Shared Memory Parallel Programming

Abhishek Somani, Debdeep Mukhopadhyay

Mentor Graphics, IIT Kharagpur

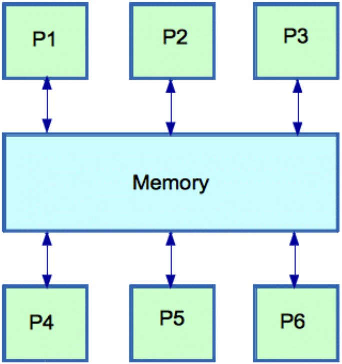
August 2, 2015

Overview

Outline

1. Introduction
2. Programming with pthreads
3. Programming with OpenMP

Programming Model



- CREW (Concurrent Read Exclusive Write) PRAM (Parallel Random Access Machine)
- Shared Memory Address Space

Requirements for Shared Address Programming

- **Concurrency** : Constructs to allow executing parallel streams of instructions
- **Synchronization** : Constructs to ensure program correctness
  - Mutual exclusion for shared variables
  - Barriers
- *Software Portability* : Across architectural platforms and number of processors
- *Scheduling and Load balance* : Efficiency
- *Ease of programming* : OpenMP versus pthreads

Fork-Join Mechanism

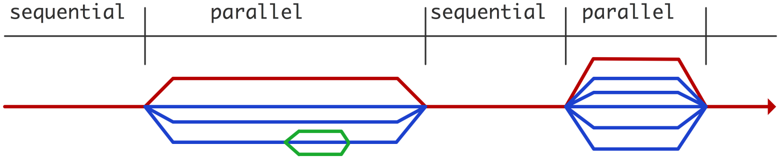


Figure : Courtesy of Victor Eijkhout

- Threads are dynamic
- Master thread is always active
- Other threads created by *thread spawning*
- Threads share data

process and thread

process

- separate address space
- heavyweight; context switching is expensive
- can consist of multiple threads
- independent of other processes
- not very different from serial programming

thread

- shared address space
- lightweight; hyperthreading support in modern hardware
- belongs to a process
- all threads of a process are interdependent
- requires careful programming for correctness and efficiency

Outline

1. Introduction
2. Programming with pthreads
3. Programming with OpenMP

POSIX threads or pthreads

```
// Necessary header
#include "pthread.h"
// Function to be called by each thread
void * thread_function(void * arg);
// Start Thread
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*thread_function) (void *),
                   void *arg);
// Stop Thread
int pthread_join(pthread_t thread,
                 void **retval);
```

pthread example 1

```c
#include <stdlib.h>
#include <stdio.h>
#include "pthread.h"

int sum=0; //Global variable touched by all threads

//Function to be called by each thread
void adder() {
    sum = sum+1;
    return;
}

int main() {
    const int numThreads=24;
    int i;
    pthread_t threads[numThreads];
    for (i=0; i<numThreads; i++) //Start threads
        if (pthread_create(threads+i, NULL, (void *)&adder, NULL) != 0)
            return i+1;
    for (i=0; i<numThreads; i++) //Stop threads
        if (pthread_join(threads[i], NULL) != 0)
            return numThreads+i+1;
    printf("Sum computed: %d\n",sum);
    return 0;
}
```

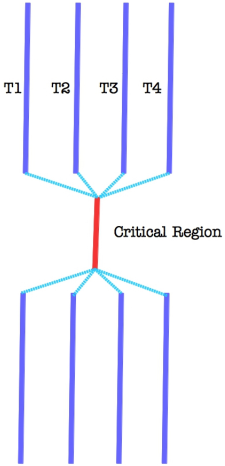
pthread example 1 while sleeping

```
//Function to be called by each thread
void adder() {
    int t = sum;
    sleep(1);
    sum = t + 1;
    return;
}
```

pthread example 1 while sleeping ...

```c
//Function to be called by each thread
void adder() {
    sleep(1);
    sum = sum + 1;
    return;
}
```

Critical Region



Lock and Key

Mutual Exclusion Locks $\Longleftrightarrow$ mutex locks

```
//The Lock
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);

//The Key
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);

//Initialization of Lock
int pthread_mutex_init (pthread_mutex_t *mutex_lock,
                        const pthread_mutexattr_t *lock_attr);
```

pthread example 1 with locks

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "pthread.h"

int sum=0; //Global variable touched by all threads
pthread_mutex_t lock; //Mutex lock

