Introduction to OpenMP

Daniel Lucio

XSEDE
Extreme Science and Engineering Discovery Environment

A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Overview

- What is it?
- History
- Goals
- How it works
- Fork/Join model
- The API
- Core Elements
- OpenMP Directives
- Parallel construct
- Hello World Examples
- How to Compile
- Setting Number of Threads
- Darter Example
- Default Thread Affinity
- Computing π
- References

What is it?

OpenMP

- OpenMP stands for Open Multi-Processing.
- It is an API that supports shared memory multiprocessing programming in C, C++ and Fortran.
- OpenMP was defined by the OpenMP Architecture Review Board (OpenMP ARB), a group of vendors who joined forces during the latter half of the 1990s to provide a common means for programming a broad range of SMP architectures.
- OpenMP is not a new programming language! It can be considered as a "notation" that can be added to a sequential program in Fortran, C or C++ to describe how the work is to be shared among threads that will execute on different processors/cores, and to order accesses to shared data as needed.

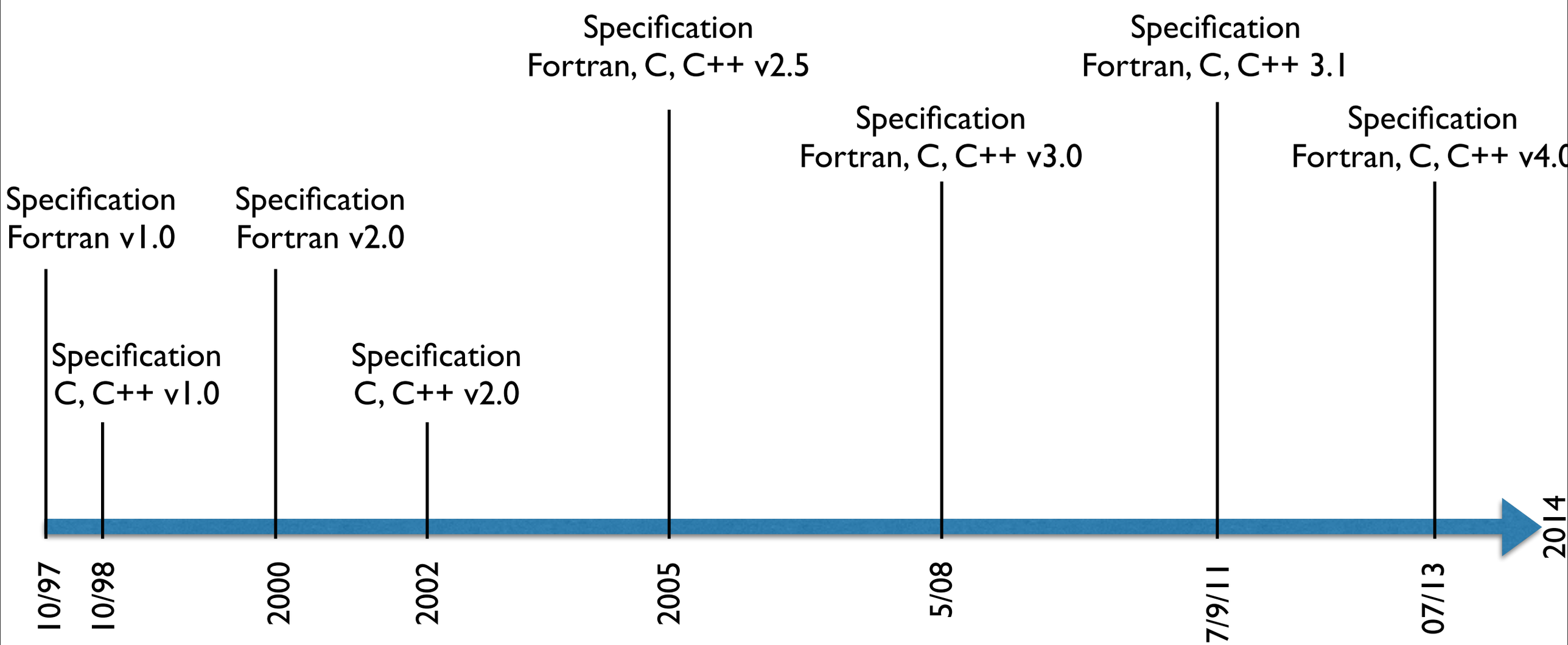
What is it?

