The Perl-OpenMP Project

Part I:

Introduction to OpenMP for the Perl Programmers

The Perl
& Raku Conference
2022

Perl-OpenMP Project

- Loosely affiliated group of individuals on Github who are interested in Perl and also OpenMP
- Contributors come various interests, including HPC, computer vision, and the Perl Data Language (PDL) Project
- The goal is simply to explore how to best leverage OpenMP, supported by GCC since 4.2 (2005) on modern multi-core

Current CPAN Releases*

- **`Alien::OpenMP`**

  makes Inline::C with OpenMP easy

- **`OpenMP::Environment`**

  provides *Perlish* way to manipulate environmental variables used by OpenMP at runtime.

** more on these in Part II!*

Goals of Part I

- Introduce OpenMP to Perl programmers
- Learn the basics to add OpenMP to C code
- Learn how to compile C with OpenMP
- Learn how to control the *runtime*

Learning OpenMP

- The basics can be covered in ~4 hours
- We have <50 minutes
- Mastering OpenMP can take a semester
- Many in-depths tutorials and YT videos

Learning Perl

- Basics can be covered in a semester
- We have 0 minutes or less today
- Mastering Perl can take a lifetime
- … so we have the advantage here .. so

Let us BEGIN!

Perl-OpenMP Project

- Born out of IRC chatz on `#pdl` and `#native`

- **Goal**:

  Ideate and prove the synergisms between OpenMP & Perl & **`perl`**.

OpenMP is ...

- An easy way to make existing single-threaded programs threaded for multi-core
- Standards based, supported by many compilers - `https://www.openmp.org/resources/openmp-compilers-tools/`
- Available on any modern computer with gcc for C, C++, & Fortran (since v4.2, 2005!) - `https://gcc.gnu.org/projects/gomp`
- Used in Open Source, e.g., ImageMagick

OpenMP consists of ...

- Declarative code annotations
- A compiler provided “runtime”
- Data environment with varying degrees of memory sharing and privacy controls
- A compiler implemented runtime API
- Awareness of some environmental variables

Lesson #1: hi.c

```
#include <stdio.h>

int main (int argc, char*argv[]) {

  {
    printf("hi!\n");
  }
  return 0;
}
```

Before OpenMP

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

int main (int argc, char*argv[]) {
  #pragma omp parallel
  {
    printf("hi!\n”);
  }
  return 0;
}
```

After OpenMP

```
prompt# gcc hi.c -o hi.x
prompt# ./hi.x
hi!
prompt#
```

```
prompt# gcc -fopenmp hi.c -o hi.x
prompt# ./hi.x
hi!
hi!
hi!
hi!
hi!
hi!
hi!
hi!
prompt#
```

Note:
OpenMP was designed to be introduced *incrementally* into *existing* “serial” (single threaded) code by scientific domain experts, who are not always the best programmers.

Now, Add a *Runtime* API Call

Before OpenMP

```
#include <stdio.h>

int main (int argc, char*argv[]) {

  {
    printf("hi!\n");

  }
  return 0;
}
```

After OpenMP

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char*argv[]) {
  #pragma omp parallel
  {
    printf("hi, from thread %d!\n”,
            omp_get_thread_num());
  }
  return 0;
}
```

```
prompt# gcc hi.c -o hi.x
prompt# ./hi.x
hi!
prompt#
```

```
prompt# gcc -fopenmp hi.c -o hi.x
prompt# ./hi.x
hi, from thread 1!
hi, from thread 0!
hi, from thread 2!
hi, from thread 7!
hi, from thread 5!
hi, from thread 4!
hi, from thread 6!
hi, from thread 3!
prompt#
```

Now Set OMP_NUM_THREADS in the %ENV

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char*argv[]) {
  #pragma omp parallel
  {
    printf("hi, from thread %d!\n”,
            omp_get_thread_num());
  }
  return 0;
}
```