//Function to be called by each thread
void adder() {
    pthread_mutex_lock(&lock);
    int t = sum; sleep(1); sum = t + 1;
    pthread_mutex_unlock(&lock);
    return;
}

int main() {
    const int numThreads=24;
    int i;
    pthread_mutex_init(&lock, NULL);
    pthread_t threads[numThreads];
    for (i=0; i<numThreads; i++) //Start threads
        if (pthread_create(threads+i, NULL, (void *)&adder, NULL) != 0)
            return i+1;
    for (i=0; i<numThreads; i++) //Stop threads
        if (pthread_join(threads[i], NULL) != 0)
            return numThreads+i+1;
    printf("Sum computed: %d\n",sum);
    return 0;
}
```

Producer-Consumer work queues

```
pthread_mutex_t task_queue_lock; // Initialized in main
int task_available; //Initialized to 0 in main
```

producer

```
while (!done()) {
    inserted = 0;
    create_task(&my_task);
    while (inserted == 0) {
        pthread_mutex_lock(&task_queue_lock);
        if (task_available == 0) {
            insert_into_queue(my_task);
            task_available = 1;
            inserted = 1;
        }
        pthread_mutex_unlock(&task_queue_lock);
    }
}
```

consumer

```
while (!done()) {
    extracted = 0;
    while (extracted == 0) {
        pthread_mutex_lock(&task_queue_lock);
        if (task_available == 1) {
            extract_from_queue(&my_task);
            task_available = 0;
            extracted = 1;
        }
        pthread_mutex_unlock(&task_queue_lock);
    }
    process_task(my_task);
}
```

Types of mutexes

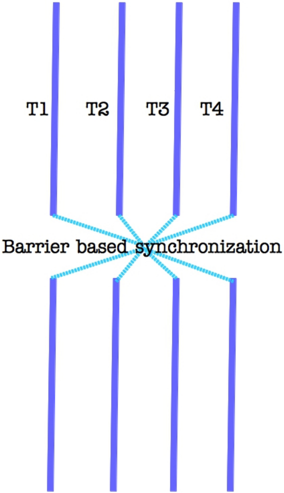
```
//Initialization of Mutex Attribute
int pthread_mutexattr_init (pthread_mutexattr_t *attr);

//Set type of Mutex
int pthread_mutexattr_settype_np (pthread_mutexattr_t *attr,
                                  int type);
```

- PTHREAD_MUTEX_NORMAL_NP : default, deadlocks on trying a second lock
- PTHREAD_MUTEX_RECURSIVE_NP : allows locking multiple times
- PTHREAD_MUTEX_ERRORCHECK_NP : reports an error on trying a second lock

Mutex Efficiency

- `pthread_mutex_trylock`
  - Faster than `pthread_mutex_lock`
  - Allows thread to do other work if already locked
- Condition Variables
  - Allows a thread to block itself until a pre-specified condition is satisfied
  - Thread performing condition wait does not use any CPU cycles
- Read-Write Locks
  - More frequent reads than writes on a data-structure
  - Multiple simultaneous reads can be allowed but only one write

Barriers



- Can be implemented using a counter, mutex or condition variable
- Threads wait at the barrier till all threads have reached
- Last thread to reach barrier wakes up all the threads

Famous words

- *A good way to stay flexible is to write less code* – Pragmatic Programmer
- *Simplicity is prerequisite for reliability* – Dijkstra
- *Any fool can write code that a computer can understand. Good programmers write code that humans can understand* – Martin Fowler
- *Programming can be fun, so can be cryptography; however they should not be combined* – Kreitzberg and Shneiderman
- *KISS - Keep It Simple, Stupid* – Anonymous

Outline

OpenMP Example 1

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

int sum=0; //Global variable touched by all threads

//Function to be called by each thread
void adder() {
#pragma omp critical
    {
        int t = sum; sleep(1); sum = t + 1;
    }
    return;
}

int main() {
    const int numThreads=24;
    int i;
    omp_set_num_threads(numThreads);
#pragma omp parallel for shared(sum)
    for(i = 0; i < numThreads; ++i)
        adder();
    printf("Sum computed: %d\n",sum);
    return 0;
}
```

OpenMP Programming in C/C++

- Based on #pragma compiler directive
- Code added by compiler, NOT preprocessor
- Directive name followed by clauses

```
#pragma omp directive [clause list]
```

```
#pragma omp parallel [clause list]
```

- Serial execution till **parallel** directive is encountered.