OpenMP

- The OpenMP Application Programming Interface (API) was developed to enable portable shared memory parallel programming.
- An OpenMP directive is an instruction in a special format that is only understood by OpenMP-aware compilers.
- OpenMP offers a shared memory programing model, where most variables are visible to all threads by default

History

OpenMP



Goals

OpenMP

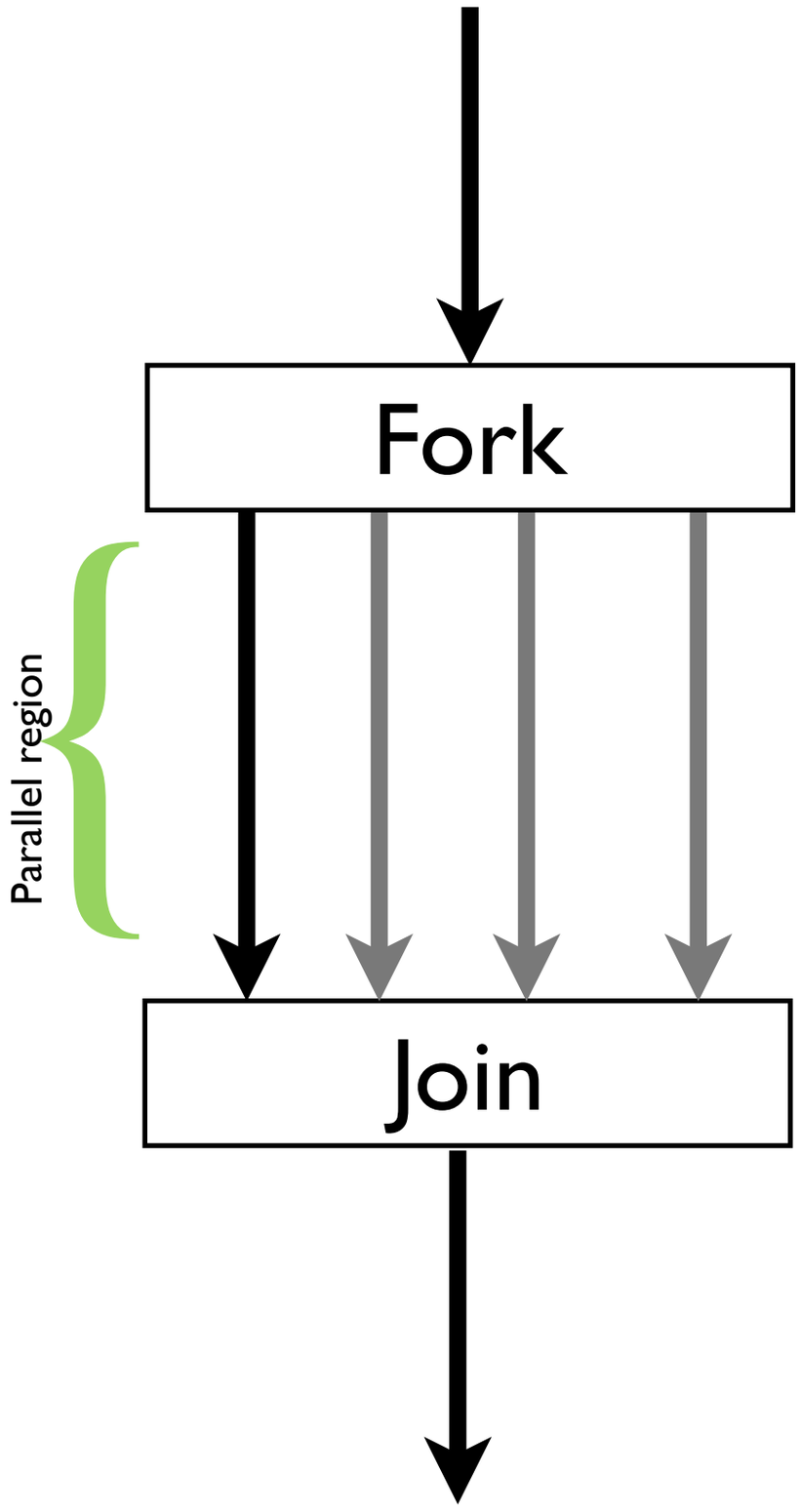
- OpenMP is designed to be a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications.
- The API is designed to permit incremental approach to parallelizing an existing code, possibly in successive steps. (Different to a all-or-nothing conversion as MPI).
- The impact of OpenMP parallelization is frequently localized, i.e. modifications are often needed in few places.
- It is possible to write the code such that the original sequential version is preserved.
- Each thread execute a parallelized section of code independently.

