Today: Gray Codes; OpenMP

Acknowledgements: OpenMP Tutorial (LLNL, XSEDE) and
OpenMP by Example, Texas A&M

Gray Code (Embedding Topologies)

Hypercube Embedding

Parallelized algorithms benefit from predictive communication pattern

→ Need a Logical Topology

Ideally Logical Topology = Physical Topology

A topology X can be embedded in a topology Y, such that every comm. link in X, has a corresponding link in Y. This is called Topological Embedding.

Tianhe-2 → Fat Tree

Titan → 3D Torus

Sequoia → 5D Torus

RIKEN → Tofu (6DTorus/Mesh)

Mira → 5D Torus

Gray Code

- Sequence of numbers, where successive numbers have a Hamming Distance of 1
- Defined as follows:

G(k) → The k bit Grey Code. Defined recursively

G(1) → 0, 1

G(k+1) → Defined through a set of ops on G(k)

1. Append 0, on the left, to all members of G(k)
2. Reverse G(k) and Append 1, on the left of this new series
3. G(k+1) = concatenate(Step 1, Step 2)

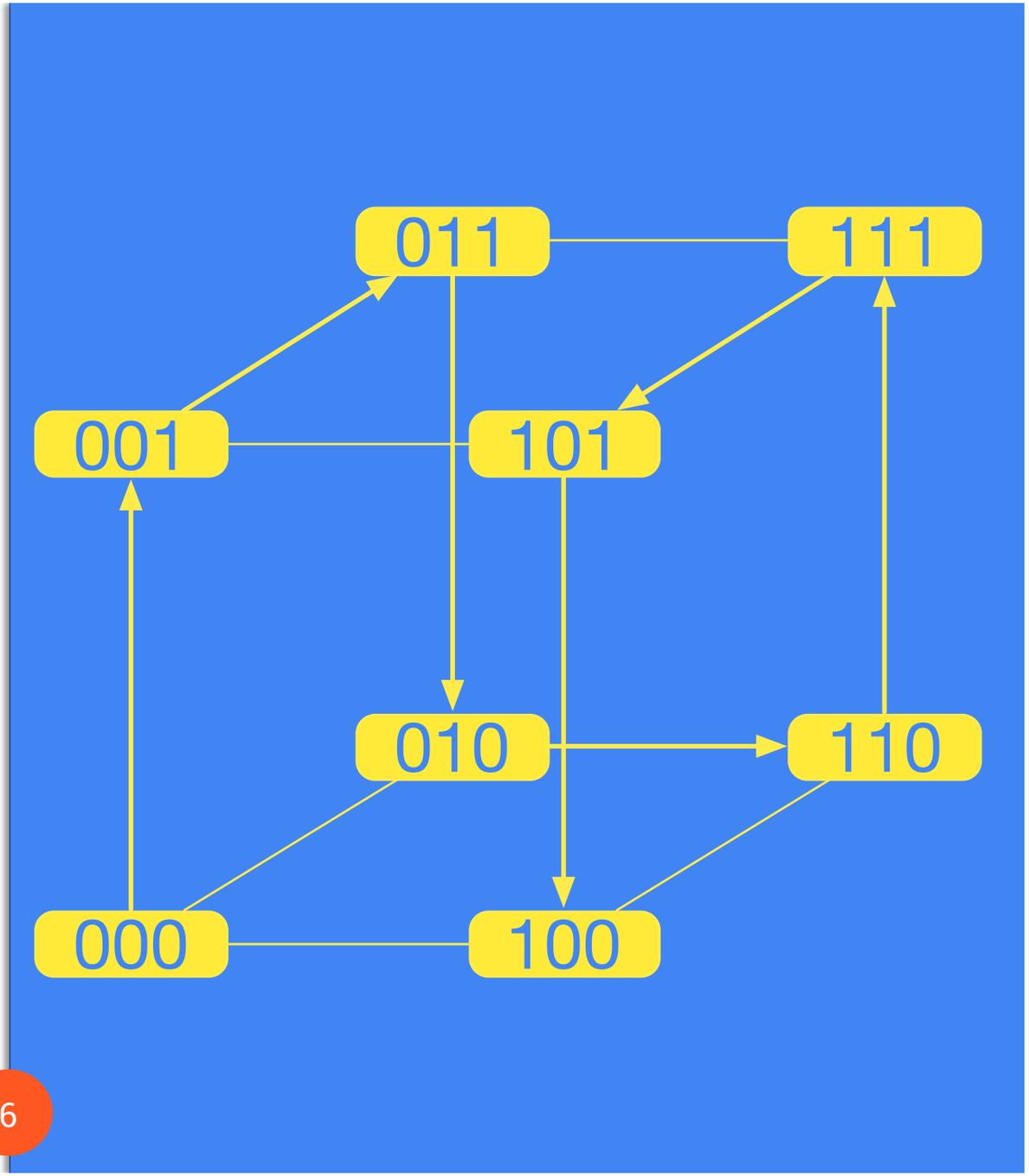
Example:

- G(1) → 0, 1
- G(2) →
  1. k = 1
  2. G(k) = 0, 1
  3. **Append 0**: 00, 01
  4. **Reverse G(k) and Append 1**: 11, 10
  5. **Concatenate**: 00, 01, 11, 10

- G(3) →
  1. k = 2
  2. G(k) = 00, 01, 11, 10
  3. **Append 0**: 000, 001, 011, 010
  4. **Reverse G(k) and Append 1**: 110, 111, 101, 100
  5. **Concatenate**: 000, 001, 011, 010, 110, 111, 101, 100

Line Topology into Hypercube

Similarly embed rings and torus into a hypercube.



OpenMP

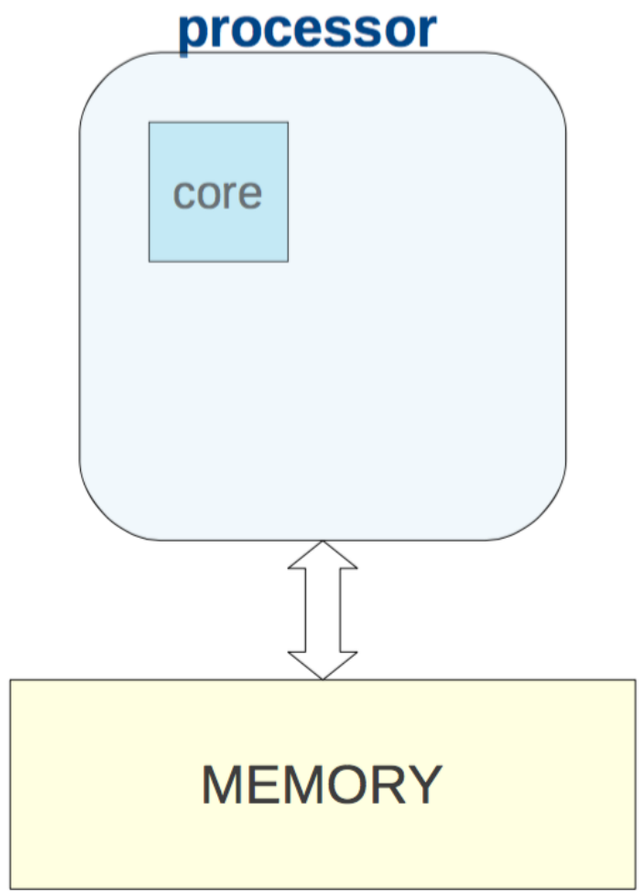
Single Processor

Traditional processors had a single core

→ 1 Core == 1 Processor

Core/Processor could exploit ILP, but that's it.