OpenMP clauses

- Conditional Parallelization

```
bool doParallel = true;
#pragma omp parallel if(doParallel)
```

- Degree of Concurrency

```
#pragma omp parallel num_threads(8)
```

- Data Handling

```
#pragma omp parallel default(none) private(x) shared(y)
```

```
#pragma omp parallel private(x) lastprivate(y)
```

```
#pragma omp parallel default(shared) firstprivate(x)
```

More about Data Handling

- Most variables are shared by default
  - Functions called from parallel regions are private; Think about thread-safety of functions
  - Automatic variables within statement block are private
- Default attributes

```
#pragma omp parallel default(private|shared|none)
```

- firstprivate : private variables initialized to value at the end of the previous serial region
- lastprivate : Final value of a private variable inside a parallel loop transmitted outside the loop to the serial variable

Data Quiz 1

```
    omp_set_num_threads(8);
    int x = -1;
#pragma omp parallel
    {
        sleep(1);
        //Get thread number
        x = omp_get_thread_num();
    }
    printf("The value of x = %d\n", x);
```

Data Quiz 2

```
    omp_set_num_threads(8);
    int x = -1;
#pragma omp parallel
    {
        sleep(1);
        int x = 6;
    }
    printf("The value of x = %d\n", x);
```

Data Quiz 3

```
    omp_set_num_threads(8);
    int x = -1;
#pragma omp parallel private(x)
    {
        sleep(1);
        x = 6;
    }
    printf("The value of x = %d\n", x);
```

Data Quiz 4

```
    omp_set_num_threads(8);
    int x = 0;
#pragma omp parallel
    {
        sleep(1);
        x = x + 1;
    }
    printf("The value of x = %d\n", x);
```

Data Quiz 5

```
    omp_set_num_threads(8);
    int x = 2;
#pragma omp parallel private(x)
    {
        sleep(1);
        const int threadId = omp_get_thread_num();
        x += threadId;
        printf("Thread %d : x %d\n", threadId, x);
    }
    printf("Final value of x = %d\n", x);
```

Data Quiz 6

```
    omp_set_num_threads(8);
    int x = 2;
#pragma omp parallel firstprivate(x)
    {
        sleep(1);
        const int threadId = omp_get_thread_num();
        x += threadId;
        printf("Thread %d : x %d\n", threadId, x);
    }
    printf("Final value of x = %d\n", x);
```

Data Quiz 7

```
    const int numThreads = 8;
    omp_set_num_threads(numThreads);
    int x = 2;
#pragma omp parallel for firstprivate(x) lastprivate(x)
    for(int i = 0; i < numThreads; ++i)
    {
        sleep(1);
        const int threadId = omp_get_thread_num();
        x += threadId;
        printf("Thread %d : x %d\n", threadId, x);
    }
    printf("Final value of x = %d\n", x);
```

Pi

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

$$\pi \approx \sum_{i=0}^{n} \frac{4}{1+{x_i}^2} \triangle x$$

where, $\triangle x = \frac{1}{n+1}$

and $x_i = (i + \frac{1}{2})\triangle x$

Serial program for π

```c
#include <stdio.h>

int main() {
    const int numPoints = 10000000;
    const double deltaX = 1.0/(double)numPoints;
    double pi = 0.0;
    for(int i = 0; i < numPoints; ++i)
    {
        double xi = (i + 0.5) * deltaX;
        pi += 4.0/(1 + xi * xi);
    }
    pi *= deltaX;
    printf("Value of pi: %.10g\n", pi);
}
```

Parallel program for π : Attempt 1

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

int main() {
    const int numThreads = 24;
    const int numPoints = 10000000;
    const double deltaX = 1.0/(double)numPoints;
    double pi = 0.0;
    omp_set_num_threads(numThreads);
    double components[numThreads];
#pragma omp parallel shared(components)
    {
        const int nt = omp_get_num_threads();
        const int pointsPerThread = numPoints/nt;
        const int threadId = omp_get_thread_num();
        components[threadId] = 0.0;
        double xi = (0.5 + pointsPerThread * threadId) * deltaX;
        for(int i = 0; i < pointsPerThread; ++i)
        {
            components[threadId] += 4.0/(1 + xi * xi);
            xi += deltaX;
        }
    }
    for(int i = 0; i < numThreads; ++i)
        pi += components[i];
    pi *= deltaX;
    printf("Value of pi: %.10g\n", pi);
    return 0;
}
```

Parallel program for π : Attempt 1 ...