OpenMP

How it works

- OpenMP is an implementation of multithreading, where a master thread forks a specified number of slave threads and a task is divided among them.
- The main approach is based in identifying the parallelism in the program, and not in reprogramming the code to implement the parallelism.
- The section of code that is meant to run in parallel is marked with a preprocessor directive.
- Each thread has a unique identifier called a thread id, where the master thread has an id of 0.
- Both task and data parallelism can be achieved using OpenMP by using work-sharing constructs to divide a task.

Fork/Join Model

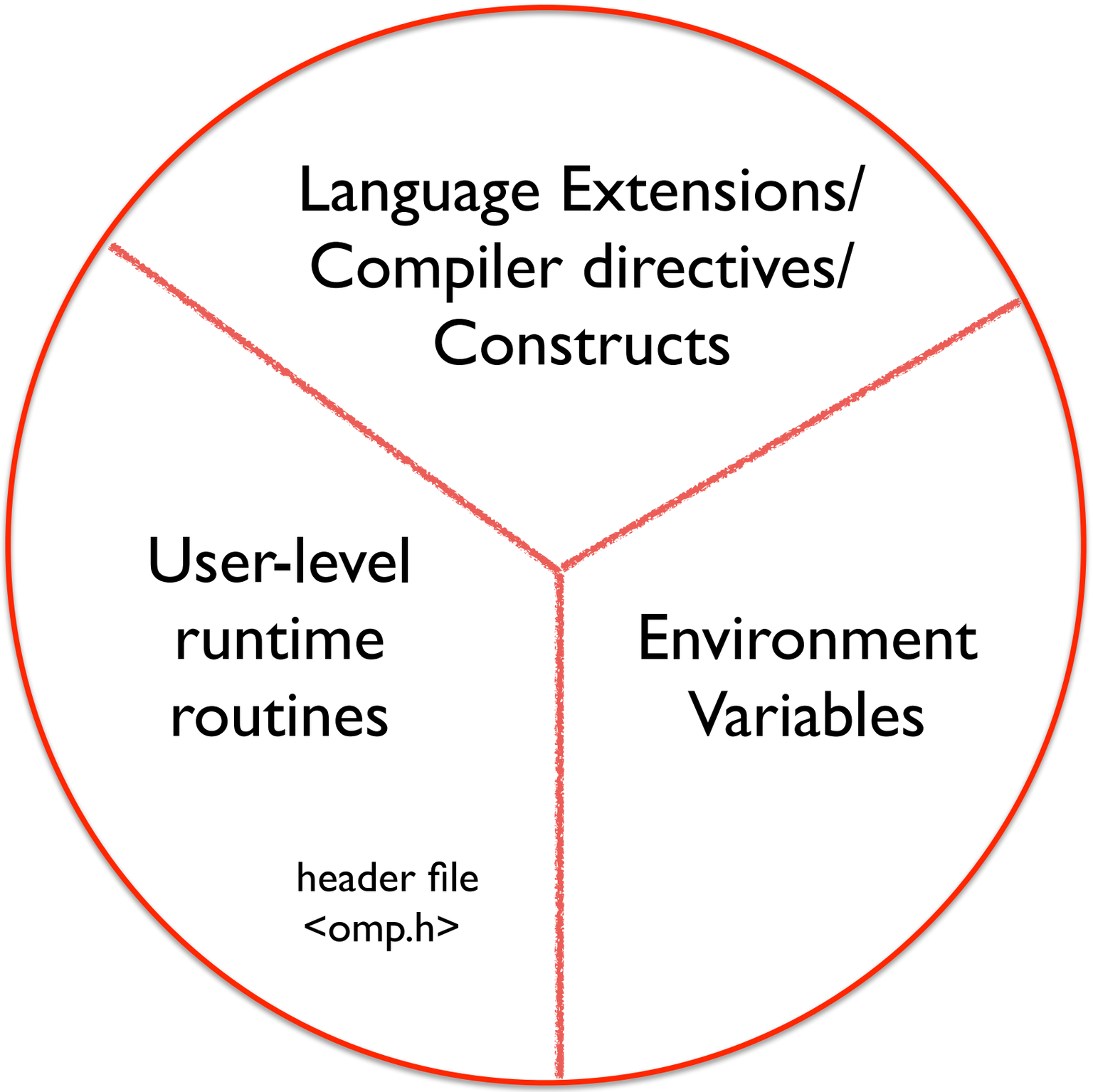


The program starts as a single thread just like a sequential program. The thread that executes this code is referred to as the *initial thread*.

Whenever an OpenMP *parallel* construct is encountered, a team of threads is created (fork). The initial thread becomes the master of the team. They all execute the code enclosed in the construct.

At the end of the construct, only the original thread continues and all other terminate (join).

