CS528
Pthread and OpenMP

A Sahu

Dept of CSE, IIT Guwahati

Outline

- Threading
- OpenMP
- Cilk
- MPI

Multiprocessor Programming using Threading

Threading Language and Support

- Initially threading used for
  - Multiprocessing on single core : earlier days
  - Feeling/simulation of doing multiple work simultaneously even if on one processor
  - Used TDM, time slicing, Interleaving
- Now a day threading mostly used for
  - To take benefit of multicore
  - Performance and energy efficiency
  - We can use both TDM and SDM

Threading Language and Support

- Pthread: POSIX thread
  - Popular, Initial and Basic one
- Improved Constructs for threading
  - c++ thread : available in c++11, c++14
  - **Java thread : very good memory model**
    - **Atomic function, Mutex**
- Thread Pooling and higher level management
  - OpenMP (loop based)
  - Cilk (dynamic DAG based)

Pthread, C++ Thread, Cilk and OpenMP

```
pthread_t tid1, tid2;
pthread_create(&tid1, NULL, Fun1, NULL);
pthread_create(&tid2, NULL, Fun2, NULL);
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
```

Pthread

```
thread t1(Fun1);
thread t1(Fun2, 0, 1, 2); // 0, 1,2 param to Fun2
t1.join();
t2.join();
```

C++ thread

```
#pragma omp parallel for
for(i=0;i<N;i++)
     A[i]=B[i]*C[i];
//Auto convert serial code to threaded code
// $gcc -fopenmp test.c; export OMP_NUM_THREADS=10; ./a.out
```

```
cilk fib (int n){//Cilk dynamic parallism, DAG recursive code
      if (n<2) return n;
      int x=spawn fib(n-1); //spawn new thread
      tnt y=spawn fib(n-2); //spawn new thread
      sync;
      return x+y;
}
```

Programming with Threads

- Threads
- Shared variables
- The need for synchronization
- Synchronizing with semaphores
- Thread safety and reentrancy
- Races and deadlocks

Traditional View of a Process

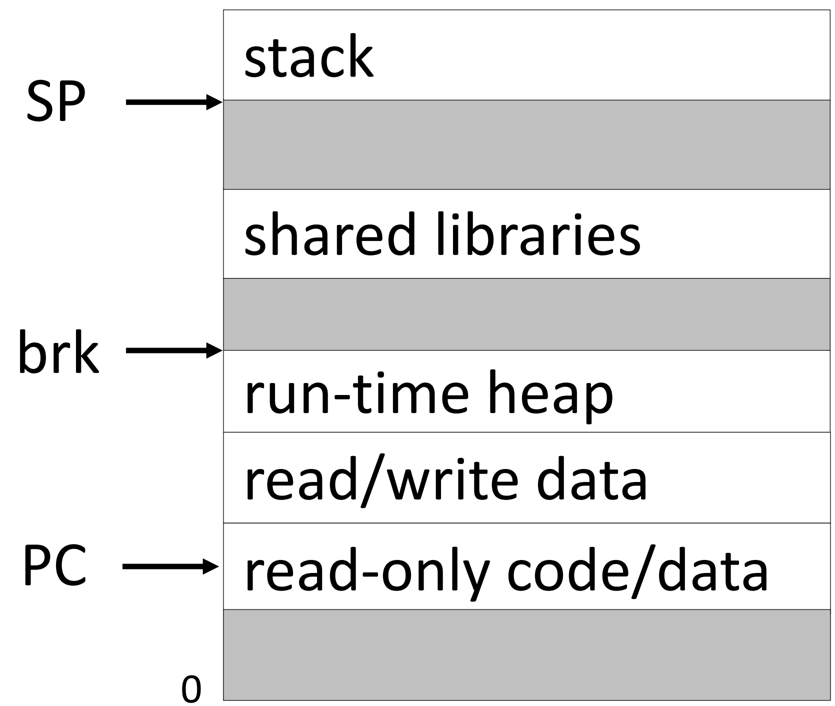
- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