```
prompt# gcc -fopenmp hi.c -o hi.x
prompt# OMP_NUM_THREADS=8 ./hi.x
hi, from thread 1!
hi, from thread 0!
hi, from thread 2!
hi, from thread 7!
hi, from thread 5!
hi, from thread 4!
hi, from thread 6!
hi, from thread 3!
```

```
prompt# gcc -fopenmp hi.c -o hi.x
prompt# OMP_NUM_THREADS=4 ./hi.x
hi, from thread 1!
hi, from thread 0!
hi, from thread 2!
hi, from thread 3!
prompt#
```

```
prompt# gcc -fopenmp hi.c -o hi.x
prompt# OMP_NUM_THREADS=2 ./hi.x
hi, from thread 1!
hi, from thread 0!
prompt#
```

Recap Lesson #1

- OpenMP provides declarations hidden behind language comment sentinels; e.g.:

```
#pragma omp parallel
```

- OpenMP provides a Runtime API; e.g.:

```
omp_get_thread_num();
```

- OpenMP is environmentally aware, e.g.:

```
OMP_NUM_THREADS=8
```

What are Declarations?

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

What's a Runtime?

- 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

Data – Shared & Private

- SMP = Shared Memory Programming
- OpenMP shares all data available outside the “fork” point by default
- Private variables are able to be declared
- Initialization of these private variables can be controlled

Controlling Data Sharing

- Default for all variables

```
#pragma omp parallel shared(A,B,C,..)
```

- Copies existing variables type/size for each thread

```
#pragma omp parallel private(D,E,F,..)
```

- Like private, but takes initial value from main thread scope

```
#pragma omp parallel firstprivate(H,I,J,...)
```

- Thread specific global variables (in *bonus* slides).

```
#pragma omp threadprivate(K,L,...)
```

What's the Runtime API?

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`

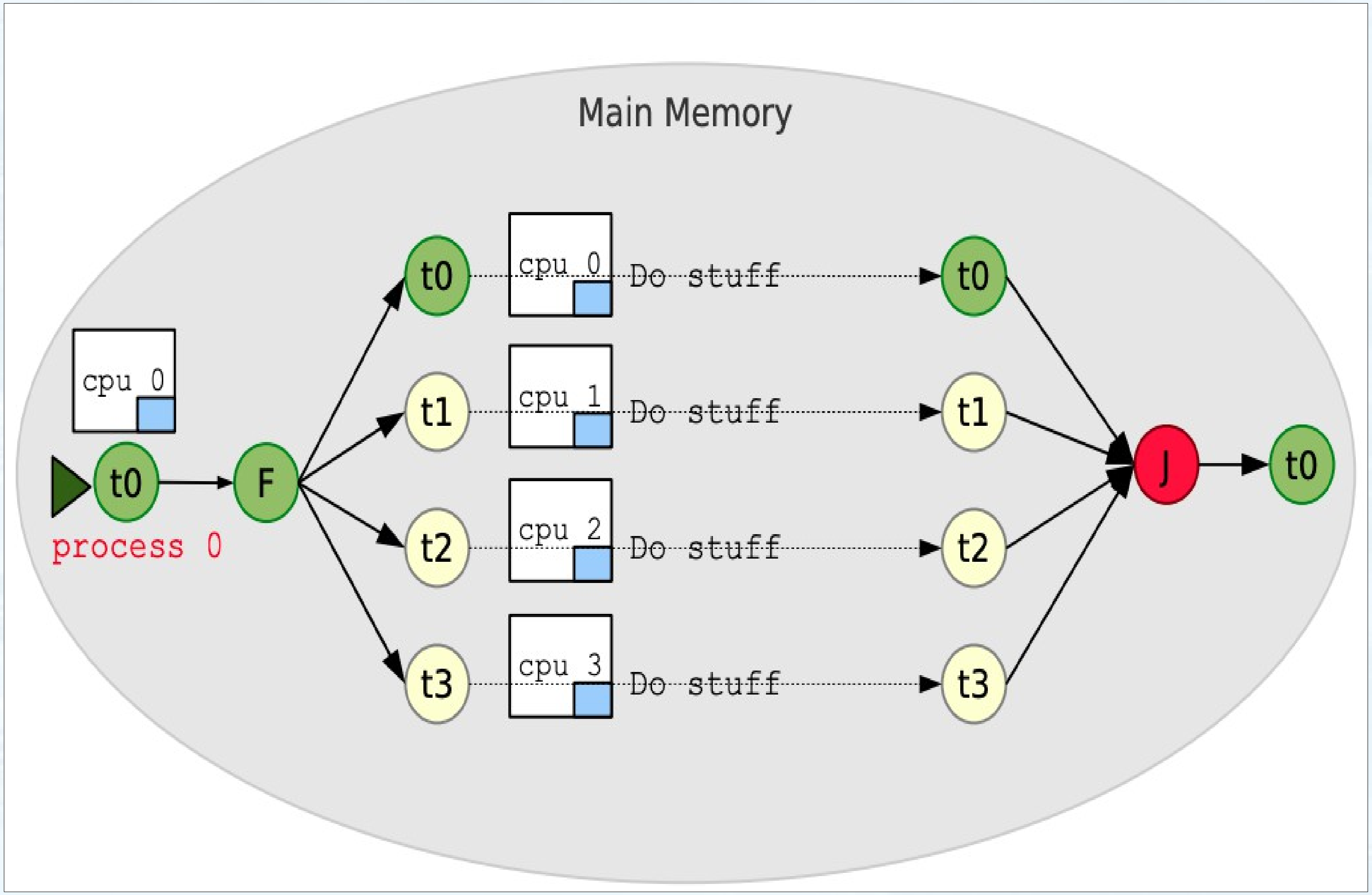
What are the Environmental Vars?

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

Lesson 2: OpenMP Models

- OpenMP's execution model is primarily called, “fork and join”
- OpenMP's memory model is called a “relaxed consistency” model

The Fork and Join Model



The Memory Model

- OpenMP uses a “relaxed consistency” model
- In contrast especially 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,)
```

