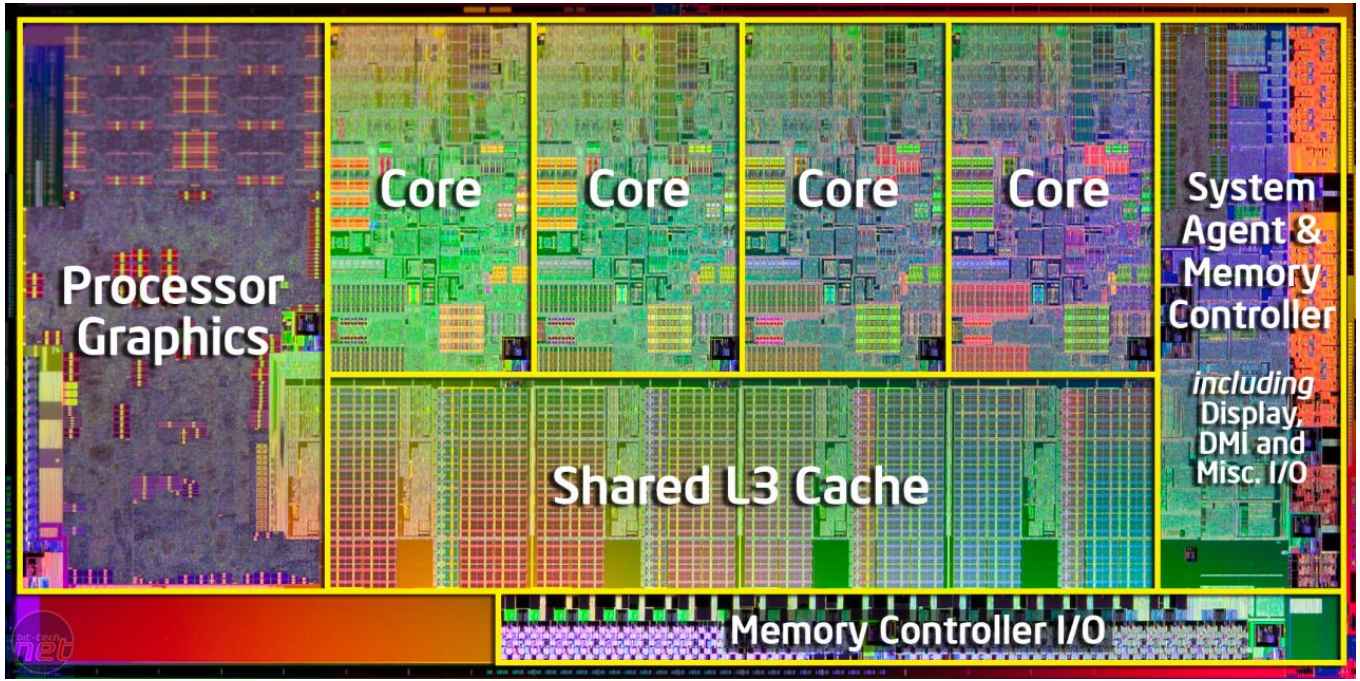
Intro to OpenMP

Our Favorite Picture

~6400 nodes are configured with two Xeon E5-2680 processors and one Intel Xeon Phi SE10P Coprocessor (on a PCIe card).

Another favorite...



Introduction

Each node of the Stampede machine contains 16 cores with shared access to 32 GB of memory. Now it’s time to take advantage of the cores about how to write programs that can effectively use these cores for concurrent computation using OpenMP.