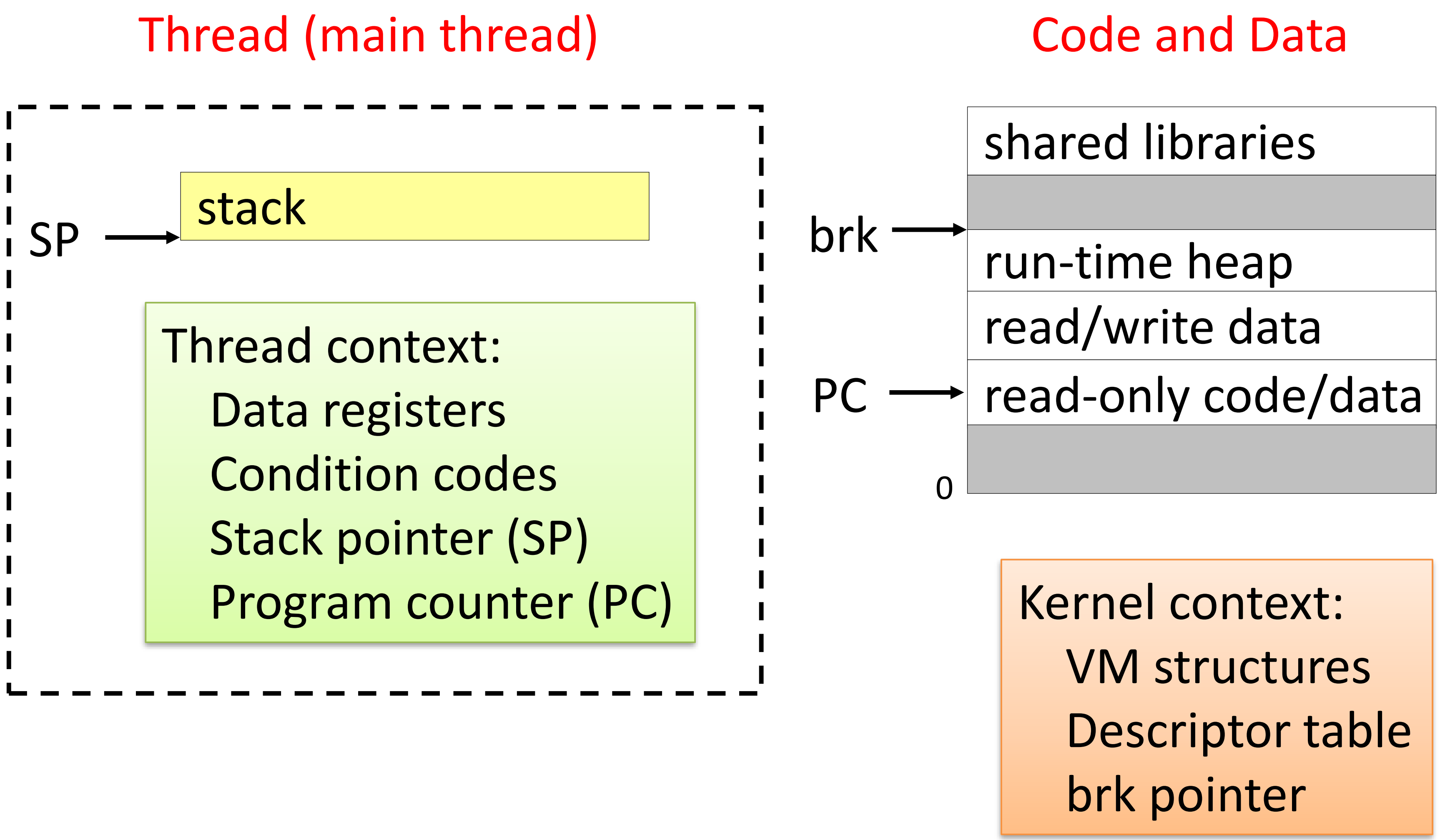
Kernel context:
- VM structures (VMem)
- Descriptor table
- brk pointer