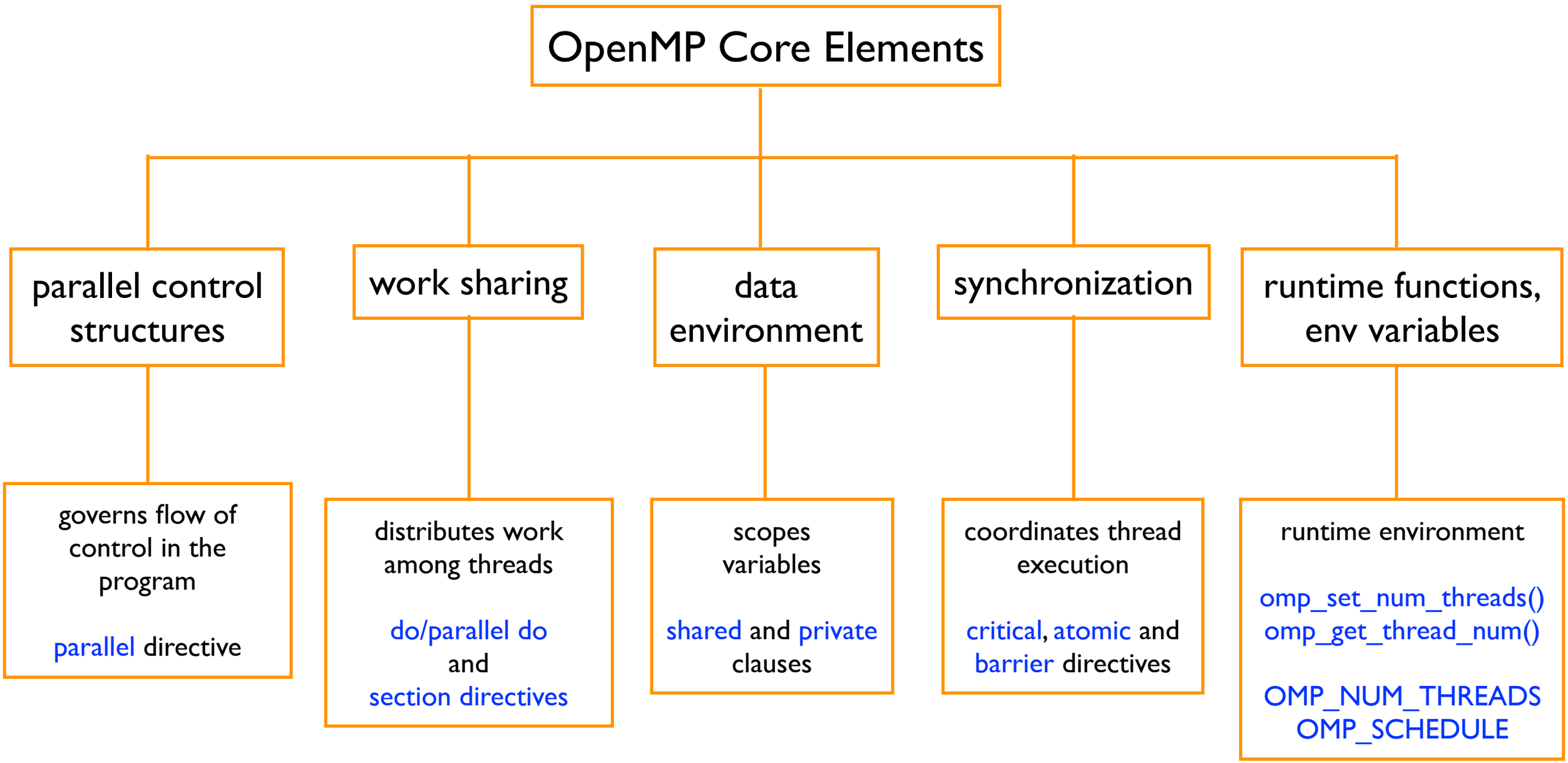
The API



The OpenMP API provides directives, library functions and environment variables to create and control the execution of parallel programs.

Core elements

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data environment variables, thread synchronization, user-level runtime routines and environment variables.



OpenMP Directive

General form of an OpenMP directive

C/C++ syntax

```
#pragma omp directive-name [clause[[,] clause]...] new-line
```

Note: In C/C++ directives are case sensitive!

Fortran syntax

```
!$omp directive-name [clause[[,] clause]...] new-line
c$omp directive-name [clause[[,] clause]...] new-line
*$omp directive-name [clause[[,] clause]...] new-line
```

Note: In Fortran all directives must begin with a directive sentinel. In fixed-source format Fortran there are three choices, but, the sentinel must start in column one. In free-source format only the first sentinel is supported.

Parallel construct

This construct is used to specify the code that should be executed in parallel. The code **not** enclosed by a parallel construct will **be** executed in serial!

C/C++ syntax

```
#pragma omp parallel [clause[[,] clause]...]
{
   /* parallel section */
}
```

