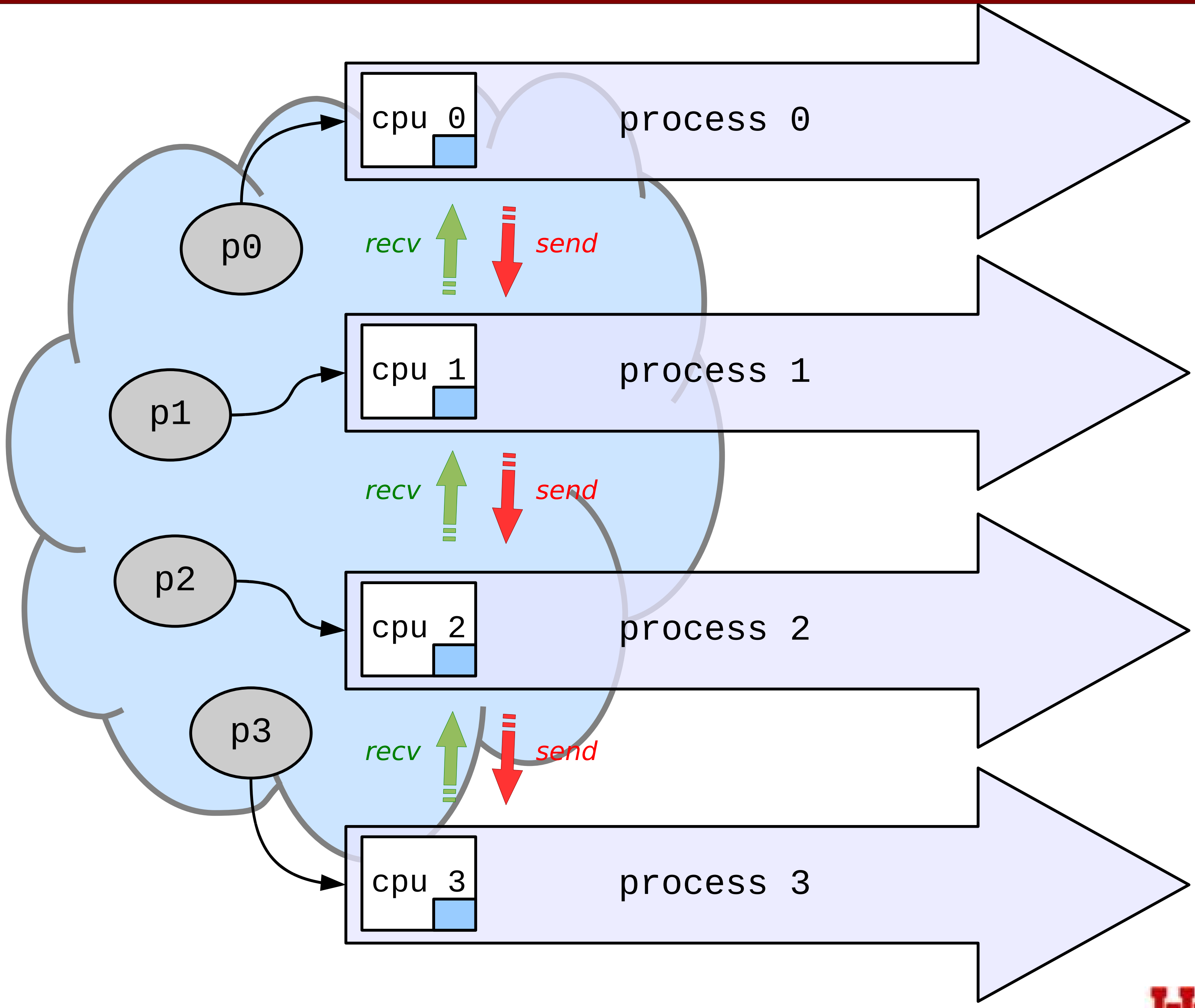
An Example Shared Memory System & Cache Hierarchy



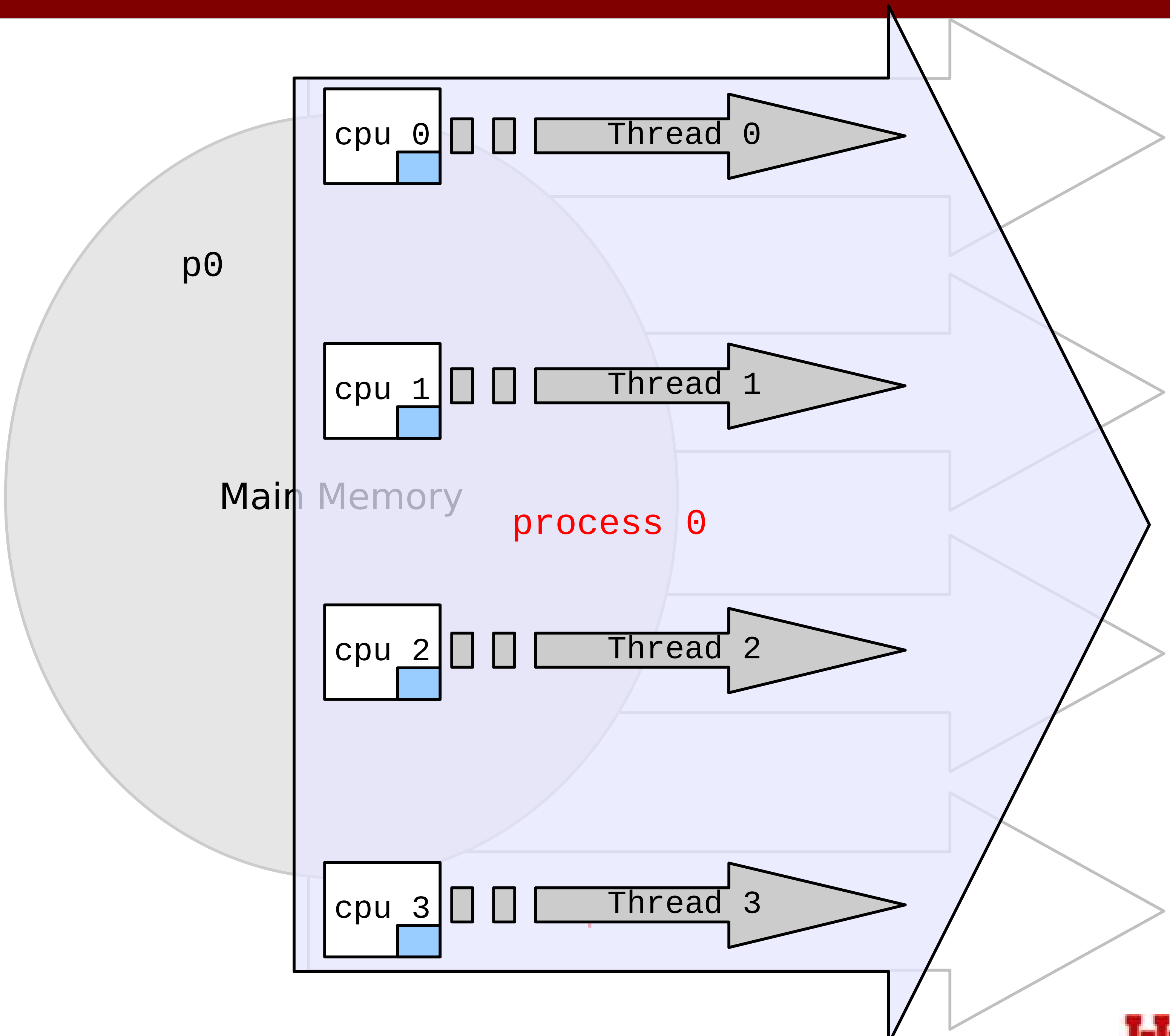
Multiple Processes on this System



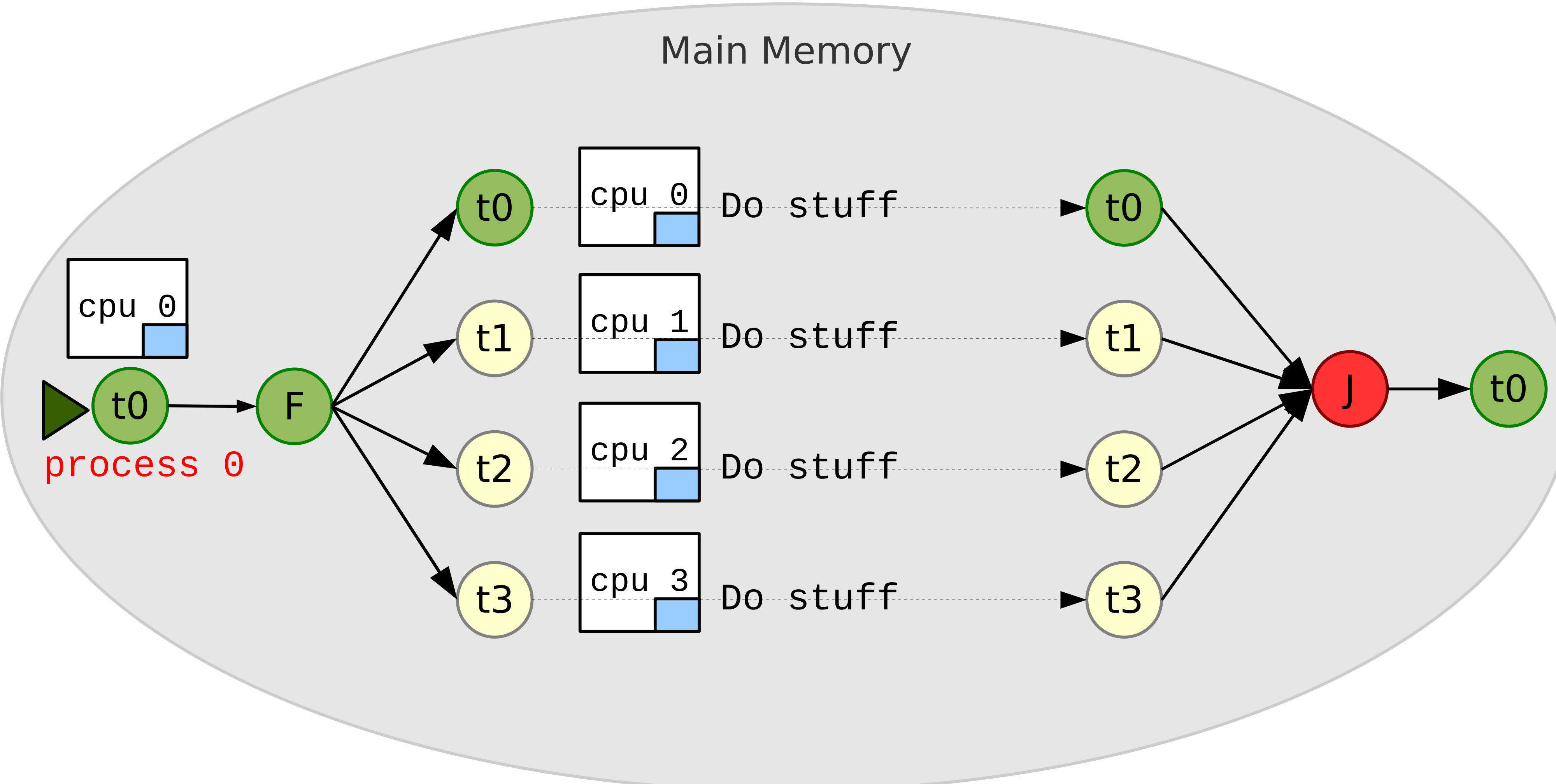
Multiple Processes on this System – *logically no different than MPI's model*