Developer relied on POSIX threads



Modern Processors

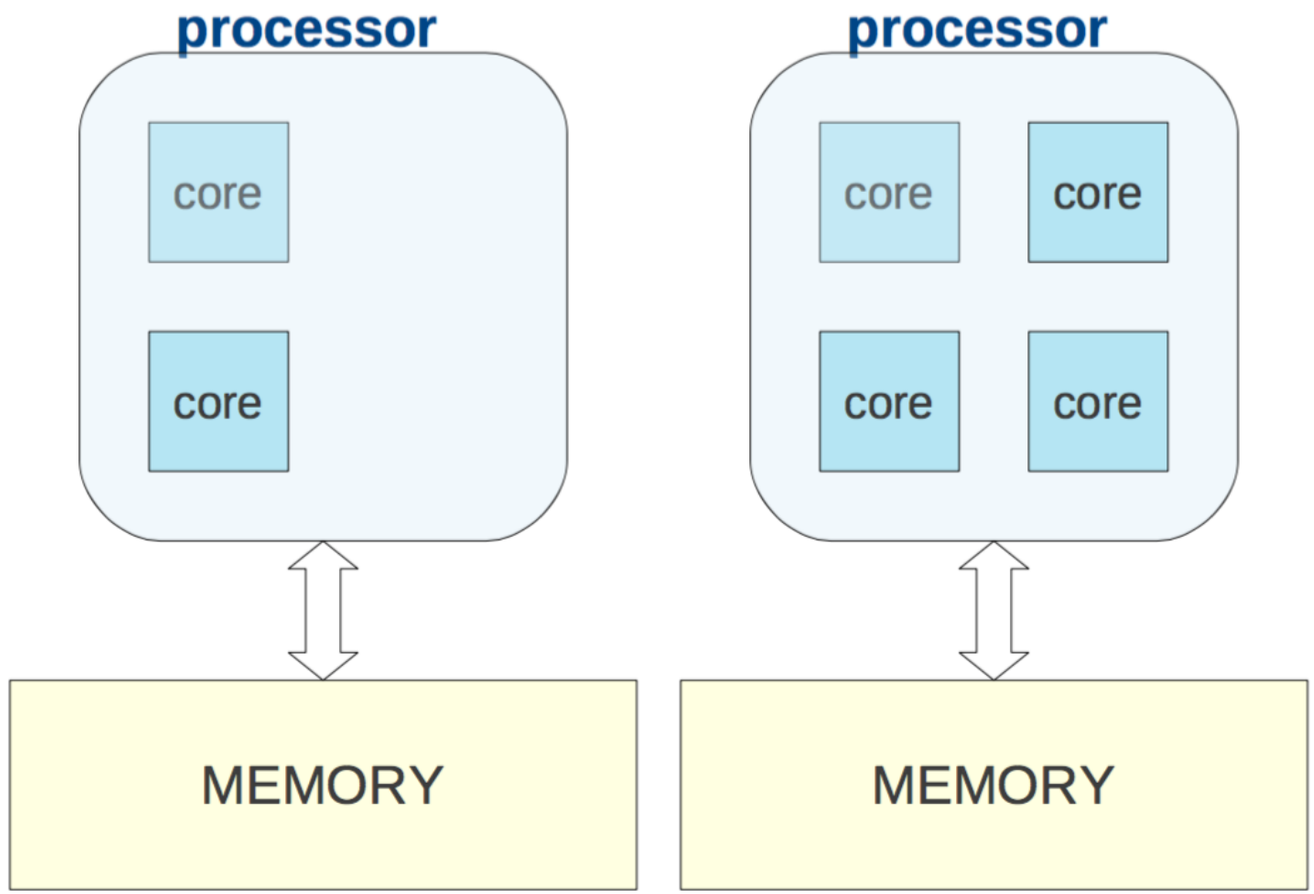
Modern Processors have multiple cores

>1 Core == 1 Processor

Cores work independently

sysctl -n hw.ncpu

cat /proc/cpuinfo



Intel i5 (Most Macbook Air)

Intel i7 (Most Macbook Pro)