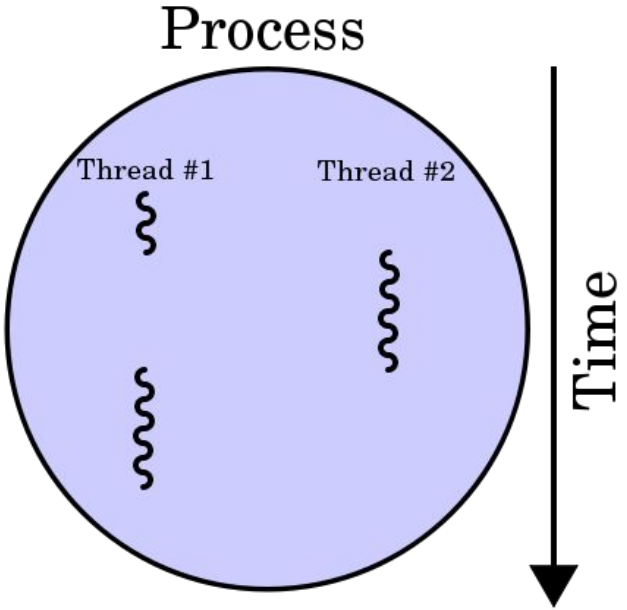
At the node level, you have several options for utilizing the 16 cores. You could run 16 separate tasks there, communicating among them using messages through the MPI interface, or you could run a single task that utilizes 16 threads for concurrent computation using the OpenMP interface, or you could use some combination of these two.

***Programs that require more than 16 cores will use several nodes of the Stampede machine. Since memory is not shared between nodes, MPI must be used for communicating between tasks on separate nodes, but these programs can use OpenMP to exploit the multiple cores within a node in a style that is called hybrid programming.

Introduction to Threads

...a thread is a component of a process. Multiple threads can exist within one process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given time).

On one processor, multithreading is generally implemented by time slicing (as in multitasking), and the central processing unit (CPU) switches between different *software threads*. This context switching generally happens often enough that users perceive the threads or tasks as running at the same time (in parallel). On a multiprocessor or multi-core system, multiple threads can be executed in parallel (at the same instant), with every processor or core executing a separate thread simultaneously.



Time Slicing

The period of time for which a process is allowed to run in a preemptive multitasking system is generally called the *time slice*, or *quantum*. The scheduler is run once every time slice to choose the next process to run. The length of each time slice can be critical to balancing system performance vs process responsiveness - if the time slice is too short then the scheduler will consume too much processing time, but if the time slice is too long, processes will take longer to respond to input.