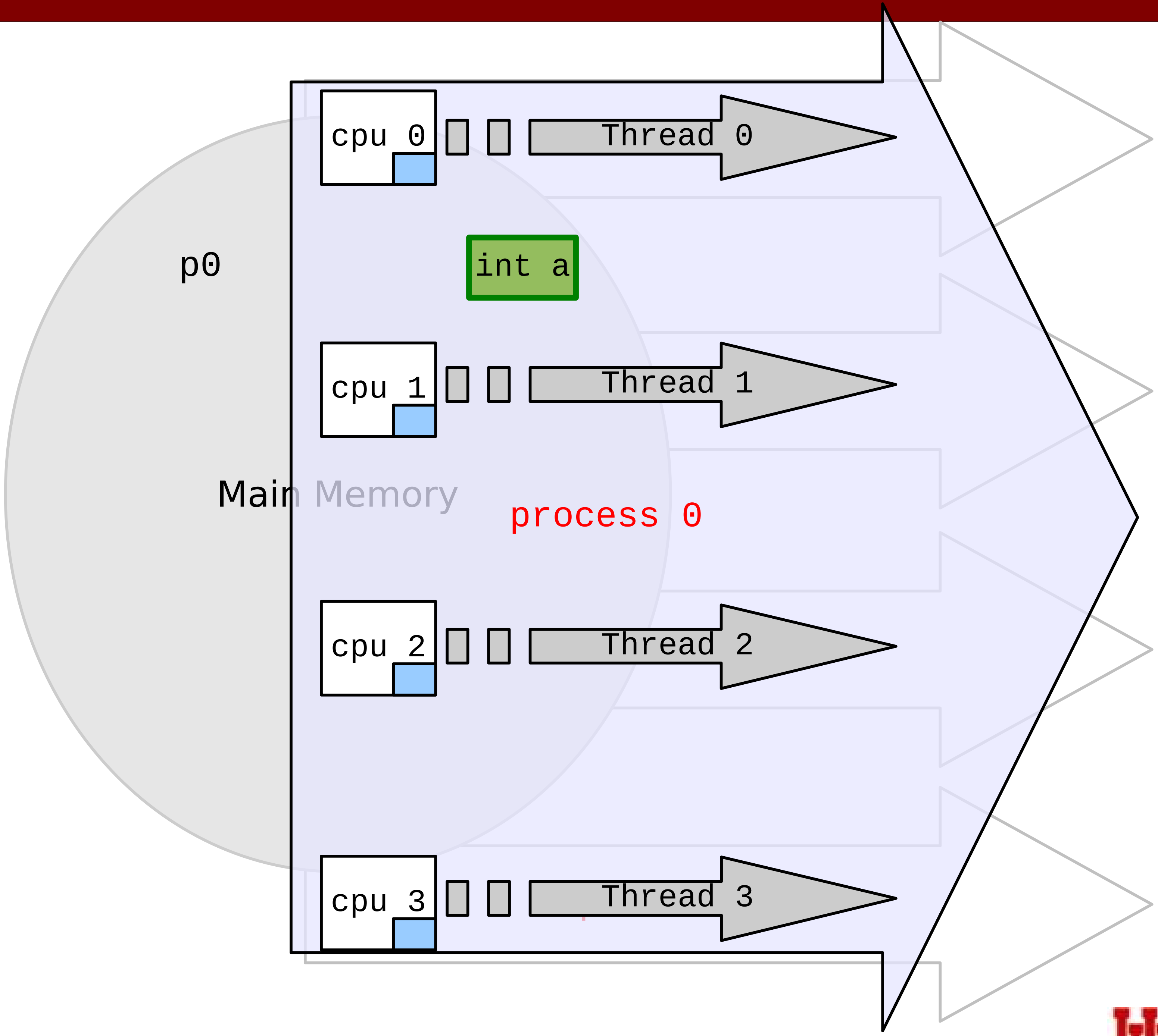
A Single Process with Multiple Threads on this System



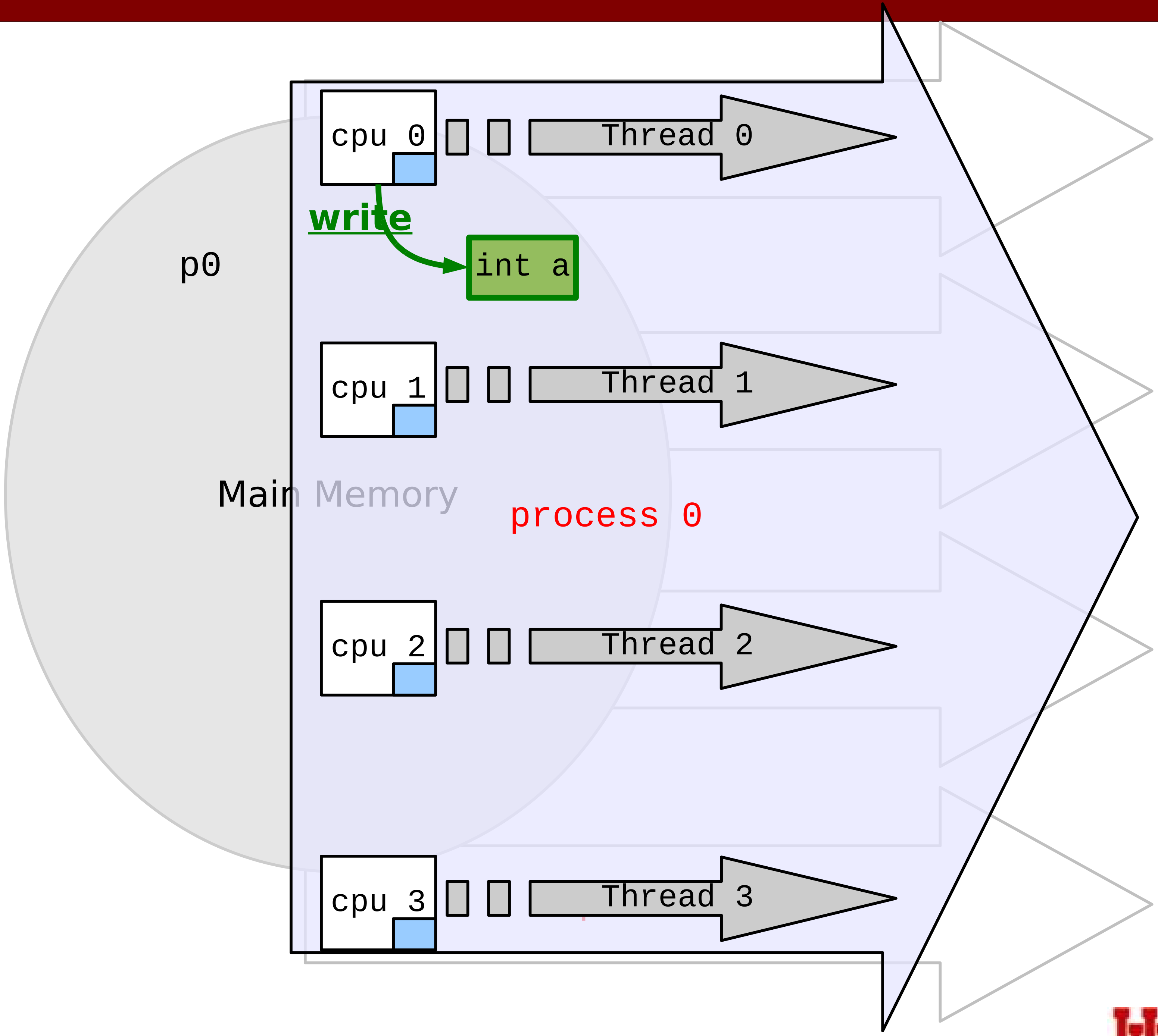
OpenMP's Execution Model – *Fork & Join*