Fortran syntax

```
!$omp parallel [clause[[,] clause]...]
   /* parallel section */
!$omp end parallel
```

Basic Hello World Example

OpenMP

Serial C

```c
#include <stdio.h>

int main()
{
  printf("Hello World!\n");

  return 0;
}
```

C

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

int main()
{
  #pragma omp parallel
  {
     printf("Hello World!\n");
  }

  return 0;
}
```

C++

```cpp
#include <omp.h>
#include <iostream>

int main()
{
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }

  return 0;
}
```

Fortran 77

```
C*************************
      PROGRAM HELLO
!$OMP PARALLEL
      PRINT *, 'Hello World!'
!$OMP END PARALLEL
      END
```

Fortran 90

```
!*********************************
program main
  implicit none
!$omp parallel
  write ( *, * ) '  Hello, world!'
!$omp end parallel
  stop
end
```

How to Compile?

OpenMP

GNU

```
$ gcc -fopenmp -o hello
hello.c
$ ./hello
Hello World!
Hello World!
```

PGI

```
$ cc -mp -o hello hello.c
$ ./hello
Hello World!
```

```
$ cc -mp=allcores -o hello hello.c
$ ./hello
Hello World!
Hello World!
```

CCE

```
$ craycc -o hello hello.c
$ ./hello
Hello World!
```

```
$ export OMP_NUM_THREADS=2
$ craycc -o hello hello.c
$ ./hello
Hello World!
Hello World!
```

Intel