Lesson 3 – the Rest of OpenMP

- More pragmas
  - Workshare (for loops, mutual exclusion)
  - Synchronization (barriers, atomics)
  - Initializing and sharing variables (memory)
- More runtime API functions
  - Setters, getters
  - Locking, timing routines
- More environmental variables
  - Somewhat related to the API

Pragmas Kinds

- Worksharing
  - #pragma omp **parallel**
  - #pragma omp parallel **for**
  - #pragma omp parallel **section**
- Mutual Exclusion
  - #pragma omp **critical**
  - #pragma omp critical (***name***)
- Singling-out Threads
  - #pragma omp **master**
  - #pragma omp **single**
- Synchronizations and Barriers
  - #pragma omp **barrier**

Nuances of OpenMP for Perl Programmers

- **`OpenMP::Environment`** will be discussed in Part II
- When an OpenMP executable starts, %ENV is read once and only once (at least for `gcc`)
- nota bena: for when we're discussing Part II of this talk regarding

Considerations for Perlers

- Creating library interfaces to compiled code (e.g., via `Inline::C`, XS, FFIs)
- Calling external executables (i.e., `system`, `qx//`, ``` `` ```)
- Perlish ways to manipulate %ENV
- Opportunities for use in **`perl`**, itself

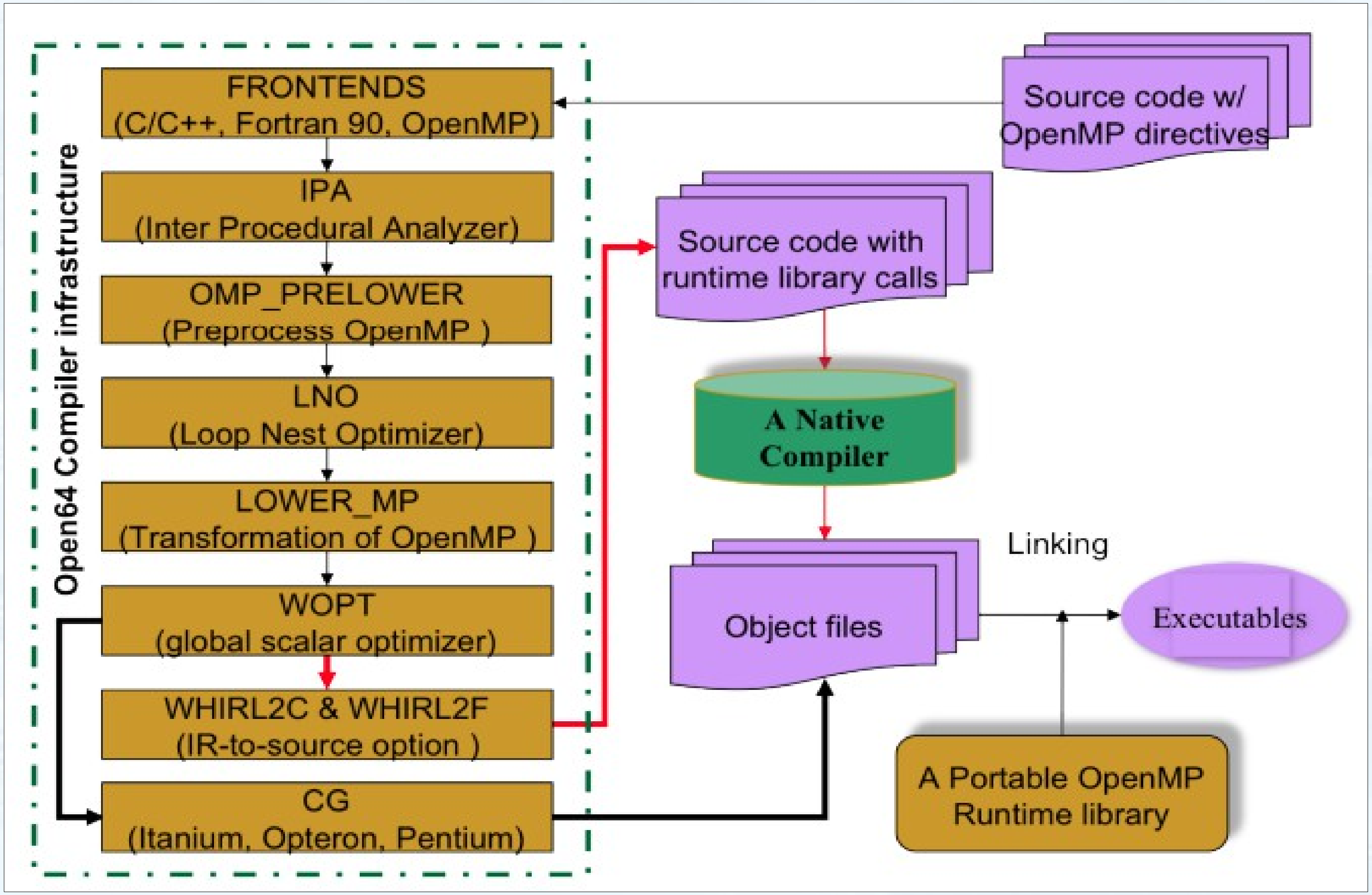
Stay tuned for *Part II!*

Time Permitting ...

- Questions
- Bonus OpenMP material

Bonus – Compilers* Support OpenMP

(* "*openuh,*" not **gcc**)



Bonus – Compilers* Support OpenMP

(* "*openuh,*" not **gcc**)