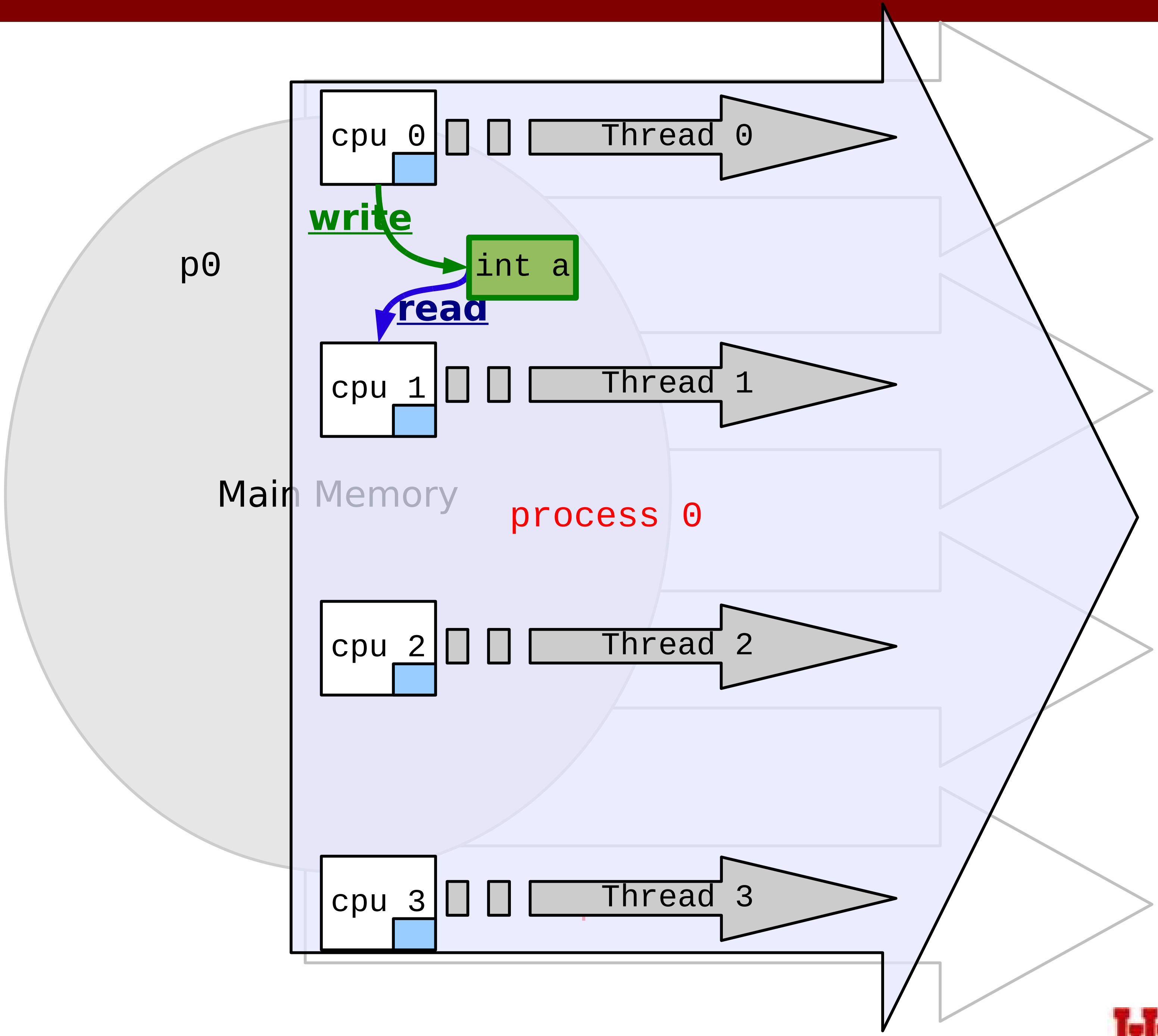
An Example of Sharing Memory



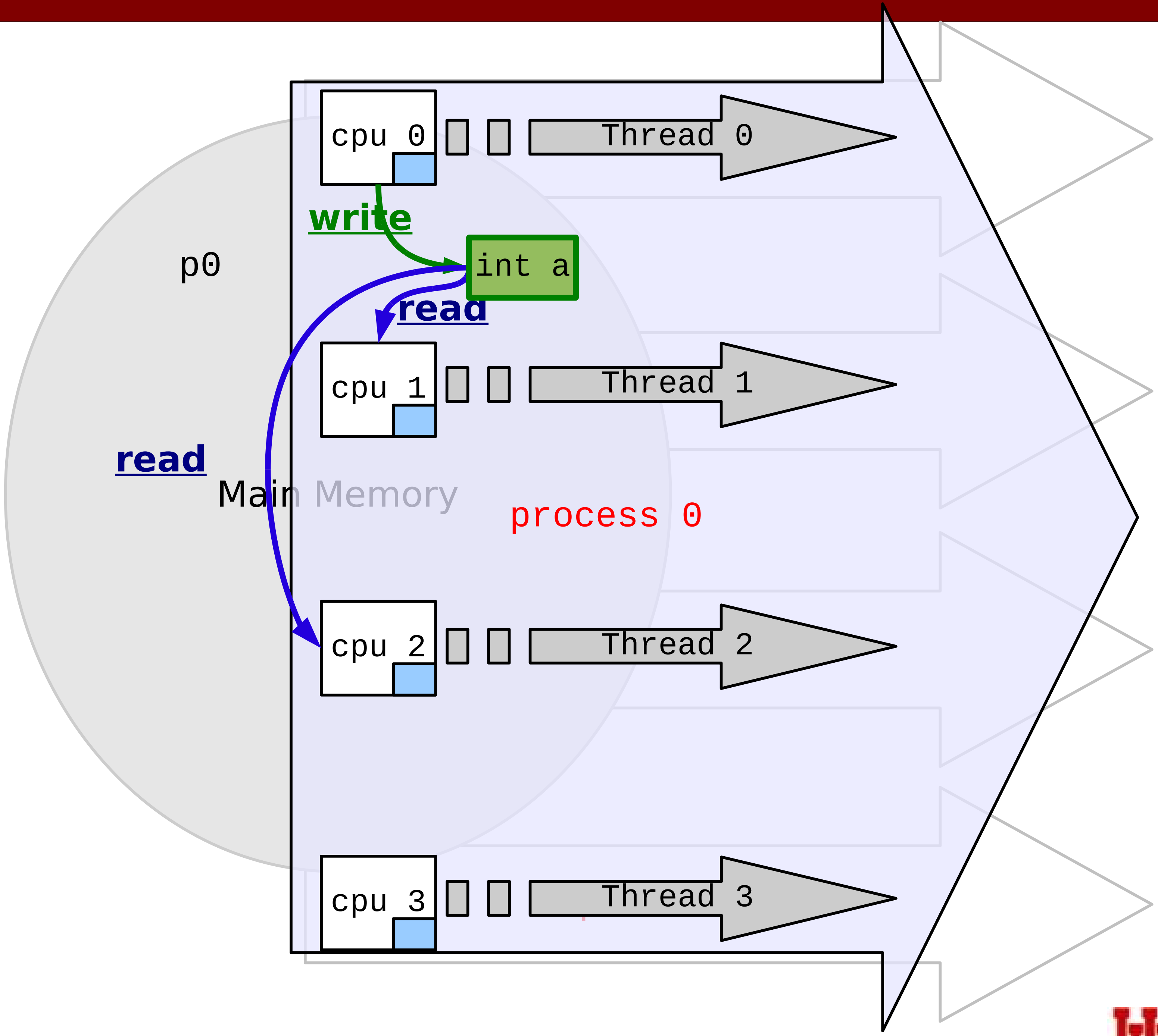
An Example of Sharing Memory



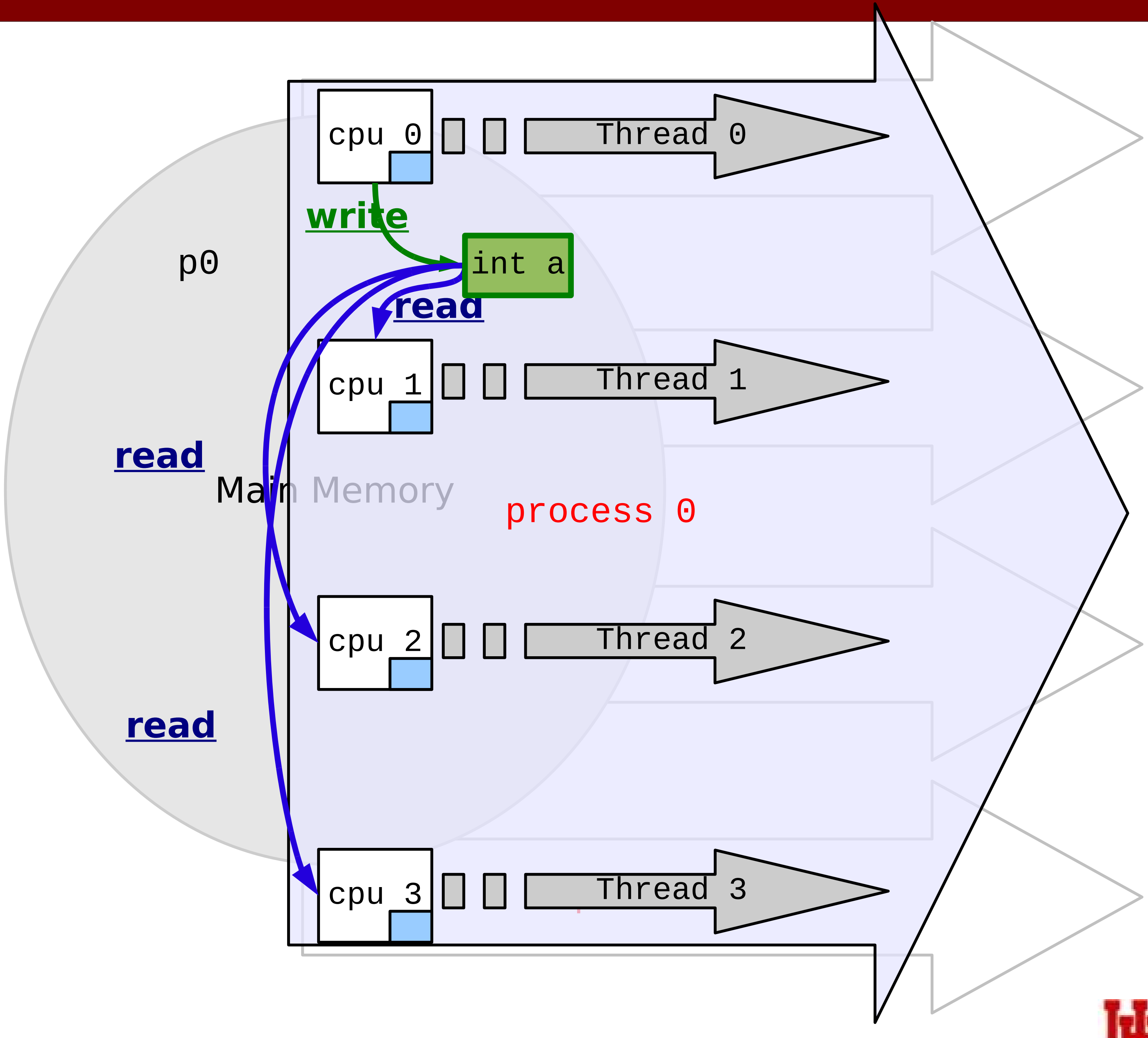
An Example of Sharing Memory



An Example of Sharing Memory



An Example of Sharing Memory



Notes on the Example of Sharing Memory

- Okay, so that was *highly* idealized
- Read/Write order matters (R/W hazards apply)
- Could represent a *race condition*
- Race conditions introduce *non-determinism* (not good)
- Threaded programs can be extremely difficult to debug
- Proper precautions must be made to eliminate these

What is OpenMP?