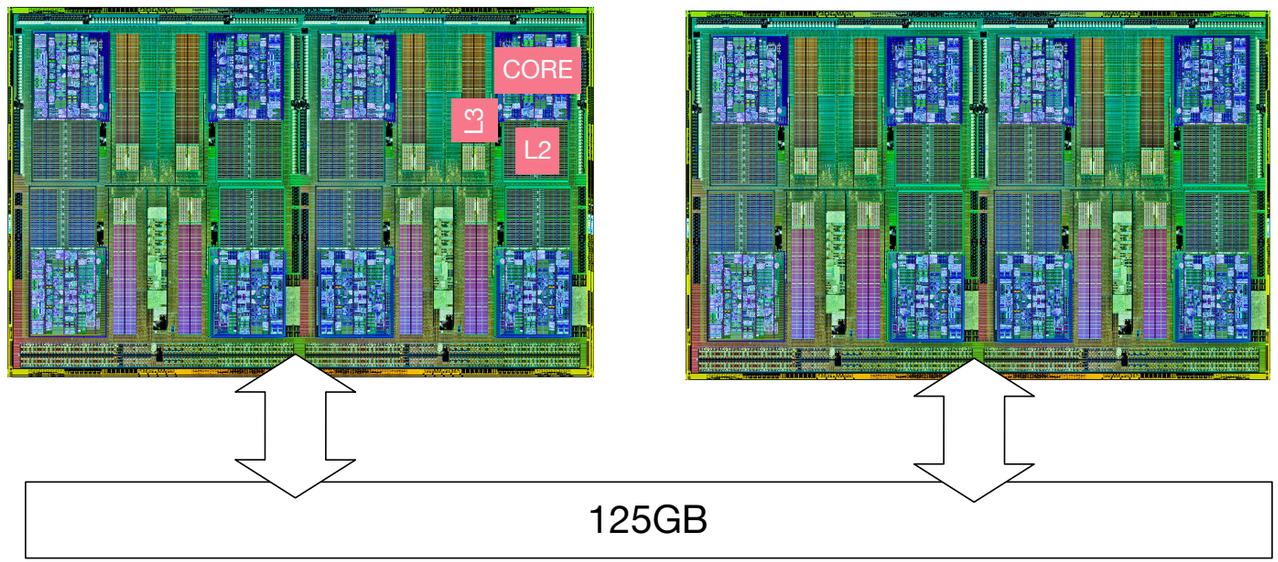
CCI Cluster

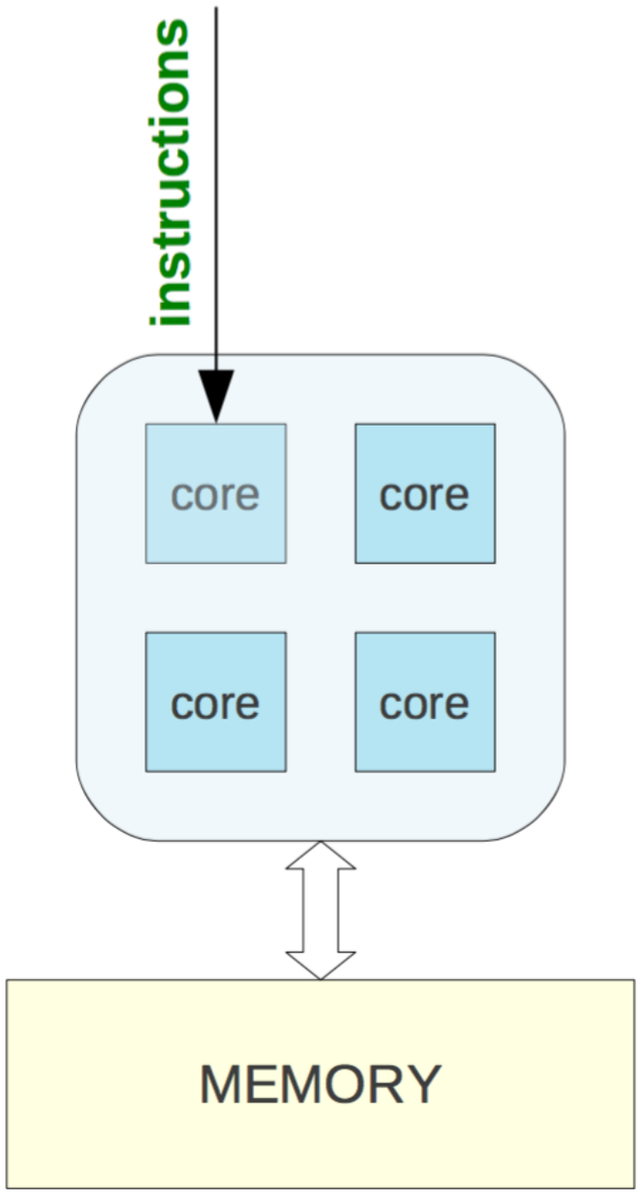
16 Cores/Processor

2 Processors/Node

11 Nodes → 352 Cores

128GB/Node → 1.4TB





Sequential Code

1 Core is Busy
Other cores are Idle

Why is this bad ☹ ?