```
$ icc -openmp -o hello hello.c
$ ./hello
OMP: Warning #72: KMP_AFFINITY: affinity only supported for Intel(R) processors.
OMP: Warning #71: KMP_AFFINITY: affinity not supported, using "disabled".
Hello World!
Hello World!
```

```
$ export KMP_AFFINITY=disabled
$ ./hello
Hello World!
Hello World!
```

The Intel® runtime library has the ability to bind OpenMP threads to physical processing units. The interface is controlled using the KMP_AFFINITY environment variable. This thread affinity interface is supported only for genuine Intel® processors.

Setting number of threads

hello_default.c

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

int main (int argc, char *argv[]) {
  int tid;
  printf("Hello world from Master thread:\n");
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hi there from thread: <%d>\n", tid);
  }
  printf("After parallel we are done!\n");

  return 0;
}
```

hello_custom.c

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

int main (int argc, char *argv[]) {
  int tid;
  printf("Hello world from Master thread:\n");
  #pragma omp parallel private(tid) num_threads(5)
  {
    tid = omp_get_thread_num();
    printf("Hi there from thread: <%d>\n", tid);
  }
  printf("After parallel we are done!\n");

  return 0;
}
```

Using number of cores available

```
$ gcc -Wall -fopenmp -o hello_default hello_default.c
$ ./hello_default
Hello world from Master thread:
Hi there from thread: <0>
Hi there from thread: <1>
After parallel we are done!
```

Using environment variables

```
$ export OMP_NUM_THREADS=4
$ ./hello_default
Hello world from Master thread:
Hi there from thread: <0>
Hi there from thread: <1>
Hi there from thread: <2>
Hi there from thread: <3>
After parallel we are done!
```

Using 'num_threads(NUM)' construct

```
[lucio@acai defNumThd]$ gcc -Wall -fopenmp -o
hello_custom hello_custom.c
[lucio@acai defNumThd]$ ./hello_custom
Hello world from Master thread:
Hi there from thread: <1>
Hi there from thread: <2>
Hi there from thread: <0>
Hi there from thread: <3>
Hi there from thread: <4>
After parallel we are done!
```

Darter example

Source Code:

hello_openmp.c

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

int main (int argc, char *argv[]) {
  int tid;
  printf("Hello world from Master thread:\n");
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hi there from thread: <%d>\n", tid);
  }
  printf("After parallel we are done!\n");

  return 0;
}
```

Job script:

hello_job.pbs

```
#PBS -l size=16
#PBS -l walltime=00:05:00
#PBS -A UT-SUPPORT

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=7

aprun -n1 -d7 ./hello_openmp
```

Darter example


```
lucio@darter1:~/openMP/darterExample> cc -o hello_openmp hello_openmp.c
lucio@darter1:~/openMP/darterExample> qsub hello_job.pbs
129787.sdb-darter.nics.utk.edu
lucio@darter1:~/openMP/darterExample> qstat 129787
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
129787.sdb-darter          hello_job.pbs    lucio           00:00:00 C batch
lucio@darter1:~/openMP/darterExample> ls
hello_job.pbs  hello_job.pbs.e129787  hello_job.pbs.o129787  hello_openmp*  hello_openmp.c
lucio@darter1:~/openMP/darterExample> cat hello_job.pbs.o129787
Hello world from Master thread:
Hi there from thread: <2>
Hi there from thread: <0>
Hi there from thread: <4>
Hi there from thread: <1>
Hi there from thread: <6>
Hi there from thread: <5>
Hi there from thread: <3>
After parallel we are done!
Application 216916 resources: utime ~0s, stime ~0s, Rss ~3540, inblocks ~7749, outblocks
~20940
```