- A directive based language standard
- A user level API and *runtime* environment
- A widely supported standard language specification
- A *community* of active users & researchers

The Anatomy of an OpenMP Program


```
#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;
}
```

`parallel` (fork) directive

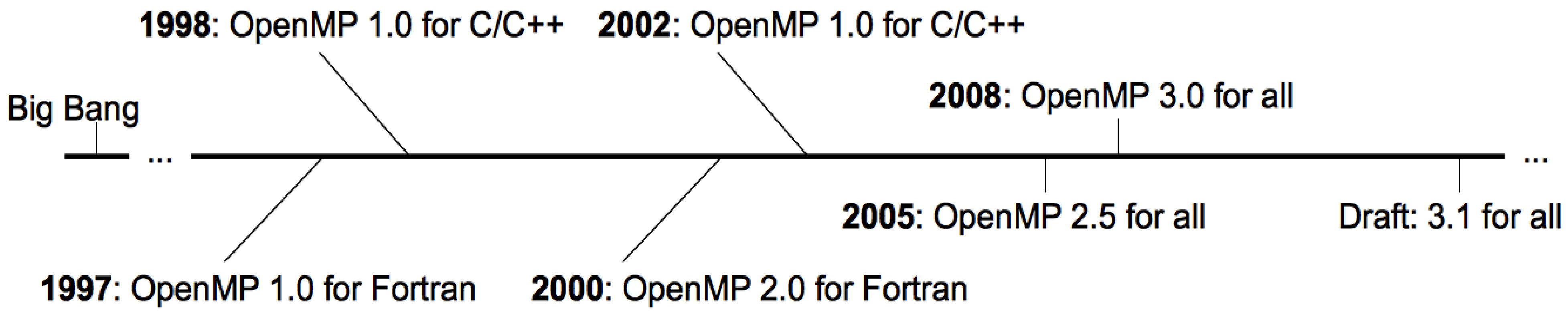
structured `parallel` block

runtime function

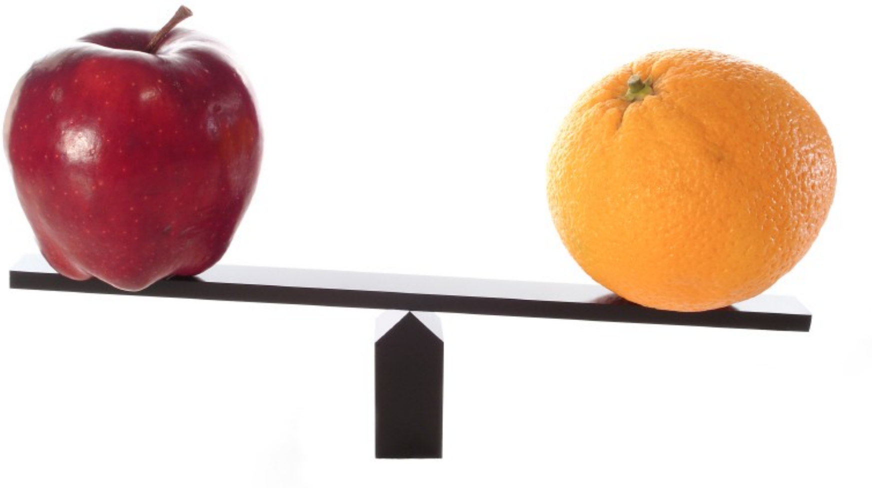
clauses

directive (thread `barrier`)

The timeline of the OpenMP Standard Specification



OpenMP vs MPI



There is no silver bullet

And that makes Teen Wolf happy

Some Benefits of using OpenMP

- It's **portable**, supported by most C/C++ & Fortran compilers
- 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
- Control via environmental 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

OpenMP Research Activities

- A lot of research goes into the OpenMP's standard
- International Workshop on OpenMP (IWOMP)
- Suites: validation, NAS, SPEC, EPCC, BOTS
- Open Source Research Compilers:
  - **OpenUH**
  - NANOS
  - Rose/{OMNI,GCC}
  - **MPC**, etc
  - Commercial R&D
- cOMPunity - `http://www.compunity.org`
- Applications research, i.e., HPC users, etc

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
```

OpenMP Directives & Constructs

- Contained inside of *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

The OpenMP *“Runtime” Library (RTL)*

- 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
- http://www2.cs.uh.edu/~estrabd/OpenUH/r593/html-libopenmp/
- `__omp_fork(...) call graph`

Useful OpenMP Environmental Variables

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

3 Types of Runtime Functions

Execution environment routines; e.g.,

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

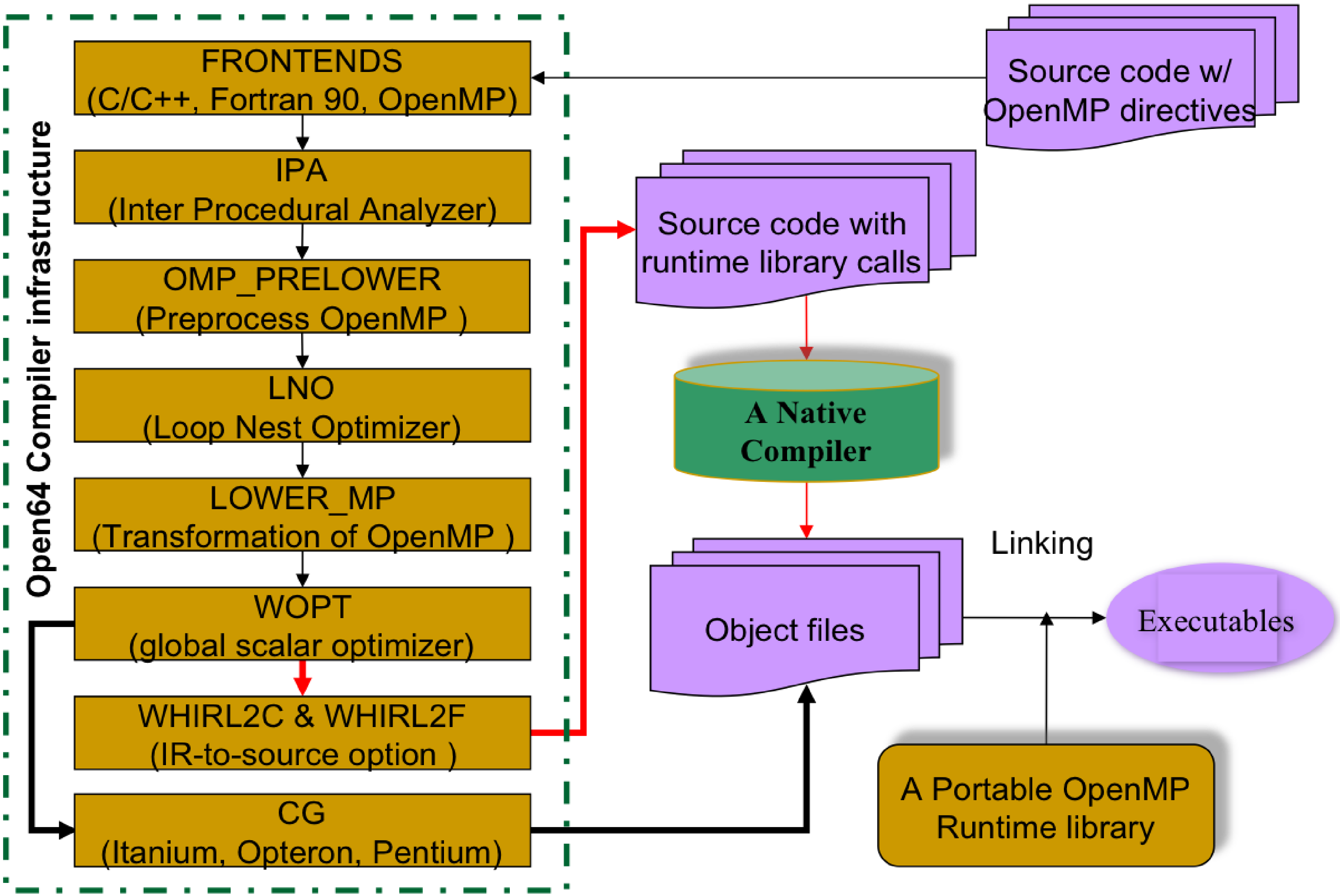
Locking routines; e.g.,

- `omp_{init,destroy}_{,nest_}lock`
- `omp_test_{,nest_}lock`
- `omp_{set,unset}_{,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

What Does the Transformed Code Look like?

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

```
#include <stdio.h>
int main() {
  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_PU(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 */
```

main is outlined to `__omprg_main_1()`

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

No body wants to code like this, so let the compiler and runtime do most all this tedious work!

The parallel Construct

- Where the "fork" occurs (`__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 lexically 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 <u>4 *threads*:</u>

```
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++

```
#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:: id, 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 <u>4 *threads*:</u>

```
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++

```
#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

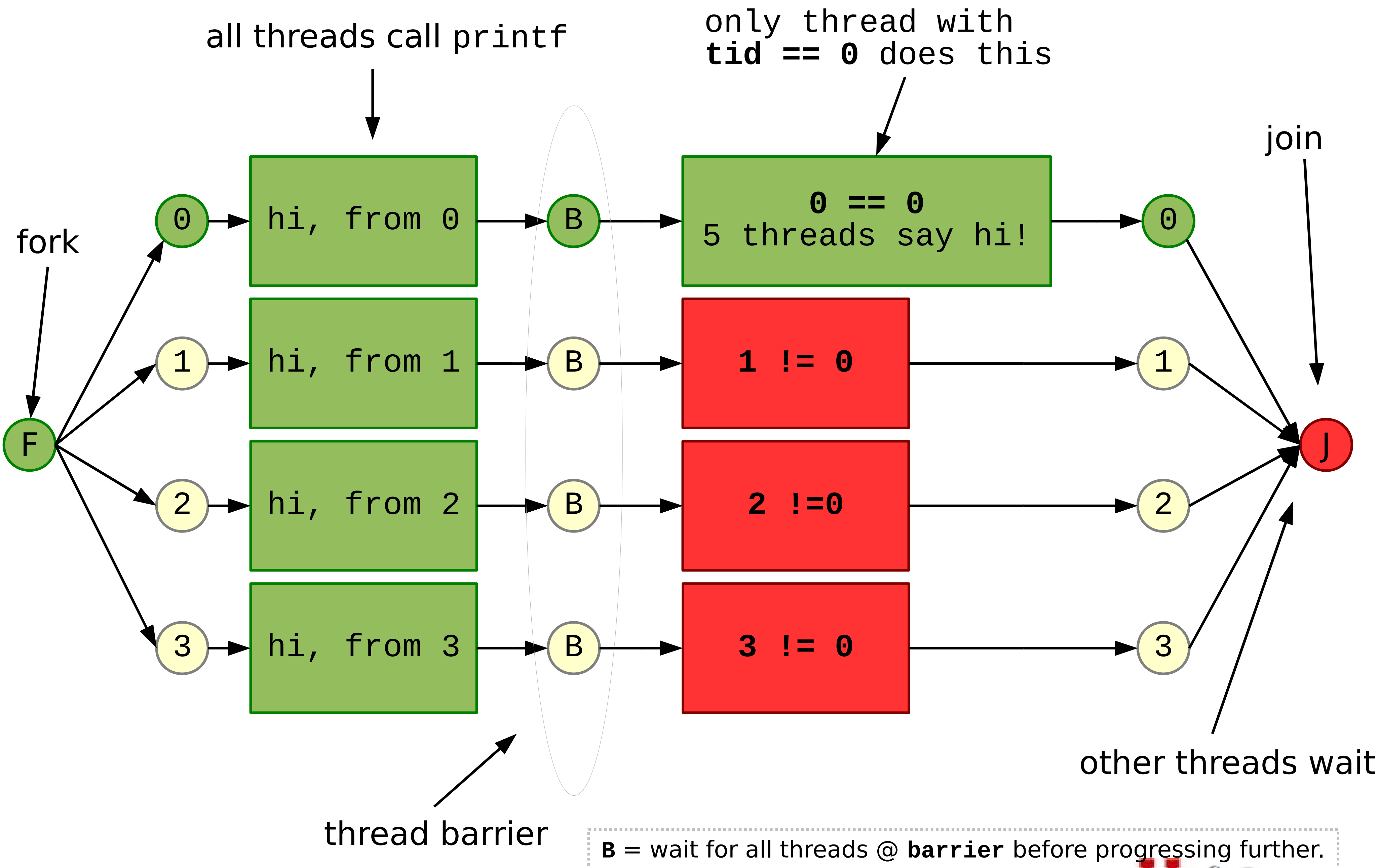
```
      program hello90
      use omp_lib
      integer:: id, numt
      numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
      tid = omp_get_thread_num()
      write (*,*) 'hi, from', tid
!$omp barrier
      if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
     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