- 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()`

Bonus – Compilers* Support OpenMP

(* “*openuh*,” not **gcc**)

```
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

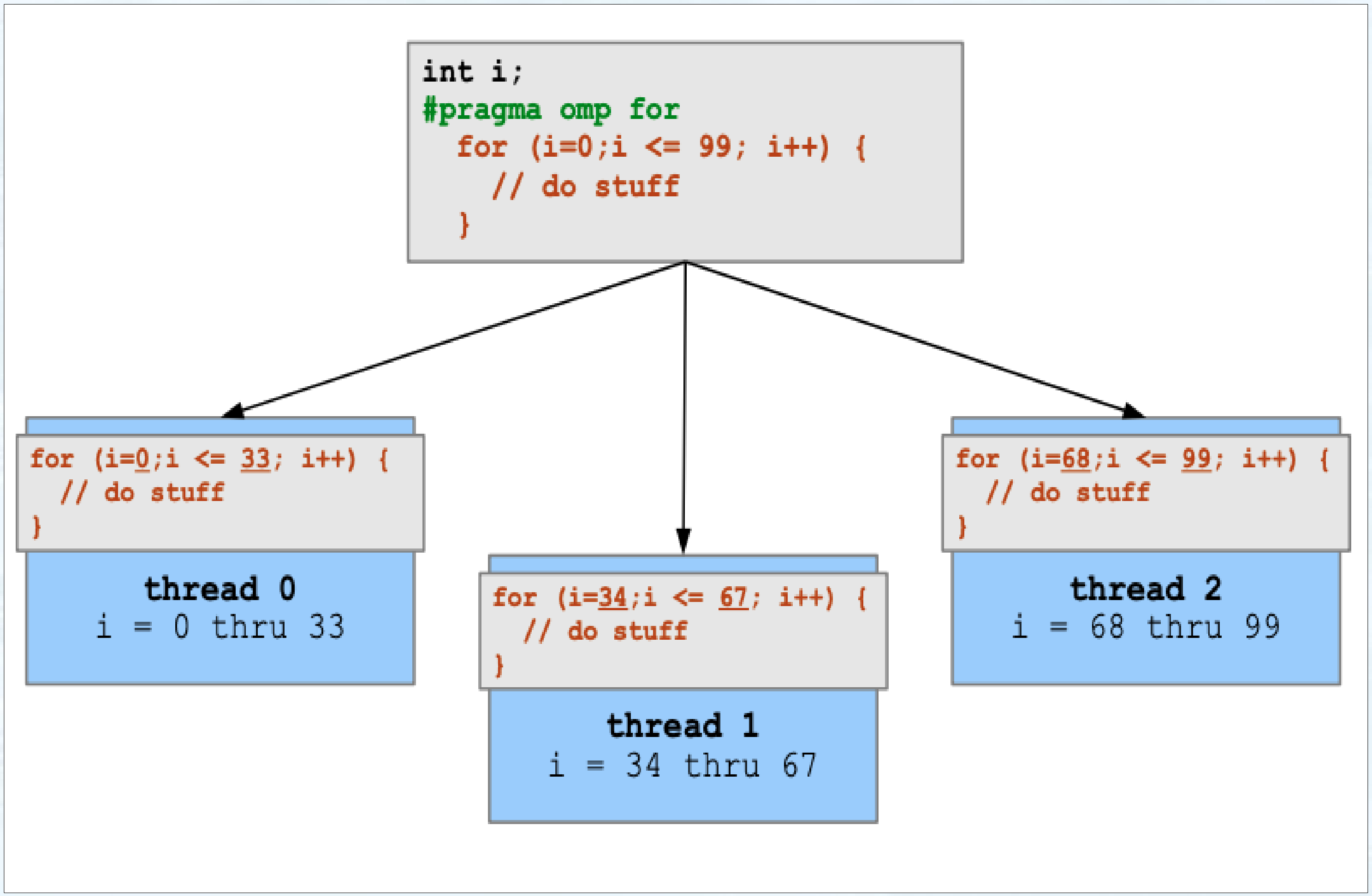
No body wants to code like this, so let the compiler and runtime do most all this tedious work!

HPCTools

Bonus – Loops in 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]);
}
```

Bonus - Loops


int i;
#pragma omp for
for (i=0;i <= 99; i++) {
// do stuff
}
for (i=0;i <= 33; i++) {
// do stuff
}
thread 0
i = 0 thru 33
for (i=34;i <= 67; i++) {
// do stuff
}
thread 1
i = 34 thru 67
for (i=68;i <= 99; i++) {
// do stuff
}
thread 2
i = 68 thru 99