Code, data, and stack



Alternate View of a Process

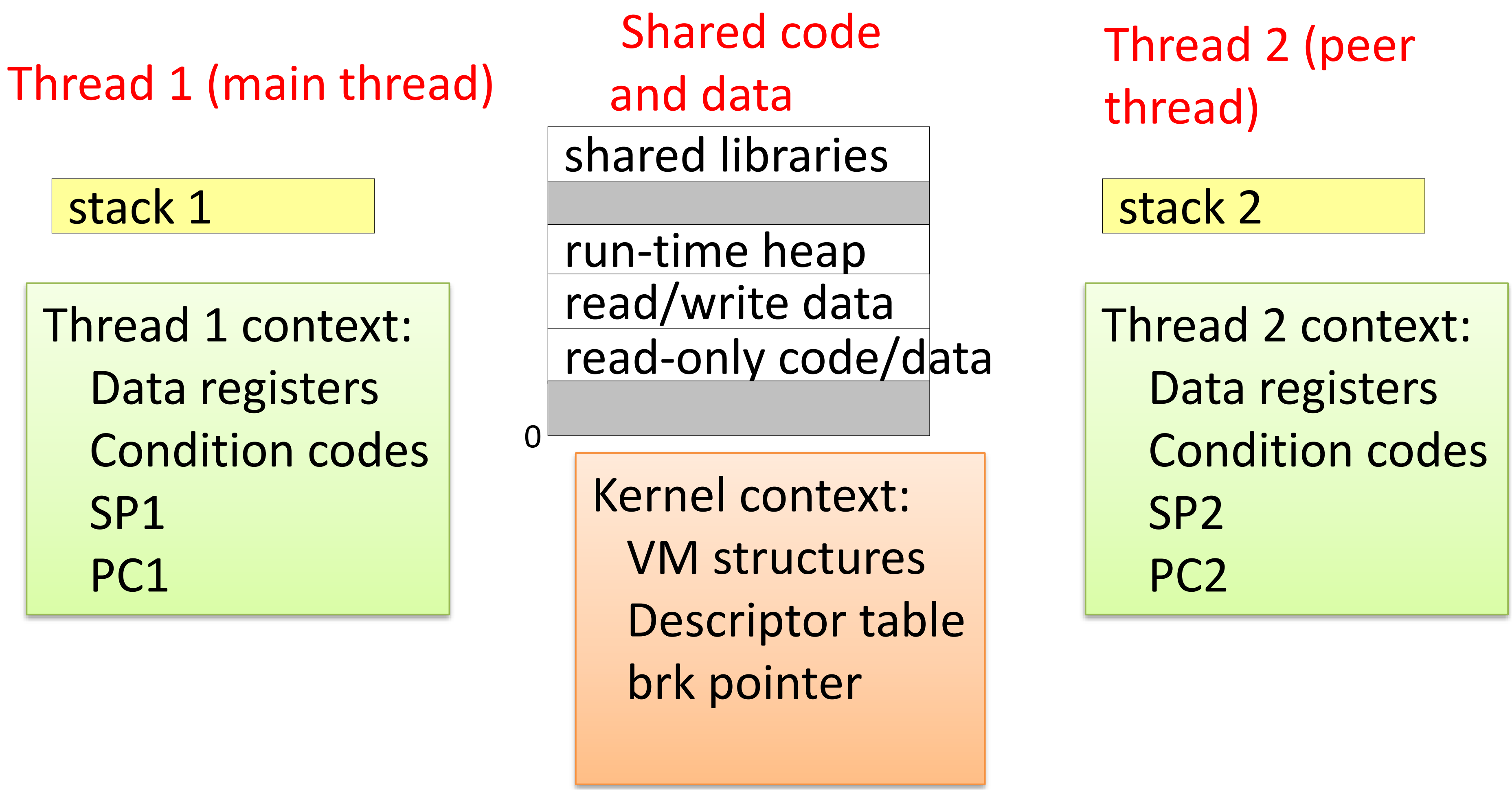
- Process = thread+ code, data & kernel context