Output using <u>4 *threads*:</u>

```
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



Controlling the Fork at Runtime

- 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
  }
```

HPCTools

The Data Environment Among Threads

- `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` - its own directive

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

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

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

Private Variable Initialization

- `private(list,)`
  - Initialized value of variable(s) is undefined
- `firstprivate(list,)`
  - Initialized private variables with value at time of fork to the `master`'s value
- `copyin(list,)`
  - Initialize private variables with the value of `master`'s list
- `threadprivate(list,)`
  - Provides for the initialized of private variables that are treated as global variables inside of each thread
  - **`static`** variables in C/C++
  - **`COMMON`** blocks in Fortran

Getting Data Out

- Variables in `list` are technically shared
- `copyprivate(list,)`
  - Used by `single` to pass list to corresponding private vars in the other threads
- `lastprivate(list,)`
  - vars in `list` will be assigned the last value assigned to it by a thread
  - supported by loop & `sections` construct
- `reduction(<op>:list,)`
  - aggregates vars in list using the defined operation
  - supported by `parallel`, loop, & `sections` constructs
  - <op> must be an actual operator or an intrinsic function

private & shared in that 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;
}
```

F90

```
      program hello90
      use omp_lib
      integer:: id, 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 <u>4 *threads*:</u>

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

Note, thread order not guaranteed!

Memory Consistency Model

- OpenMP uses a "relaxed consistency" model
- In contrast to "sequential consistency"
- 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,)
```

Synchronization

- **Explict** sync points are enabled with a `barrier`:

```
#pragma omp barrier
```

```
!$omp barrier
```

- **Implicit** sync points exist at the end of:
  - `parallel, for, do, sections, single, WORKSHARE`
- Implicit barriers can be turned off with, "`nowait`"
- There is no barrier associated with:
  - `critical, atomic, master`
- Explicit barriers must be used if this is required

An explicit barrier in that 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;
}
```

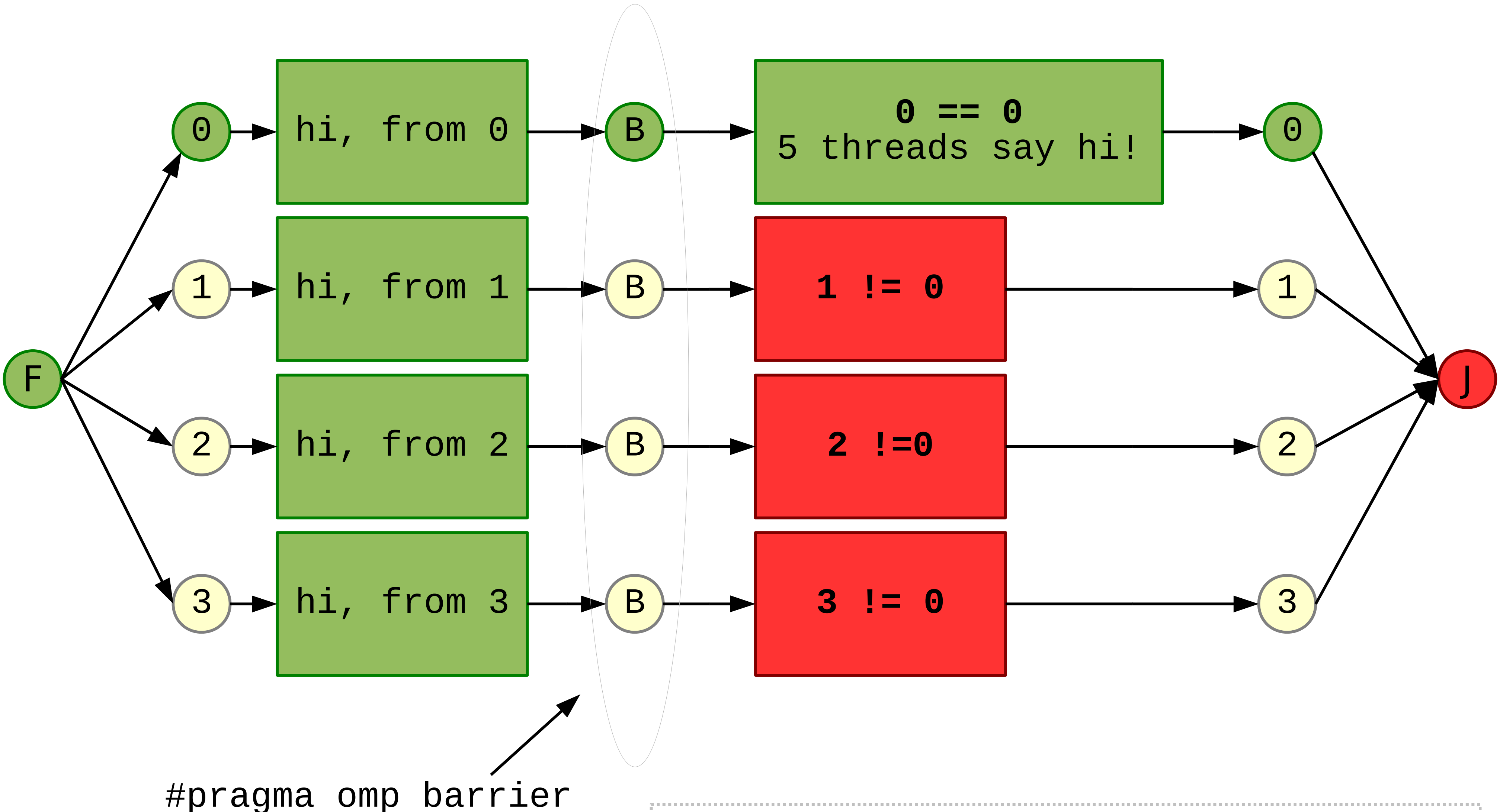
F90

```
      program hello90
      use omp_lib
      integer:: id, 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 <u>4 *threads*</u>:

```
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.

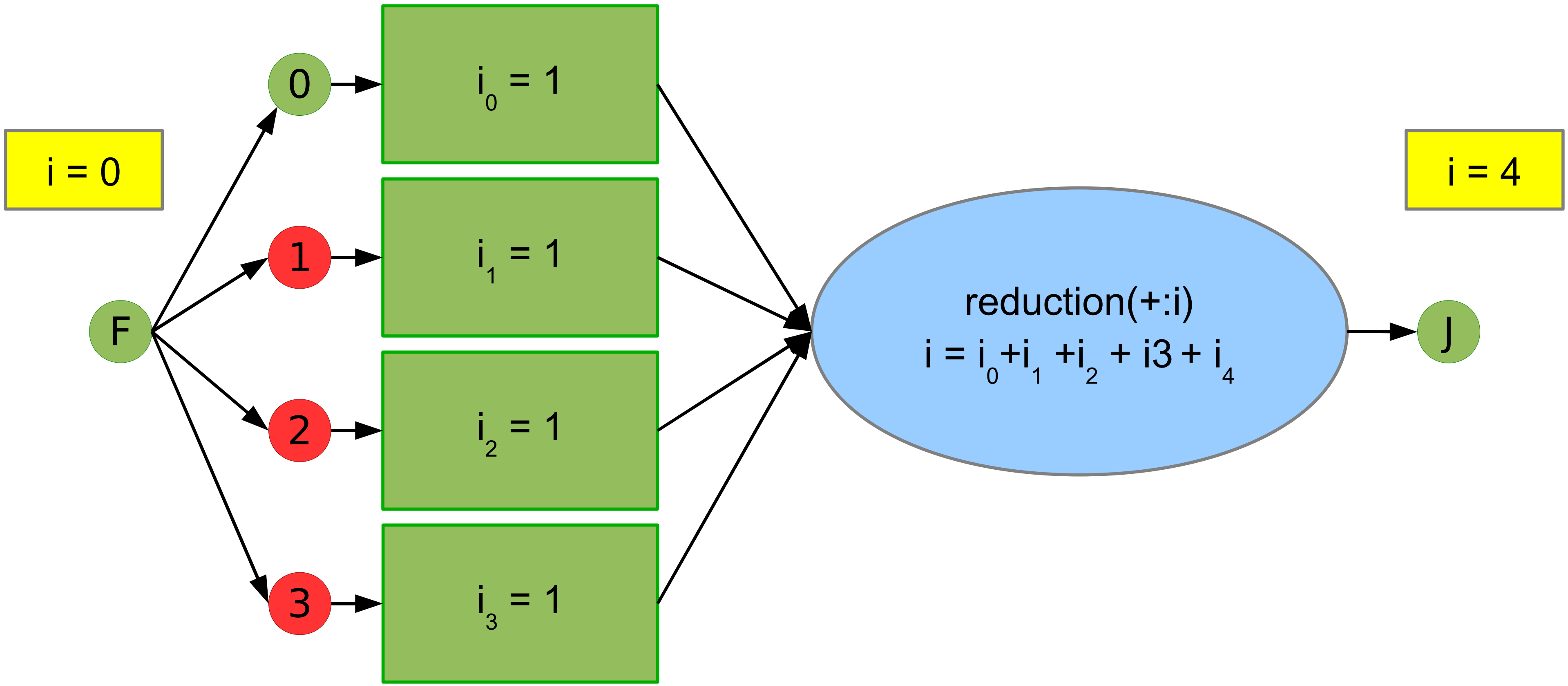
The `reduction` Clause

- Supported by parallel and worksharing constructs
  - `parallel, for, do, sections`
- Creates a private copy of a shared var for each thread
- At the end of the construct containing the `reduction` clause, all private values are *reduced* into one using the specified operator or intrinsic function

```
#pragma omp parallel reduction(+:i)
```

```
!$omp parallel reduction(+:i)
```