Bonus – 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

Bonus – Loop Scheduling

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];
 }
```

Bonus – Loops w/ Ordered Sections

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

Bonus – Loop Collapsing

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
      }
  }
```

Bonus – tasks: producer/consumer

- One thread acts as producer
- All threads act as consumers
- ..code example, next slde

```c
#include <stdio.h>
#include <omp.h>
int counter = 0;
#pragma omp threadprivate(counter)
void inc_count() {
  counter++;
}
int main() {
  int tid;
  #pragma omp parallel private(tid)
  {
    // one thread will generate tasks
    #pragma omp single
    {
      // task body, will get picked up by available
      // worker threads
      int i;
      for (i = 1; i<=10; i++) {
        #pragma omp task
        {
          int j;
          tid = omp_get_thread_num();
          for (j = 1; j<=tid; j++) {
            inc_count();
          }
          printf("Counter %d is now: %d\n",tid,counter);
        }
      }
    }
  }
  return 0;
}
```

OpenMP Tasks -
“producer/consumer” model

Bonus – Reductions

Supported by parallel and worksharing constructs

- `parallel, for, do, sections`

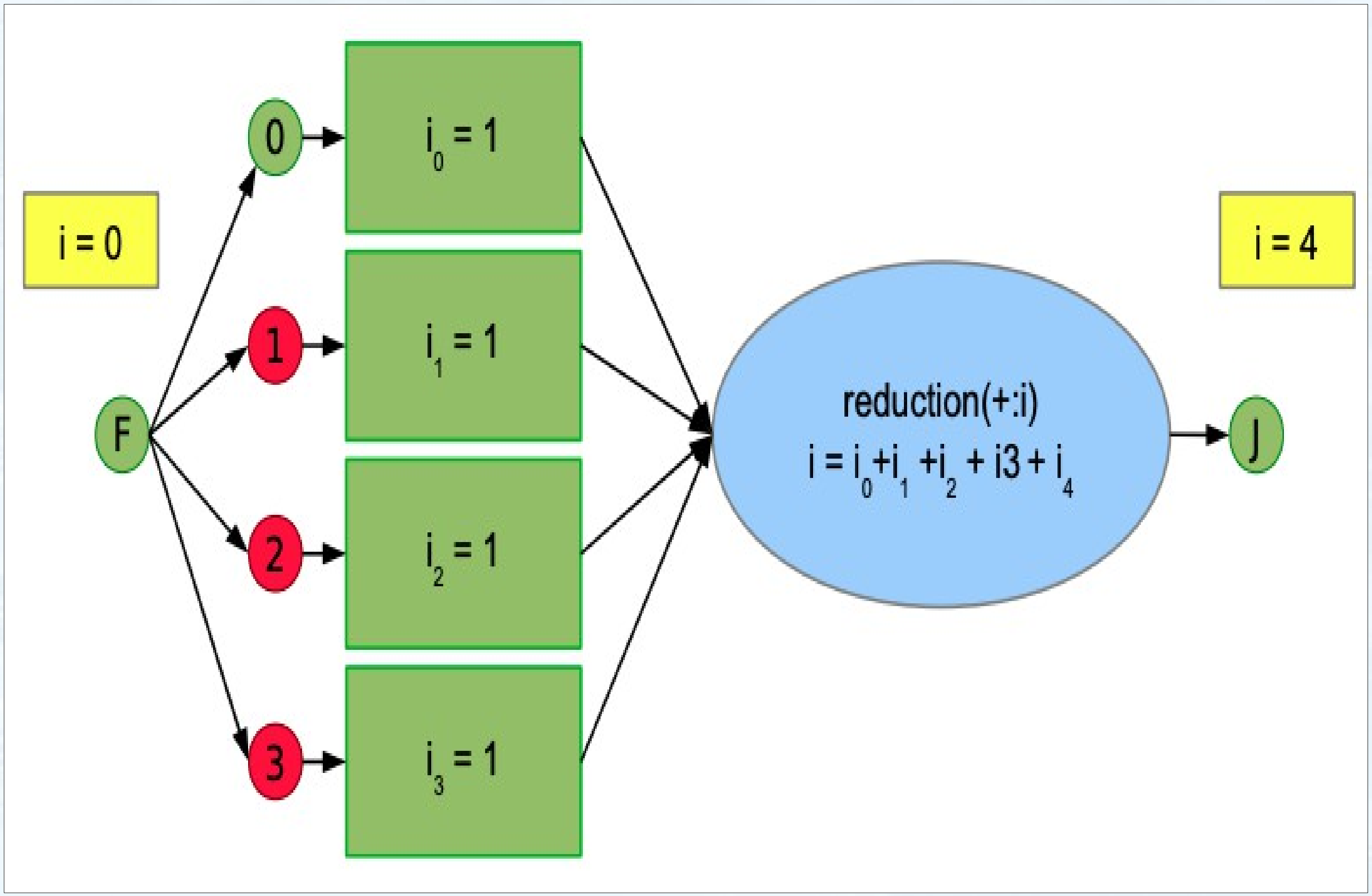
Creates a private copy of a shared var for each thread