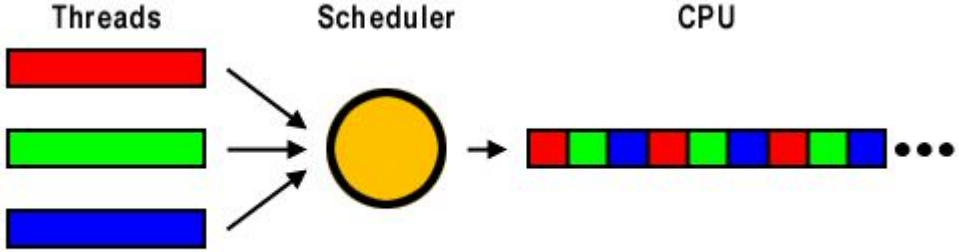
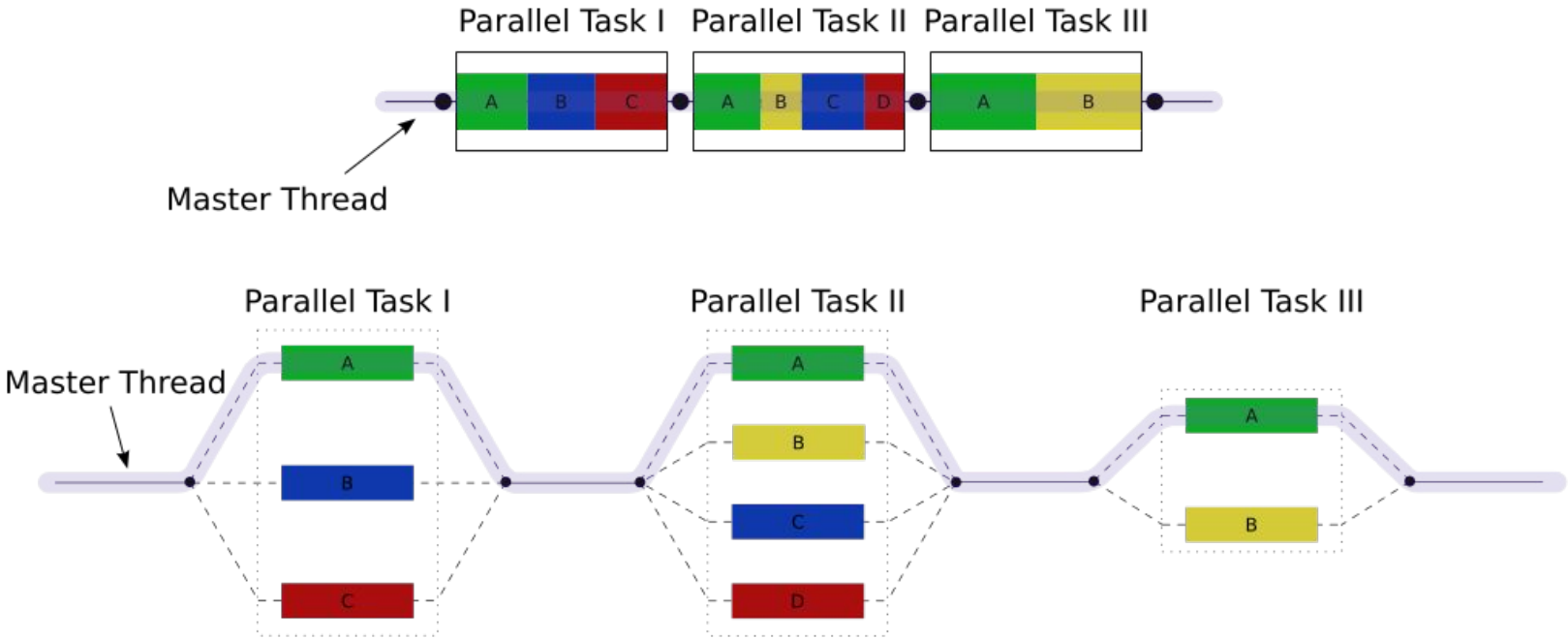


Fig. 1: Thread scheduling

Fork-Join Model

the **fork–join model** is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached.

Fork-Join Model



Putting those concepts together: Intro to OpenMP

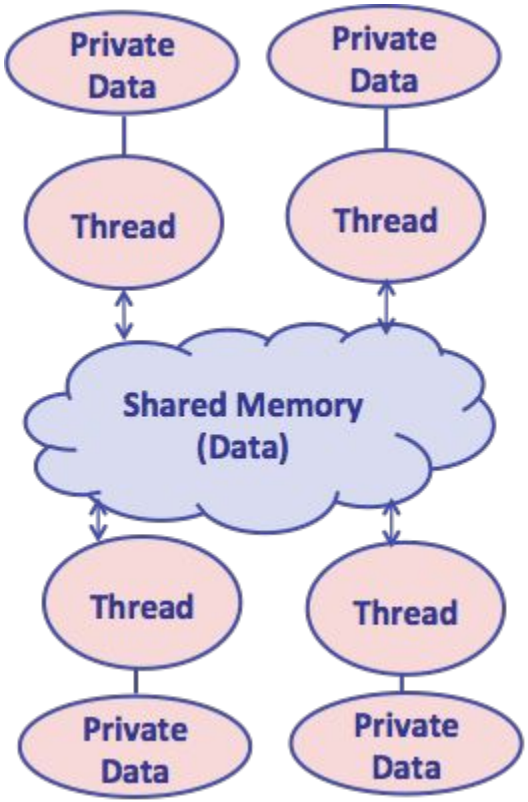
OpenMP is an implementation of **multithreading**, a method of parallelizing whereby a master *thread* (a series of instructions executed consecutively) *forks* a specified number of slave *threads* and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a **preprocessor directive** that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.