Trace of a variable reduction



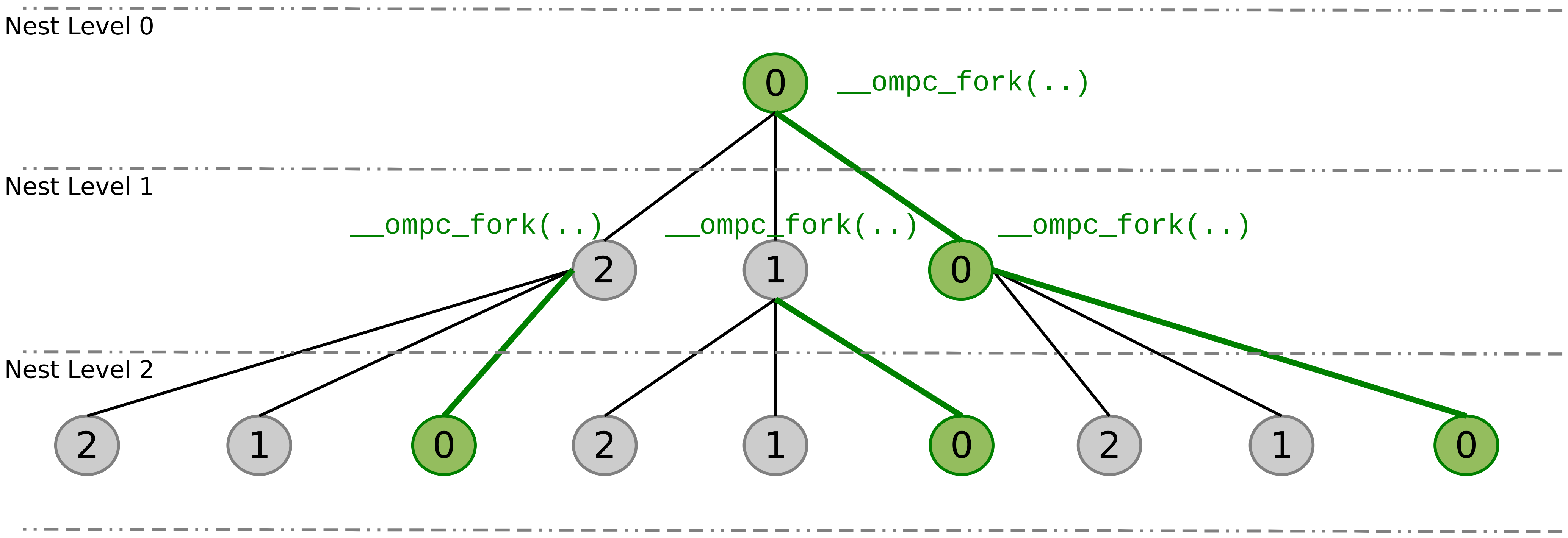
Valid Operations for the `reduction` Clause

- 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 Uniques of Thread Numbers in Nesting



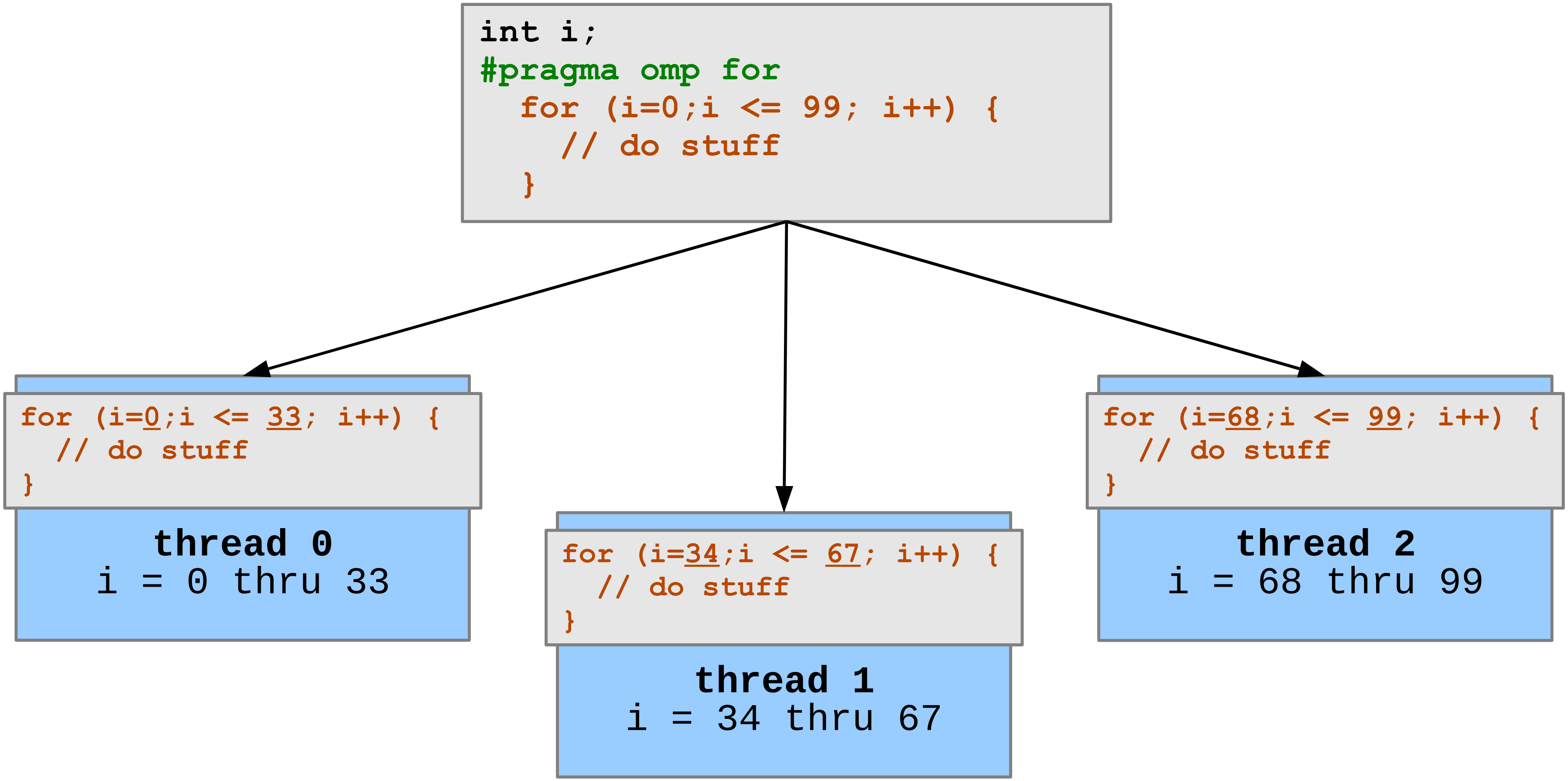
Work Sharing Constructs

- Threads share work in shared memory.
- OpenMP provides "work sharing" contructs
- These constructions include:
  - `for, DO`
  - `sections`
  - `WORKSHARE` (Fortran only)
  - `single, master`

The Loop Constructs: `for` & `DO`

- The loop constructs distribute iterations among threads according to some schedule (default is *static*)
- Among first constructs used when introducing OpenMP
- The clauses supported by the loop constructions are: `private, firstprivate, lastprivate, reduction, schedule, order, collapse, nowait`
- The loop's <u>schedule</u> refers to the runtime policy used to distribute work among the threads.

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++


```
#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 - 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
```

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

Parallel Loop Scheduling

- 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 at 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];
 }
```

The `ordered` Clause and `ordered` construct

- An `ordered` loop contains code that must execute in serial order
- The ordered code must be inside of an `ordered` construct

```
#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp for ordered
    for (i = 0; i <= 99; i++) {
      // do a lot of stuff concurrently
#pragma omp ordered
      {
        a = i * (b + c);
        b = i * (a + c);
        c = i * (a + b);
      }
    }
  }
```

ordered clause

ordered construct

The `collapse` Clause

- Specifies how many loop levels are to be associated with the loop construct
- The n levels are collapsed into a combined iteration space
- The `schedule` applies the entire iteration space as usual

```
#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp for schedule(dynamic,4) collapse(2)
    for (i = 0; i <= 99; i++) {
      for (j = i; j <= 99; j++) {
        // do stuff for each i,j
      }
  }
```

The WORKSHARE Construct (Fortran Only)

- Provides for parallel execution of code using F90 array syntax
- The clauses supported by the WORKSHARE construct are: `private, firstprivate, copyprivate, nowait`
- There is an implicit barrier at the end of this construct
- Valid Fortran code enclosed in a `workshare` construct:
  - Array & scalar variable assignments
  - FORALL statements & constructs
  - WHERE statements & constructs
  - User defined functions of type ELEMENTAL
  - OpenMP `atomic`, `critical`, & `parallel`

The `sections` Construct

- The `sections` construct defines code that is to be executed once by exactly one thread
- A barrier is implied
- Supported clauses include: `private, firstprivate, lastprivate, reduction, nowait`

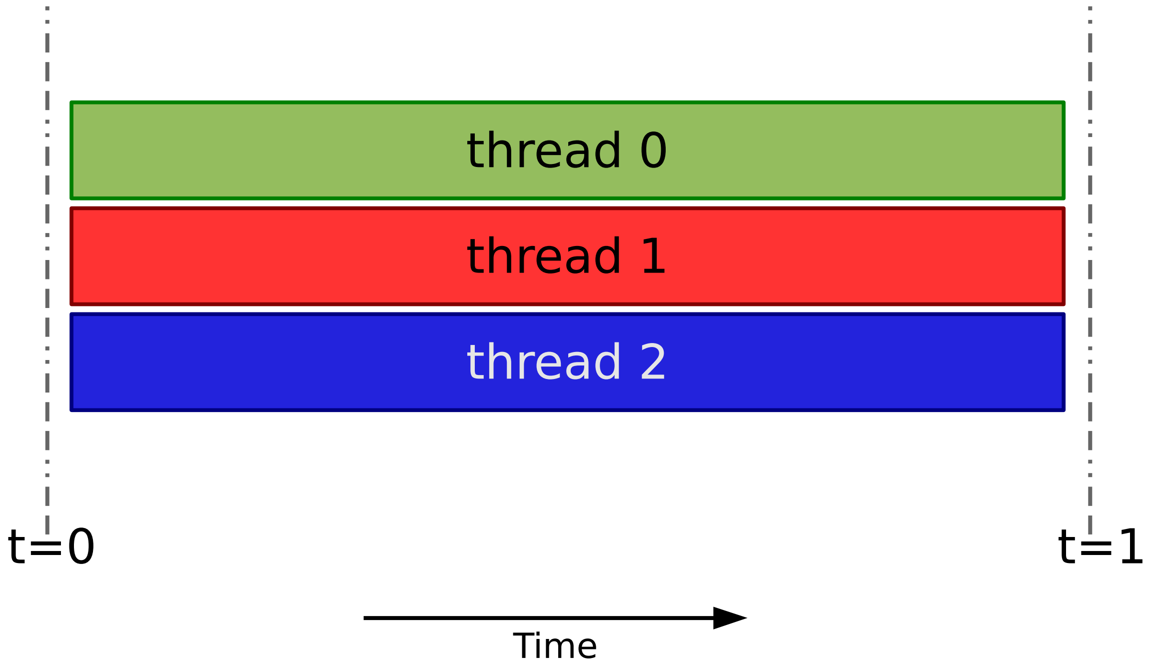
A section Construct Example


```
#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
 {
#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);
     }
  }
```



Combined `parallel` Constructs

- `parallel` may be combined with the following:
  - `parallel, for, do, sections, WORKSHARE`
- Semantics are identical to usage already discussed

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

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

Singling Out Threads with `master` and `single` Constructs

- Code inside of A `master` construct will only be executed by the master thread.
- There is NO implicit barrier associated with `master`; other threads ignore it.