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)
```

Bonus – Reductions



Bonus – Reductions

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

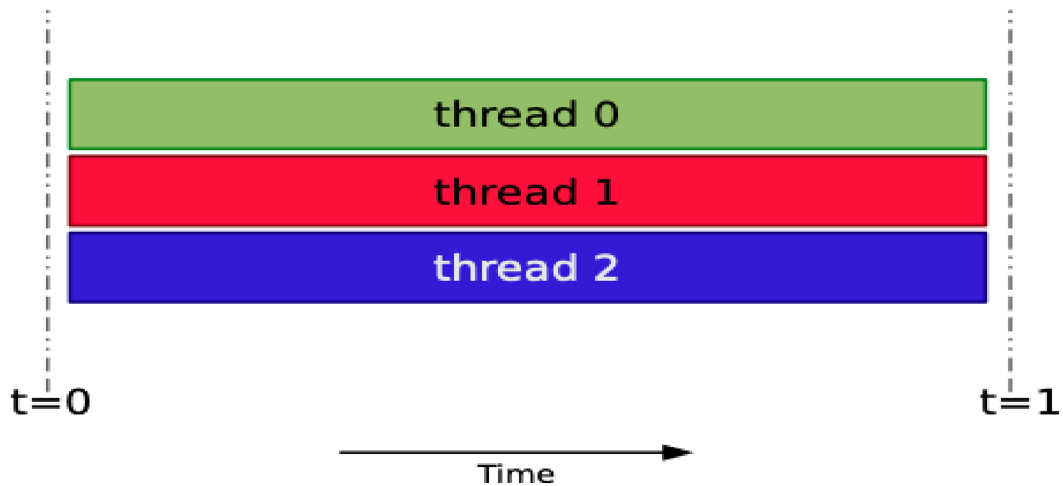
Note: initialized value matters!

Bonus - Sections