```
   double components[numThreads];
#pragma omp parallel shared(components)
   {
       const int nt = omp_get_num_threads();
       const int pointsPerThread = numPoints/nt;
       const int threadId = omp_get_thread_num();
       components[threadId] = 0.0;
       double xi = (0.5 + pointsPerThread * threadId) * deltaX;
       for(int i = 0; i < pointsPerThread; ++i)
       {
           components[threadId] += 4.0/(1 + xi * xi);
           xi += deltaX;
       }
   }
   for(int i = 0; i < numThreads; ++i)
       pi += components[i];
```

Synchronization directives

- critical : Only one thread can enter a *critical* region at any time
- atomic : Provides mutual exclusion in updates to a memory location
  - Applicable to a single variable at a time
  - Limited to binary, increment, decrement operations
- barrier : Each thread waits at a *barrier* until all threads arrive

Parallel program for π : Attempt 2

```
#pragma omp parallel
    {
        const int nt = omp_get_num_threads();
        const int pointsPerThread = numPoints/nt;
        const int threadId = omp_get_thread_num();
        double xi = (0.5 + pointsPerThread * threadId) * deltaX;
        double component = 0.0;
        for(int i = 0; i < pointsPerThread; ++i)
        {
            component += 4.0/(1 + xi * xi);
            xi += deltaX;
        }
#pragma omp critical
        {
            pi += component;
        }
    }
```

Loop worksharing

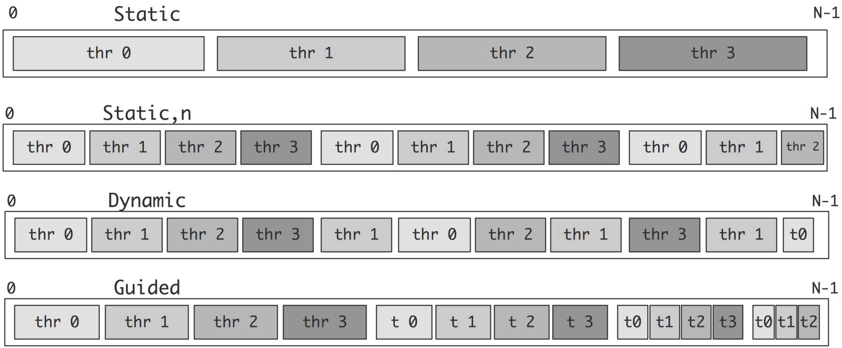


Figure : Courtesy of Victor Eijkhout

Parallel program for π : Attempt 3

```
#pragma omp parallel
    {
        double component = 0.0;
#pragma omp for schedule(static)
        for(int i = 0; i < numPoints; ++i)
        {
            double xi = (0.5 + i) * deltaX;
            component += 4.0/(1 + xi * xi);
        }
#pragma omp critical
        {
            pi += component;
        }
    }
```

Reduction clause

```
#pragma omp parallel for reduction(op:list)
```

- In each thread, local copy of each variable in *list* is made and initialized
- Local copy is updated in each thread
- Operators supported : $+, -, *, min, max$, boolean operators
- All local copies are then reduced to a single value for the operator *op*

Parallel program for π : Attempt 4

```
#pragma omp parallel for schedule(static) reduction(+: pi)
    for(int i = 0; i < numPoints; ++i)
    {
        double xi = (0.5 + i) * deltaX;
        pi += 4.0/(1 + xi * xi);
    }
```

single and master

```
#pragma omp single
{
    //Executed by a single thread
    //Implicit barrier for other threads
}
```

```
#pragma omp master
{
    //Executed by master thread
    //All other threads bypass this section
}
```

single quiz

```
    omp_set_num_threads(8);
    int x = 2;
#pragma omp parallel
    {
#pragma omp single
        {
            sleep(1);
            x = 12;
        }
        const int threadId = omp_get_thread_num();
        printf("Thread %d : x %d\n", threadId, x);
    }
```

master quiz

```
    omp_set_num_threads(8);
    int x = 2;
#pragma omp parallel
    {
#pragma omp master
        {
            sleep(1);
            x = 12;
        }
        const int threadId = omp_get_thread_num();
        printf("Thread %d : x %d\n", threadId, x);
    }
```

Further Reading

- Introduction to Parallel Computing, Second Edition - Grama, Gupta, Karypis, Kumar : Chapter 7
- Introduction to High Performance Computing for Scientists and Engineers - Hager, Wellein : Chapter 6
- `http://openmp.org/mp-documents/OpenMP-4.0-C.pdf`
- `http://openmp.org`