What is OpenMP?

- OpenMP stands for **Open Multi-Processing**
- An Application Programming Interface (API) for developing parallel programs for shared memory architectures
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Standard specifies C, C++, and FORTRAN Directives & API
- http://www.openmp.org/ has the specification, examples, tutorials and documentation

Architecture



- Data: shared or private
- Shared data: all threads can access data in shared memory
- Private data: can only be accessed by threads that own it
- Data transfer is transparent to the programmer

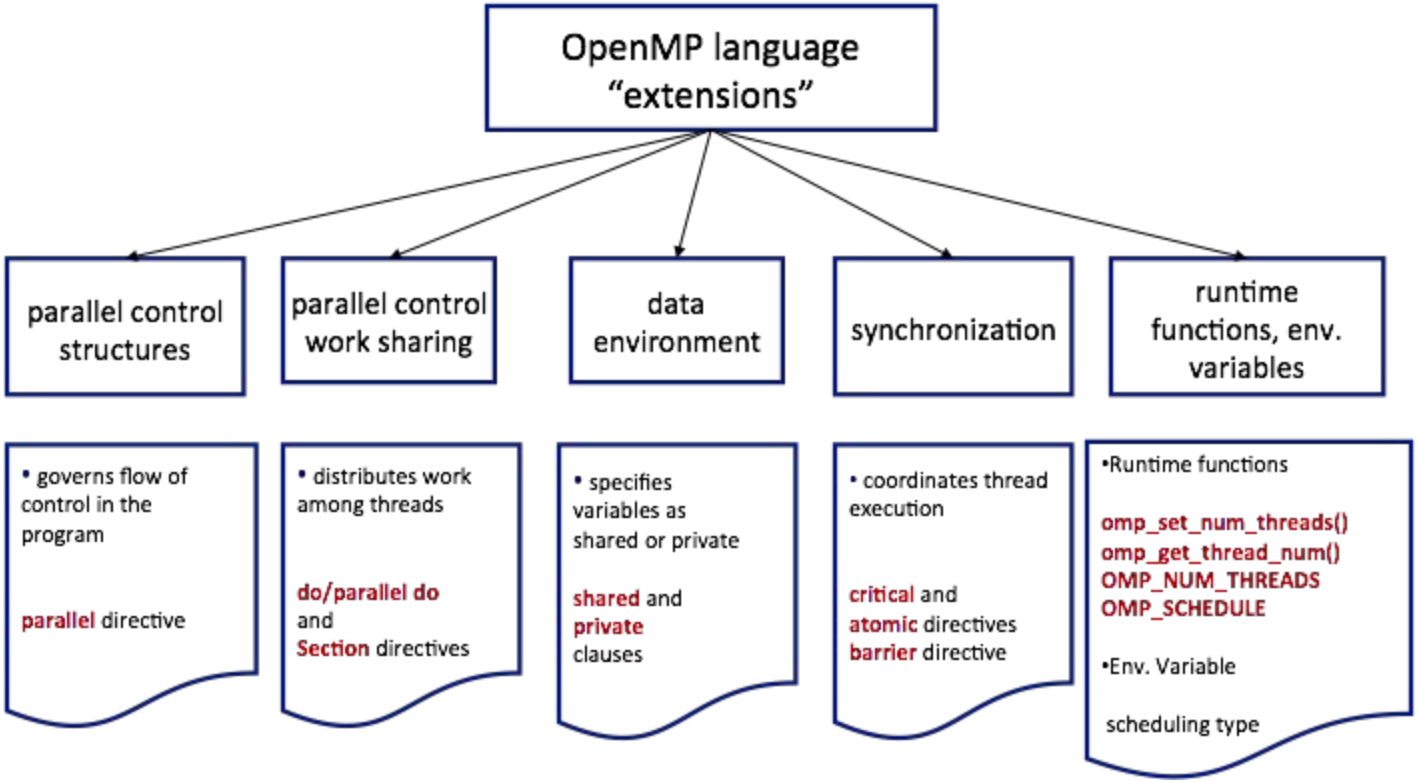
Round Up

- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process
- Each thread is given some private memory only seen by the thread
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually a bad idea. (But TS with user threads is less expensive than TS with processes)
- Implementation of threads differs from one OS to another

How do the Threads communicate?