A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its *own* logical control flow (sequence of PC values)
  - Each thread *shares* the same code, data, and kernel context
  - Each thread has its own thread id (TID)

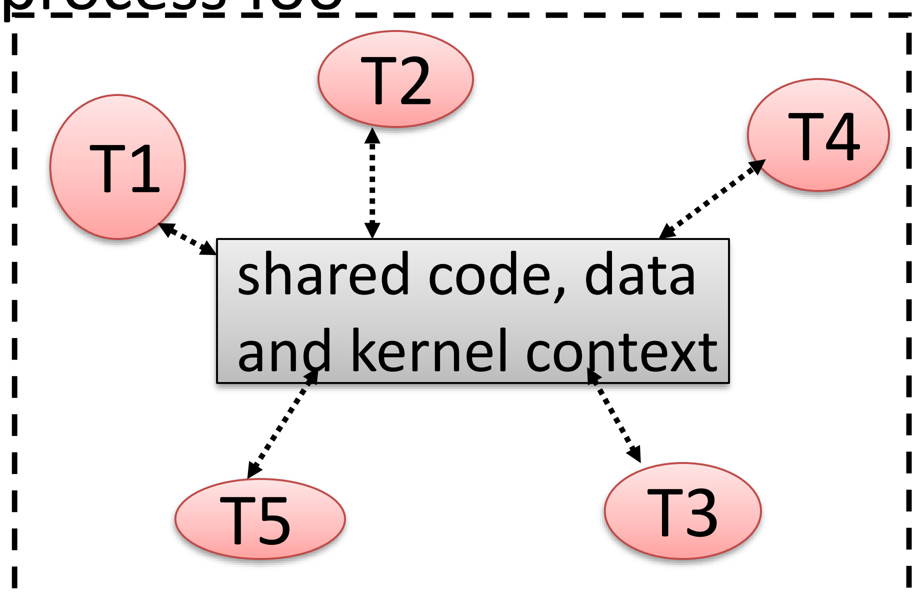
A Process With Multiple Threads



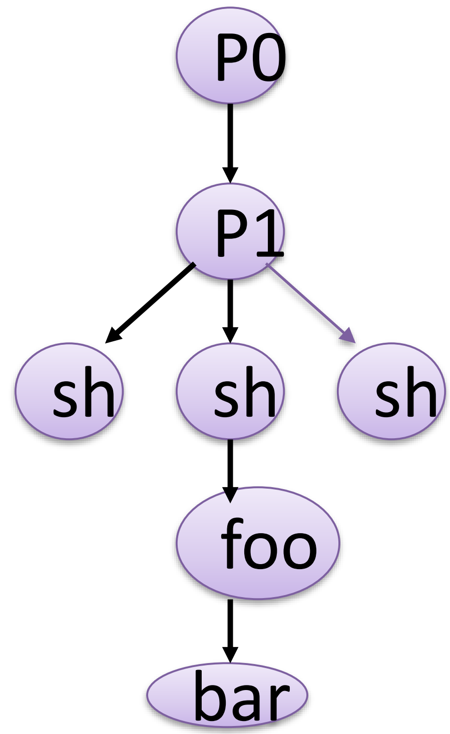
Logical View of Threads

- Threads associated with a process form a pool of peers
  - Unlike processes, which form a tree hierarchy

Threads associated with process foo



Process hierarchy



Posix Threads (Pthreads) Interface

- **Creating and reaping threads**
  - `pthread_create`, `pthread_join`
- **Determining your thread ID :** `pthread_self`
- **Terminating threads**
  - `pthread_cancel`, `pthread_exit`
  - `exit` [terminates all threads] , `return` [terminates current thread]
- **Synchronizing access to shared variables**
  - `pthread_mutex_init`, `pthread_mutex_[un]lock`
  - `pthread_cond_init`, `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