What are some alternatives?

1. POSIX Threads
   Hello World →

```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 08/09/11
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS  5

void *PrintHello(void  *threadid)
{
  long tid;
  tid = (long)threadid;
  printf("Hello  World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main(int  argc, char *argv[])
{
  pthread_t  threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0;t<NUM_THREADS;t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t],  NULL, PrintHello,  (void *)t);
    if (rc){
      printf("ERROR;  return code from pthread_create()  is %d\n", rc);
      exit(-1);
      }
    }

  /* Last thing  that main()  should  do */
  pthread_exit(NULL);
}
```

What are some alternatives?

2. Treat each core as a processor and use message passing

Portable: 1 core or 100,000 cores

Cons: ??

What are some alternatives?

3. OpenMP
Hello World →

```
#include <omp.h>
#include <stdio.h>
int main()
{
  #pragma omp parallel
  printf("Hello from thread %d,
          nthreads %d\n",
          omp_get_thread_num(),
          omp_get_num_threads());
}
```

OpenMP

- An Application Programming Interface (API)[1]
- Supports shared memory parallel programming

Consists of (in v4):

1. 44 Compiler Directives
2. 35 Runtime Library (Subroutines/Functions)
3. 13 Environment Variables

openmp.org/mp-documents/OpenMP3.1-CCard.pdf

OpenMP : Thread Based Parallelism

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- Thread → smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.

OpenMP : Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

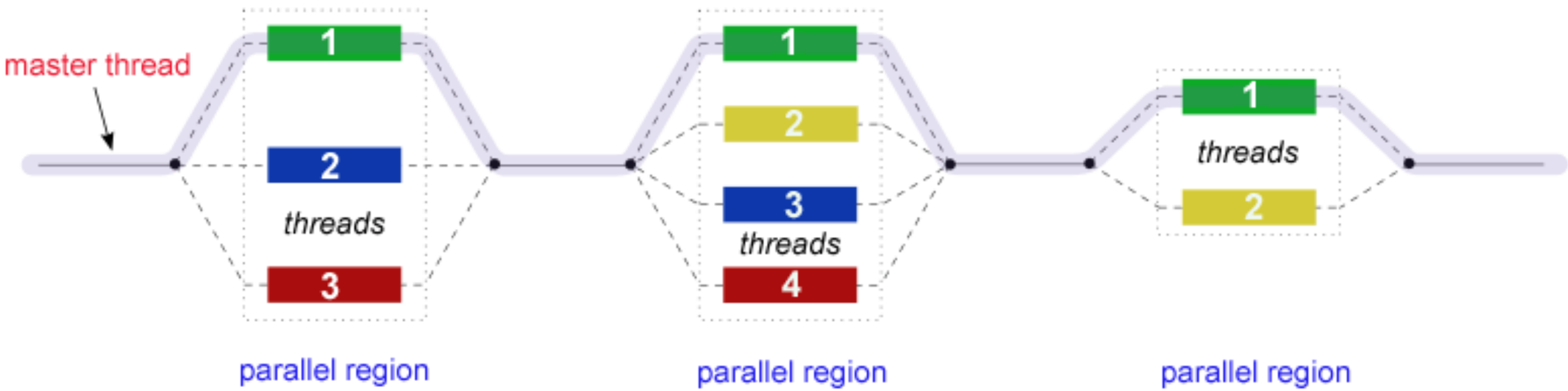
Features:

1. Compiler Based Directives: You specify the parts of the code you want to parallelize
2. Nested:
   - You can parallelize within a parallelized region
3. Dynamic Threads:
   - The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
   - Implementation may/may not support this

Features (cont...)

4. I/O
   - specifies nothing about parallel I/O
   - entirely up to you to ensure that I/O is conducted correctly within the context of a multi-threaded program.
   - Non-issue, if every thread conducts I/O to different file
5. Shared Memory
   - Provides a "relaxed-consistency" and "temporary" view of thread memory. i.e., threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
   - When it is critical that all threads view a shared variable identically, the dev is responsible for insuring that the variable is FLUSHed by all threads as needed.
   - More on this later...