- Every thread has access to “global” memory (shared)
- All threads share the same address space
- Threads communicate by reading/writing to the global memory
- Simultaneous updates to shared memory can create a *race condition*. Results change with different thread scheduling
- Use mutual exclusion to avoid data sharing - but don’t use too many because this will serialize performance

OpenMP Constructs



OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives

  #pragma omp *construct* [*clause* [[,]*clause*]…] **C**

  !$omp *construct* [*clause* [[,]*clause*]…] **F90**

- Example

  #pragma omp parallel num_threads(4) **C**

  !$omp parallel num_threads(4) **F90**

- Function prototypes and types are in the file:

  #include <omp.h> **C**

  use omp_lib **F90**

- Most OpenMP constructs apply to a "structured block", that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared memory machines

  FORTRAN : directives begin with the **!$OMP**, **C$OMP** or ***$OMP** sentinel.

  F90 : **!$OMP** free-format

  C/C++ : directives begin with the **# pragma omp** sentinel

- Parallel regions are marked by enclosing parallel directives
- Work-sharing loops are marked by parallel do/for

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel

!$OMP parallel do
  DO ...
!$OMP end parallel do
```

C/C++

```
# pragma omp parallel
  {...}


# pragma omp parallel for
  for(){...}
```

Parallel Regions

```
1   #pragma omp parallel
2           {
3           code block
4           work(…);
5           }
```

Line 1 Team of threads formed at parallel region

Lines 3-4 Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region

Line 5 All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads

Defining the Parallel Regions

Fortran

```
   ...
1  !$omp parallel
2        code statements
3        call work(…)
4  !$omp end parallel
```

C/C++

```
...
#pragma omp parallel
{  code statements
      work(…)
}
```

- Line 1: Team of threads formed.
- Lines 2-3: This is the parallel region
  - Each thread executes code block and subroutine call or function
  - No branching (in or out) in a parallel region.
- Line 4: All threads synchronize at end of parallel region (implied barrier)
- In example above, user must explicitly create independent work (tasks) in the code block and routine (using thread id and total thread count)

Parallel Region and Thread Number

Fortran

```
use omp_lib
   ...
   nt  = 1
!$omp parallel
   nt = omp_get_num_threads()
   call work(nt)
!$omp end parallel
```

C/C++

```
#include <omp.h>
  ...
   int nt=1;
#pragma omp parallel
{
   nt = omp_get_num_threads();
   ierr=work(nt);
}
```

Every thread can inquire the total number of threads (**nt** in line 4).

In a Nutshell.

OpenMP introduces parallelism into your application by launching a set of threads that execute portions of your code concurrently. There are mechanisms, described below, that determine how many threads are launched and what portion of your code and what portion of your data each thread will use. The threads that are launched on Stampede are pthreads, no different from the kind that you could launch explicitly with **`pthread_create()`**. The advantage of OpenMP is that you don't have to write any thread management code.

All OpenMP directives are inserted into your source code as comments. Thus, a compiler that knows nothing about OpenMP or a compiler that does but hasn't been given the appropriate flag will ignore them and compile your code as an ordinary serial program.

An OpenMP directive consists of a *sentinel*, which is interpreted by the compiler either as a comment or the beginning of an OpenMP directive, followed by an OpenMP construct and its parameters, if any.

The compiler flag that determines whether the OpenMP directives are interpreted by the compiler is**`-fopenmp`** for GCC 4.2 and later, and **`-openmp`** for the Intel icc compilers.

Hello World.

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

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

/* Fork a team of threads giving them their own
copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

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

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */

}
```