```
!$OMP MASTER
  … do stuff
!$OMP END MASTER
```

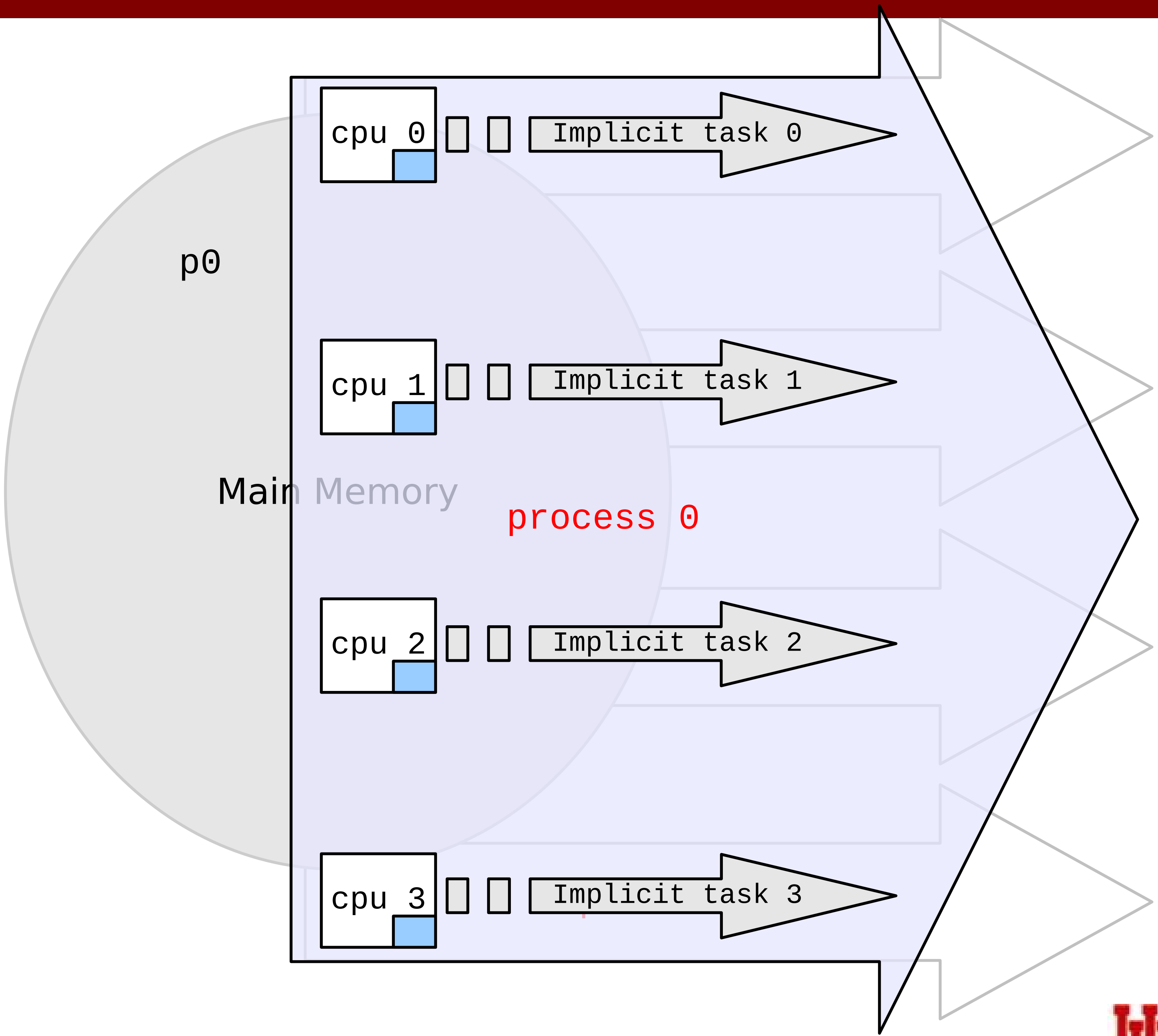
- Code inside of a `single` construct will be executed by the first thread to encounter it.
- A `single` construct contains an implicit barrier that will respect `nowait`.

```
!$OMP SINGLE
  … do stuff
!$OMP END SINGLE [nowait]
```

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

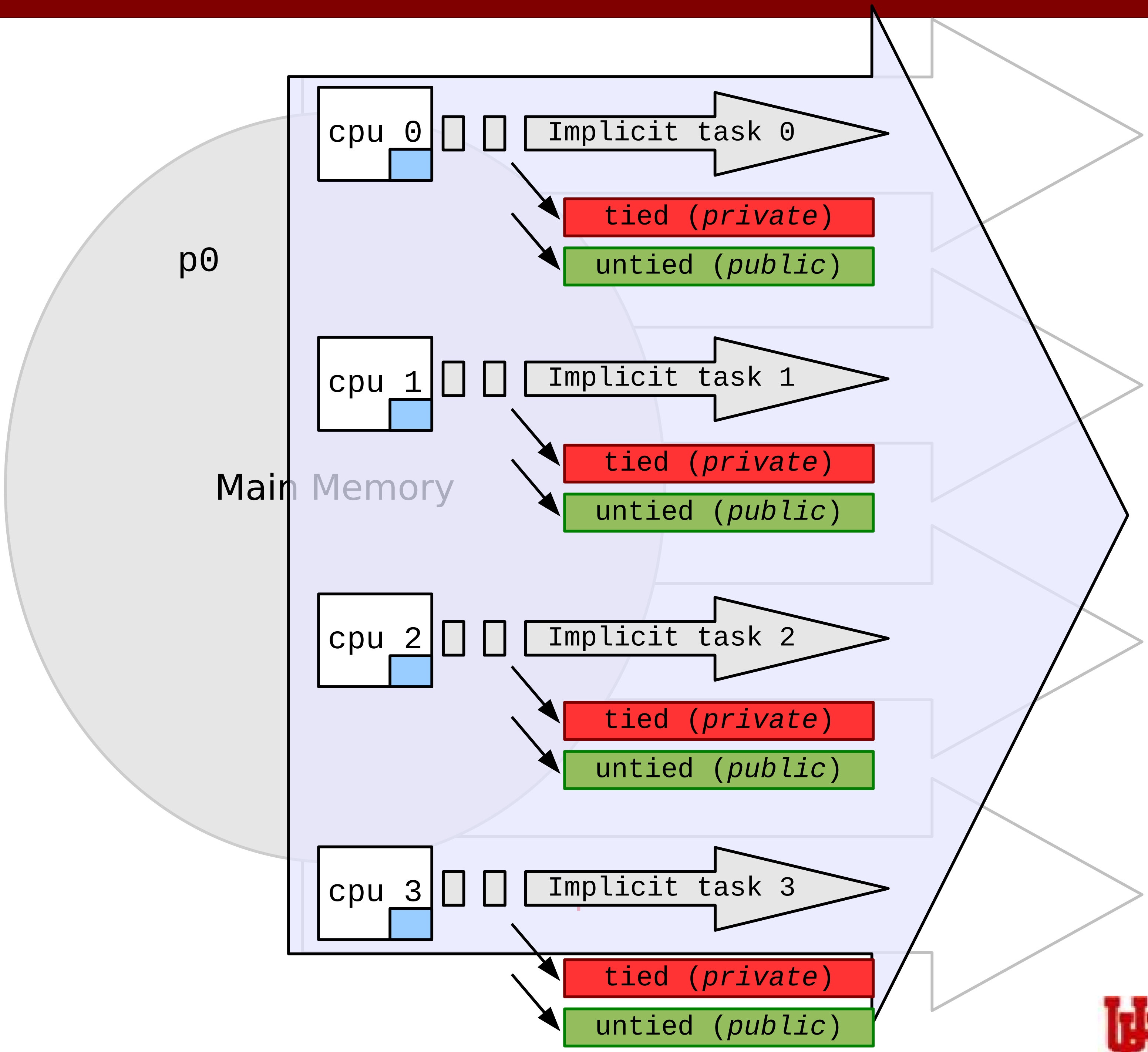
Threads are now Implicit Tasks



The `task` Construct

- 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

Each Thread Conceptually Has Both a tied & untied queue



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);
}
```

A task Construct Fortran Example


```
      RECURSIVE SUBROUTINE traverse ( P )
        TYPE Node
          TYPE(Node), POINTER :: left, right
        END TYPE Node
        TYPE(Node) :: P
        IF (associated(P%left)) THEN
!$OMP TASK ! P is firstprivate by default
          call traverse(P%left)
!$OMP END TASK
        ENDIF
        IF (associated(P%right)) THEN
!$OMP TASK ! P is firstprivate by default
          call traverse(P%right)
!$OMP END TASK
        ENDIF
        CALL process ( P )
END SUBROUTINE
```

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

The `critical` Construct

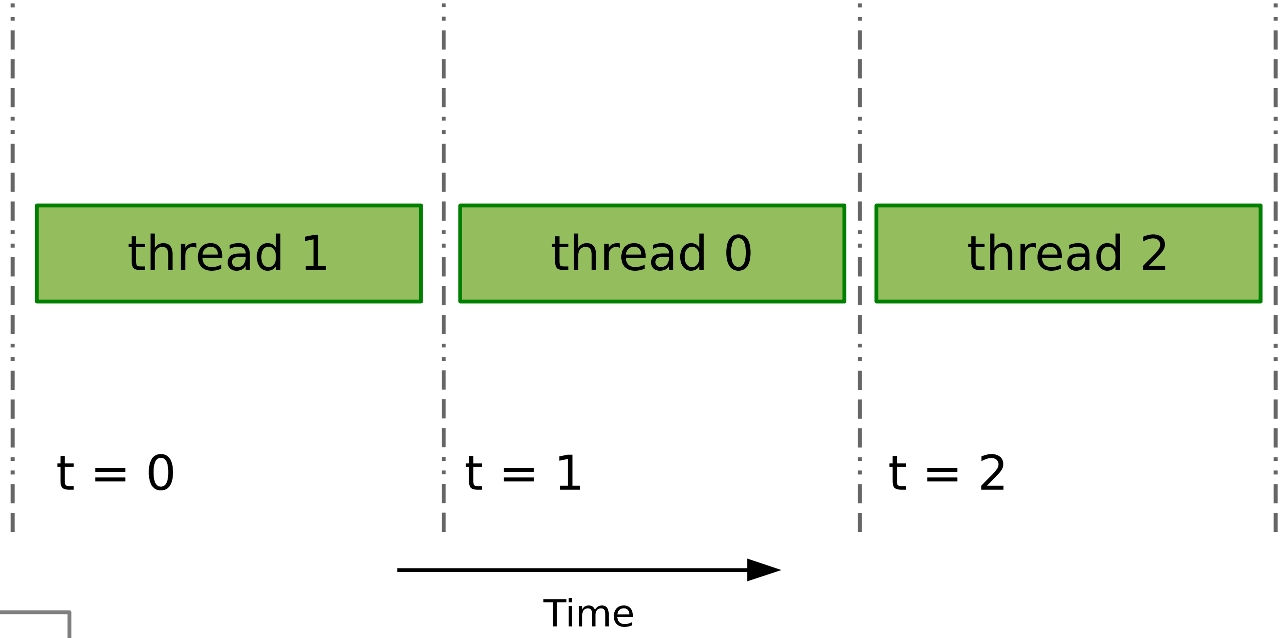
- The critical construct enclose code that should be executed by *all* threads, just in some serial order

```
#pragma omp parallel
  {
#pragma omp critical
    {
      // some code
    }
  }
```

- The effect is equivalent to a lock protecting the code

A critical Construct Example

```
#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp critical
    {
      //
      // do stuff (one thread at a time)
      //
    }
  }
```



Note:
Encountering thread
order not gauranteed!