Default Thread Affinity

Example run on Darter

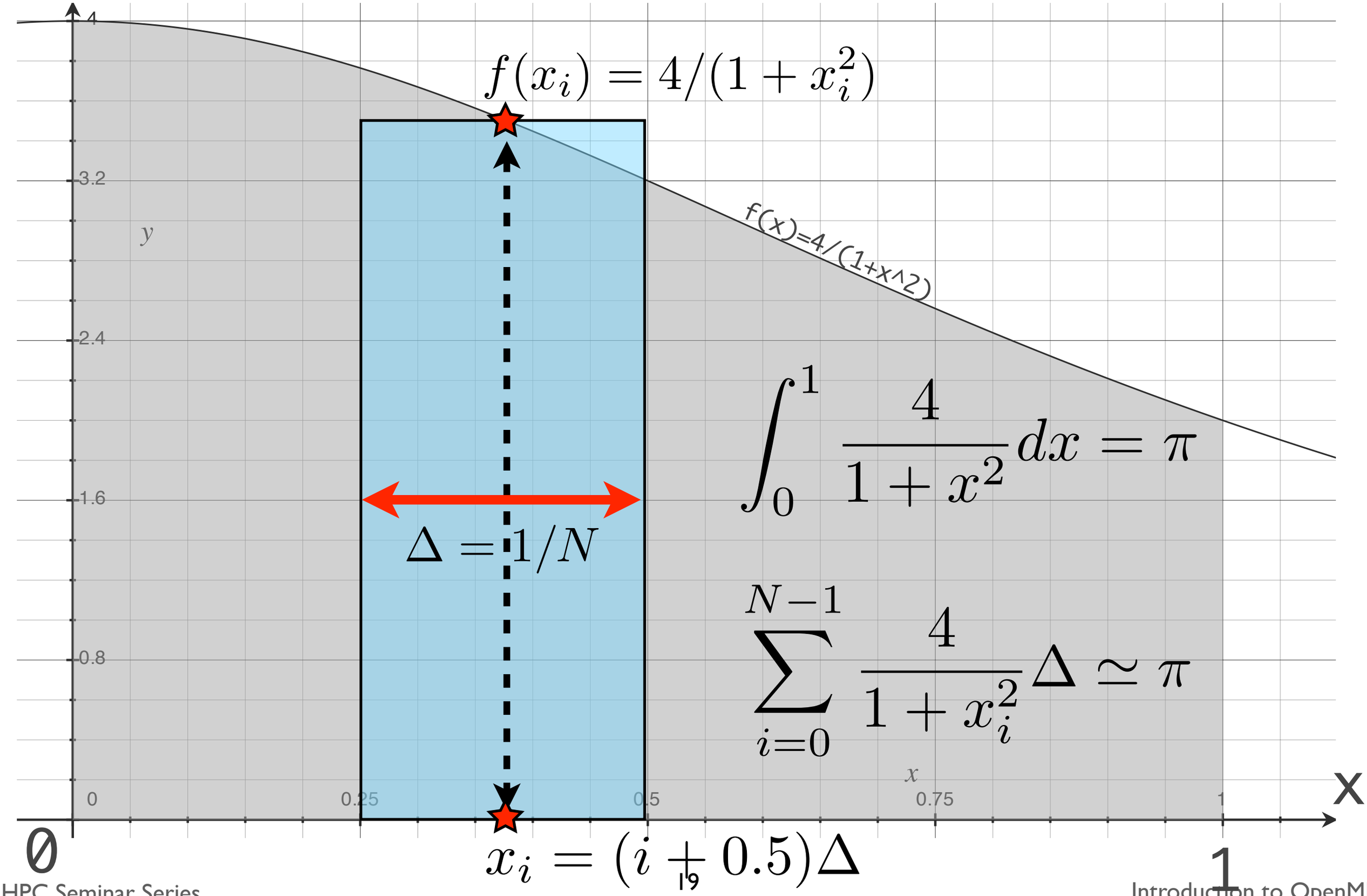
```
lucio@nid00129:~> aprun -n1 ./xthi
Hello from rank 0, thread 0, on nid00565. (core affinity = 0)


lucio@nid00129:~> OMP_NUM_THREADS=5 aprun -n1 ./xthi
WARNING: Requested total thread count and/or thread affinity may result in
oversubscription of available CPU resources!  Performance may be degraded.
Set OMP_WAIT_POLICY=PASSIVE to reduce resource consuption of idle threads.
Set CRAY_OMP_CHECK_AFFINITY=TRUE to print detailed thread-affinity messages.
Hello from rank 0, thread 0, on nid00565. (core affinity = 0)
Hello from rank 0, thread 2, on nid00565. (core affinity = 0)
Hello from rank 0, thread 3, on nid00565. (core affinity = 0)
Hello from rank 0, thread 4, on nid00565. (core affinity = 0)
Hello from rank 0, thread 1, on nid00565. (core affinity = 0)


lucio@nid00129:~> OMP_NUM_THREADS=5 aprun -n1 -d5 ./xthi
Hello from rank 0, thread 4, on nid00565. (core affinity = 4)
Hello from rank 0, thread 0, on nid00565. (core affinity = 0)
Hello from rank 0, thread 2, on nid00565. (core affinity = 2)
Hello from rank 0, thread 3, on nid00565. (core affinity = 3)
Hello from rank 0, thread 1, on nid00565. (core affinity = 1)
```