```c
#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;
}
```

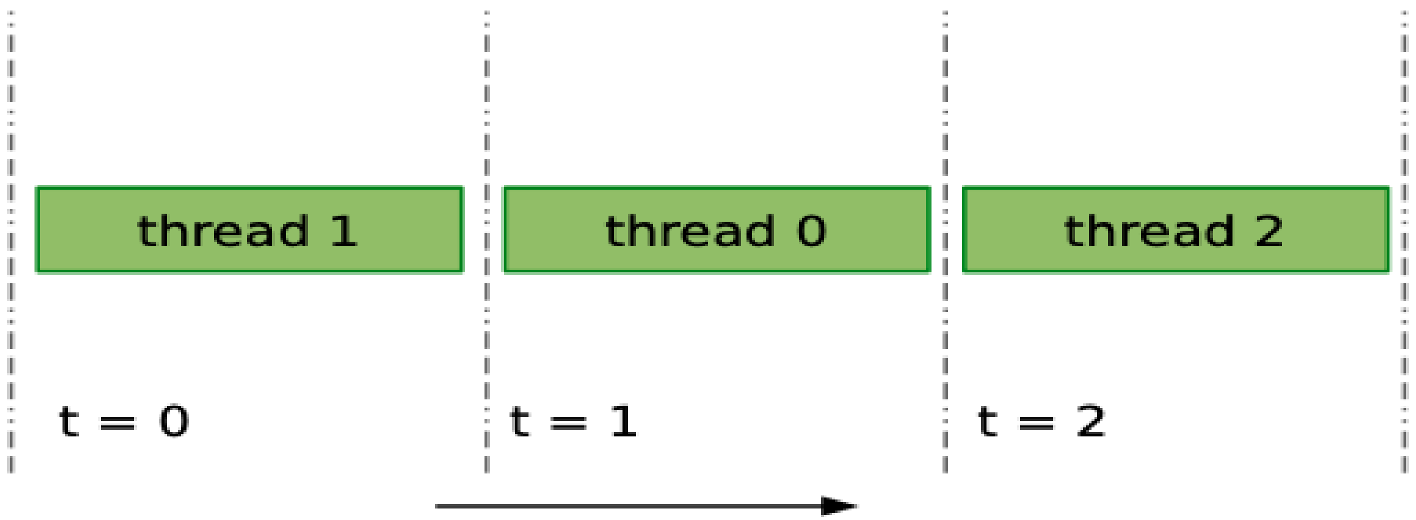
Bonus – Sections Execution Model

```
#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);
      }
   }
```



Bonus – Critical Sections (mutex)

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



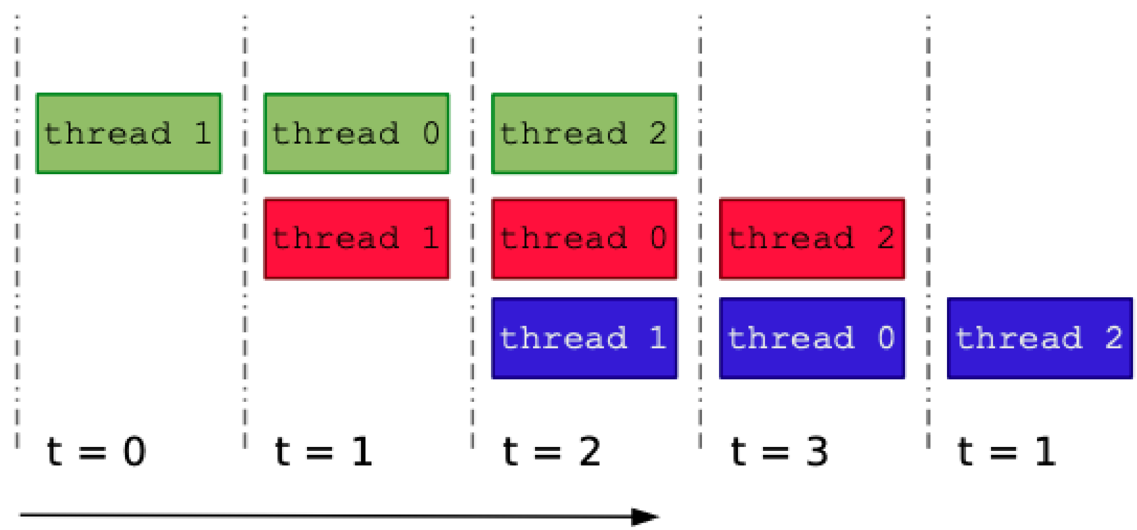
Bonus – Named Critical Sections

```
#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]);
    }
  }
}
```

Bonus – Named Critical Sections

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

Note:
Encountering thread
order not gauranteed!



Bonus – Atomic Updates

Protected writes to shared variables

Lighter weight than using a `critical` contruct

```c
#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);
}
```

Fin.