The Named `critical` Construct

- Names may be applied to critical constructs.

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

- The effect is equivalent to using a different lock for each section.

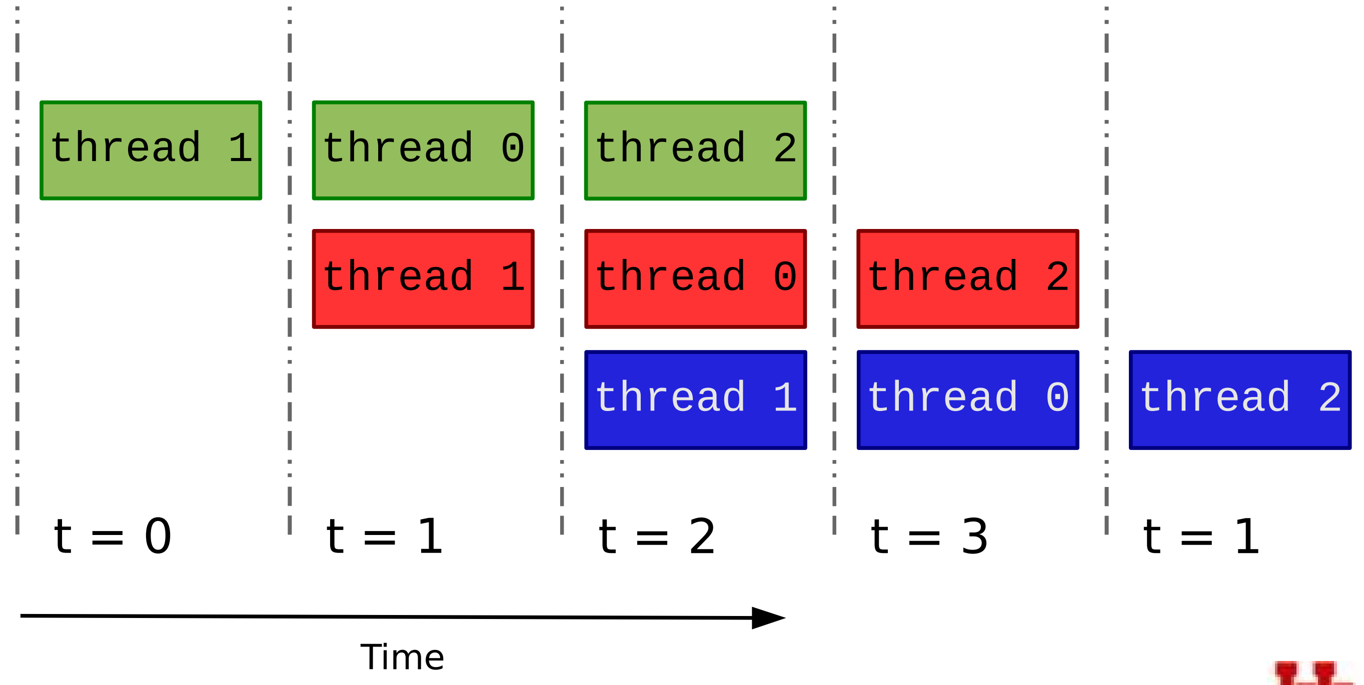
A Named `critical` Construct Example


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

int main(void)
{ float a[N], b[N], c[3];
  int i;
  /* Initialize arrays a and b. */
   for (i = 0; i < N; i++)
     { a[i] = i * 1.0 + 1.0;
       b[i] = i * 2.0 + 2.0;
     }
  /* Compute values of array c in parallel. */
#pragma omp parallel shared(a, b, c) private(i)
  {
#pragma omp critical(a)
    { for (i = 0; i < N; i++)
          C[ 0] += a[i] + b[i];
      printf("%f\n",c[0]);
    }
#pragma omp critical(b)
    { for (i = 0; i < N; i++)
          c[1] += a[i] + b[i];
      printf("%f\n",c[1]);
    }
#pragma omp critical(c)
    { for (i = 0; i < N; i++)
         c[2] += a[i] + b[i];
      printf("%f\n",c[2]);
    }
  }
}
```

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 gauranteed!



The `atomic` Construct for Safely Updating Shared Variables

- Protected writes to shared variables
- Lighter weight than using a `critical` contruct

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

int main(void) {
  int count = 0;
#pragma omp parallel shared(count)
  {
 #pragma omp atomic
      count++;
  }
  printf("Number of threads: %d\n",count);
}
```

Note:
Encountering thread order not gauranteed!

Locks in OpenMP

- `omp_lock_t, omp_lock_kind`
- Threads set/unset locks
- Nested locks can be set multiple times by the same thread before releasing them
- More flexible than `critical` construct

Using Locks in OpenMP


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

int main()
{
  int x;
  omp_lock_t lck;
  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;
#pragma omp parallel shared (x)
  {
#pragma omp master
    {
      x = x + 1;
      omp_unset_lock (&lck);
    }
/* Some more stuff. */
  }
  omp_destroy_lock (&lck);
}
```

Using Nested Locks in OpenMP


```
#include <omp.h>
typedef struct {
  int a,b; omp_nest_lock_t lck; } pair;
int work1();
int work2();
int work3();

void incr_a(pair *p, int a) {
  /* Called only from incr_pair, no need to lock. */
  p->a += a;
}
void incr_b(pair *p, int b) {
  /* Called both from incr_pair and elsewhere, */
  /* so need a nestable lock. */
  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);
}

void a45(pair *p) {
#pragma omp parallel sections
  {
#pragma omp section
    incr_pair(p, work1(), work2());
#pragma omp section
    incr_b(p, work3());
  }
}
```

Fortran Programming Tips

- In fixed form Fortran OpenMP directives can hide behind the following "sentinals"

  `!$[OMP],c$[OMP],*$[OMP]`

- Free form requires "`!$`"
- Sentinals can enable conditional compilation

  `!$ omp_set_num_threads(n)`

- Fortran directives should start in column 0
- Long directive continuations take a form similar to:

```
!$OMP PARALLEL DEFAULT(NONE)
!$OMP& SHARED(INP,OUTP,BOXL,TEMP,RHO,NSTEP,TSTEP,X,Y,Z,VX,VY,VZ,BOXL)
!$OMP& SHARED(XO,YO,ZO,TSTEP,V2T,VXT,VYT,VZT,IPRINT,ISTEP,ETOT,ERUN)
!$OMP& SHARED(FX,FY,FZ,PENER)
!$OMP& PRIVATE(I)
```

C/C++ Programming Tips

- No line continuations, entire directive on single line
- No conditional compilation sentinals, use "`#ifdef`", etc
- Coding style

all #pragmas in col. 0

braces indented as usual

```
int main () {
   ...
#pragma parallel
   {
#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);
       }
     }
   }
   return 0; /* end main */
}
```

General Programming Tips

- 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 cycle – test against serial version
  - Use barriers liberally
  - Optimize OpenMP & asynchronize **last**
- When starting from scratch
  - Start with an optimized serial version

OpenMP Idioms

- Won't cover directly, but they exist for:
- Pipelining computations
- Effectively using I/O (especially in a pipelined context)
- Creating user defined reductions (UDR) (e.g., for divide & conquer algorithms, map-reduce type applications)
- Interleaving N units of critical work with M threads to minimize idle time
- Effective use of nested `parallelism` and `tasks` for unbalanced and dynamical work loads
- ...many more

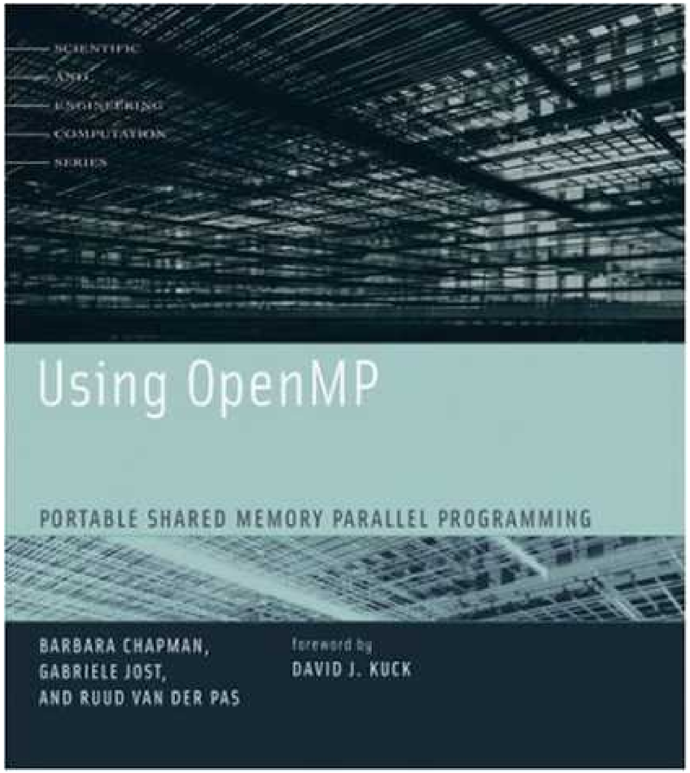
Other Issues Not Covered Directly

- Profiling & optimizations
- Debugging & troubleshooting techniques
- Real world OpenMP
- OpenMP in hybrid contexts

- It's not going anywhere; vendor buy-in is as strong as ever
- Big 3:
  - Refinement to tasking model (scheduling, etc)
  - Error handling
  - Accelerators
- Scaling
  - Thousands of threads
  - Data locality
  - More efficient synchronization constructs & implementations
- Remaining relevant

Additional Resources

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



Covers through 2.5