Computing π

f(x)



Serial π version


```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    sum += 4.0/(1.0+x*x);
  }
  pi = sum * step;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

Symbolically we know

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

which can be discretized

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \simeq \pi$$

$$x_i = (i+0.5)\Delta$$

$$\Delta = 1/N$$

```
$ pgcc -o pi pi.c
$ ./pi
Pi is 3.14159265360
```

First attempt to compute π

pi_omp1.c

```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel for
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    sum += 4.0/(1.0+x*x);
  }
  pi = sum * step;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

```
lucio@darter1:~/openMP/piValue> craycc -o pi_omp1 pi_omp1.c
lucio@darter1:~/openMP/piValue> OMP_NUM_THREADS=5 ./pi_omp1
Pi is 0.44262888469
```

Second attempt to compute π

pi_omp2.c

```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel for private(x) shared(sum)
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    sum += 4.0/(1.0+x*x);
  }
  pi = sum * step;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

```
lucio@darter1:~/openMP/piValue> craycc -o pi_omp2 pi_omp2.c
lucio@darter1:~/openMP/piValue> OMP_NUM_THREADS=5 ./pi_omp2
Pi is 0.44262888469
```

Third attempt to compute π

pi_omp3.c

```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel for private(x) shared(sum)
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    #pragma omp critical
    sum += 4.0/(1.0+x*x);
  }
  pi = sum * step;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

```
lucio@darter1:~/openMP/piValue> craycc -o pi_omp3 pi_omp3.c
lucio@darter1:~/openMP/piValue> OMP_NUM_THREADS=5 ./pi_omp3
Pi is 3.14159265360
```

A better way to compute π

pi_omp4.c

```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel for private(x) reduction(+:sum)
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    sum += 4.0/(1.0+x*x);
  }
  pi = sum * step;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

```
lucio@darter1:~/openMP/piValue> craycc -o pi_omp4 pi_omp4.c
lucio@darter1:~/openMP/piValue> OMP_NUM_THREADS=5 ./pi_omp4
Pi is 3.14159265360
```

Topics for 2nd part

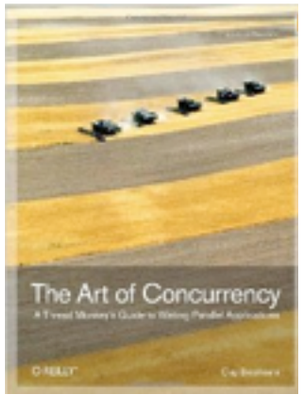
- Hybrid MPI+OpenMP
- More examples on data and task partitioning
- Tips and Tricks
- Overview of OpenMP 3.1 and 4.0
  - OpenMP Tasks
  - Accelerator support
  - Thread Affinity
  - SIMD constructs
- Fine-Tuning OpenMP codes
- Profiling OpenMP programs

Questions?

References

http://openmp.org

The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications
Clay Breshears



Using OpenMP: Portable Shared Memory Parallel Programming
Barbara Chapman, Gabriele Jost

Parallel Programming in C with Mpi and Openmp
Michael J. Quinn