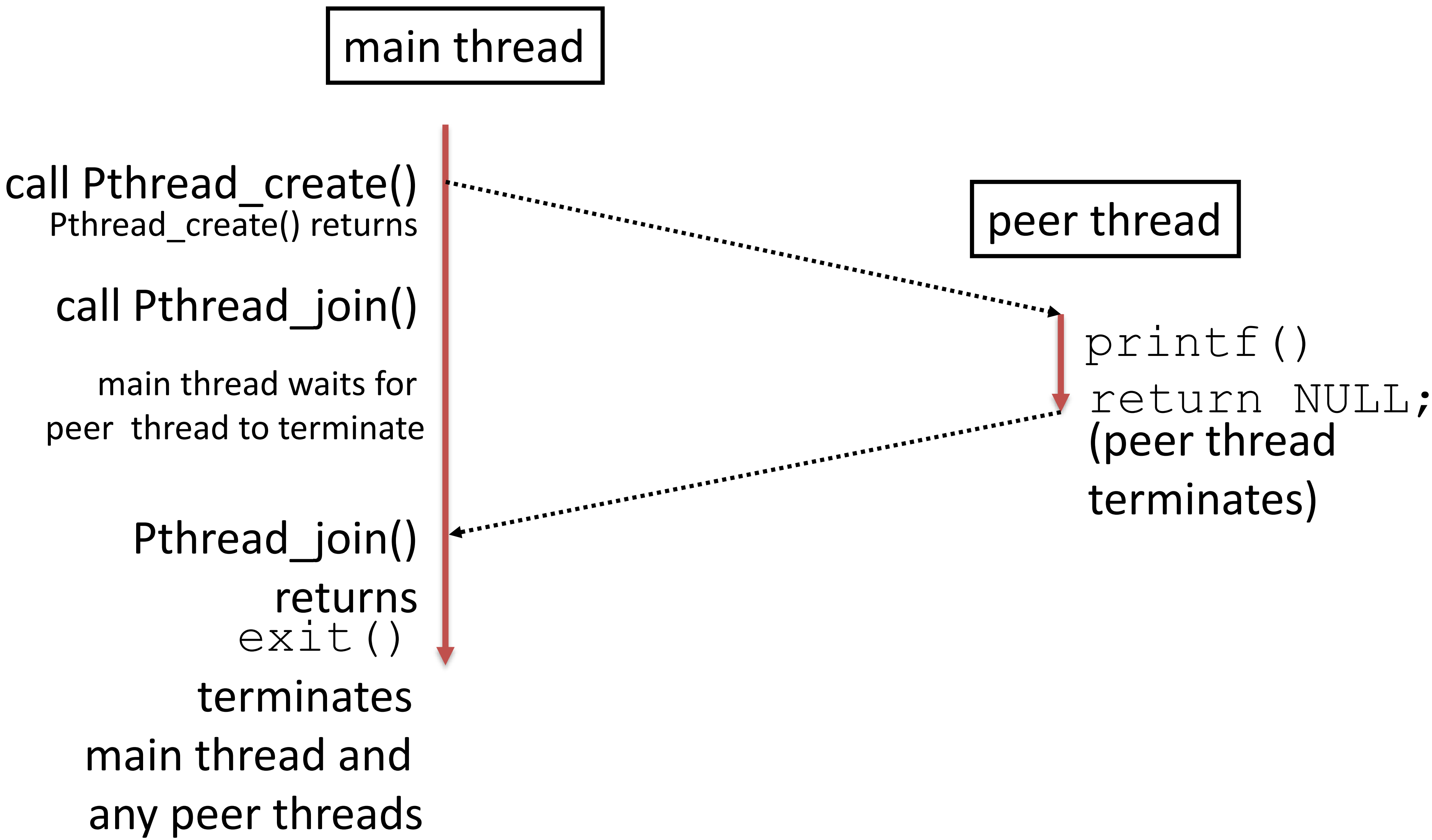
```
/* thread routine */
void *HelloW(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}

int main() {
  pthread_t tid;
  pthread_create(&tid, NULL, Hellow, NULL);
  pthread_join(tid, NULL);
  return 0;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of Threaded “hello, world”



Pros and Cons: Thread-Based Designs

- **+ Easy to share data structures between threads**
  - E.g., logging information, file cache
- **+ Threads are more efficient than processes**
- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - Ease of data sharing is greatest strength of threads
  - Also greatest weakness!

VectorSum Serial