Fork and Join:



Directive Format

```
#pragma omp parallel
{
    …
    …
}
```

```
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel

printf("Hello from thread %d,
        nthreads %d\n",
        omp_get_thread_num(),
        omp_get_num_threads() );
}
```

HelloWorld.c

Compiler Directive:

-fopenmp

OpenMP Threads

- Number of openMP threads can be set using:
  - Environmental variable OMP_NUM_THREADS
  - Runtime function omp_set_num_threads(n)

- Other useful function to get information about threads:
  - Runtime function omp_get_num_threads()
    - Returns number of threads in parallel region
    - Returns 1 if called outside parallel region
  - Runtime function omp_get_thread_num()
    - Returns id of thread in team
    - Value between [0,n-1] // where n = #threads
    - Master thread always has id 0

```
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel

printf("Hello from thread %d,
            nthreads %d\n",
            omp_get_thread_num(),
            omp_get_num_threads() );
}
```

Directive Format

```
#pragma omp parallel [clauses]
{
    …
    …
}
```

Available Clauses

if (scalar expr.) → determine whether the parallel construct will be used.

Evaluates to False if:

1. if the scalar expression in this clause evaluates to *false*
2. if the parallel construct appears while another parallel construct is active and the OpenMP nesting flag is off

Available Clauses

private (list) → followed by a list of variables that will be instantiated separately for each thread in the parallel region

- Every thread will have it's own "private" copy of variables in list
- No other thread has access to this "private" copy
- Value is undefined at the beginning of the parallel region
- Becomes undefined when parallel region ends.

firstprivate (list)→ Similar to private; The one difference is that instead of being undefined at the beginning of the parallel region, the value of each variable is initialized to the value it had when the parallel region was entered

shared (list) → followed by list of variables that are shared amongst all threads

- usually unnecessary because most variables are shared by default

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

int main(int argc, char* argv[])
{
    int niter = 1000000;            //number of iterations per FOR loop
    double x,y;                     //x,y value for the random coordinate
    int i;                          //loop counter
        int count=0;                //Count holds all the number of how many good coordinates
    double z;                       //Used to check if x^2+y^2<=1
    double pi;                      //holds approx value of pi
        int numthreads = 16;

    #pragma omp parallel firstprivate(x, y, z, i) shared(count) num_threads(numthreads)
    {
        srandom((int)time(NULL) ^ omp_get_thread_num());    //Give random() a seed value
        for (i=0; i<niter; ++i)              //main loop
        {
            x = (double)random()/RAND_MAX;      //gets a random x coordinate
            y = (double)random()/RAND_MAX;      //gets a random y coordinate
            z = sqrt((x*x)+(y*y));          //Checks to see if number is inside unit circle
        if (z<=1)
        {
                ++count;            //if it is, consider it a valid random point
        }
    }
        //print the value of each thread/rank
    }
        pi = ((double)count/(double)(niter*numthreads))*4.0;
        printf("Pi: %d\n", pi);

    return 0;
}
```


PI Example:

Available Clauses (cont…)

copyIn (list) → causes the listed variables to be copied from the master thread to all other threads in the team immediately after the threads have been created and before they do any other work

Available Clauses: Reduction

- Performs a reduction on the variables that appear in its list.
- The clause also specifies the operator for the reduction.

- A private copy of each reduction variable is created for each thread
  1. The private copies of the variables are updated as the threads execute
  2. And then the private copies from all threads are combined at the end of the parallel region.

C op	Initial value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

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

int main ()  {

int   i, n, chunk;
float a[100], b[100], result;

/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++)
  {
  a[i] = i * 1.0;
  b[i] = i * 2.0;
  }

#pragma omp parallel for default(shared) private(i) schedule(static,chunk) reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);

}
```


Reduction Example

Next…

1. OpenMP Continued…

2. Class Presentation: 20-25mins + 5-10min Q&A
3. Q&A → Presenter + Class
4. Homework 2 — OpenMP
    1. Get familiar with qsub and running simple hello world jobs.
    2. Will post sample codes in github
    3. Feel free to share any useful links on Slack