```c
int A[VSize], B[VSize], C[VSize];

void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

Suppose Size=1000

0-249	250-499	500-749	750-999
T1	**T2**	**T3**	**T4**

VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];

void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

- Independent
- Divide work into equal for each thread
- Work per thread: Size/numThread

VectorSum Parallel

```
void *DoVectorSum(void *tid){
    int j, SzPerthrd, LB, UB, TID;
     TID= *((int *)tid);
     SzPerthrd=(VSize/NUM_THREADS);
     LB= SzPerthrd*TID;UB=LB+SzPerthrd;

    for(j=LB;j<UB;j++)
     A[j]=B[j]+C[j];
}
```

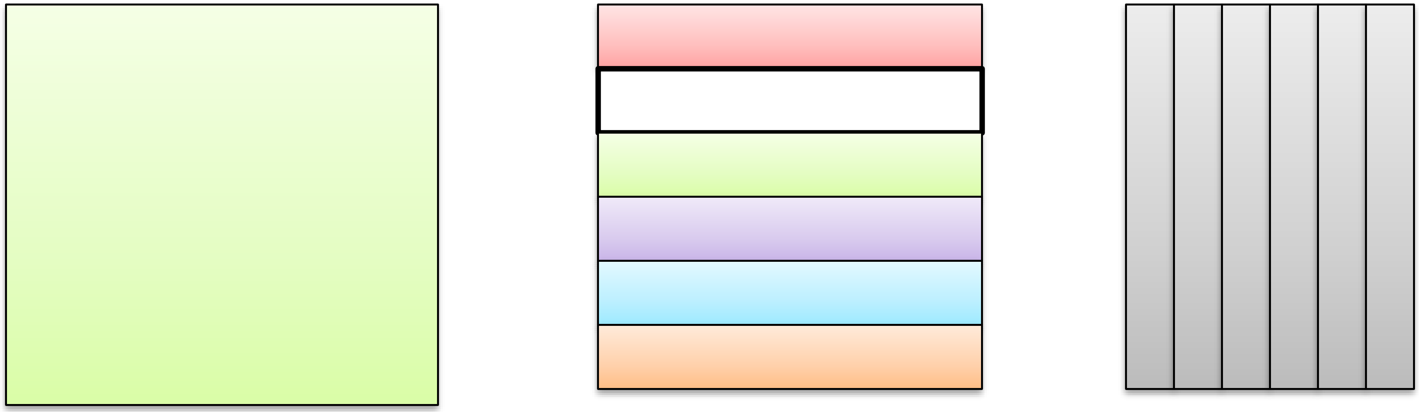
VectorSum Parallel

```
int main(){
    int i;
    pthread_t thread[NUM_THREADS];
     for (i = 0; i < NUM_THREADS; i++)
         pthread_create(&thread[i],
         NULL, DoVectorSum, (void*)&i);
     for (i = 0; i < NUM_THREADS; i++)
        pthread_join(thread[i], NULL);

     return 0;
}
```

Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$$



Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i, j] = \sum_{k=1}^{N} A[i,k] \times B[k, j]$$

- Divide the whole rows to T chunks
  - Each chunk contains : N/T rows, Assume N%T=0

Matrix multiply Serial

```
void MatMul(){
   int i,j,k,S;
   for(i=0;i<Size;i++)
    for(j=0;j<Size;j++){
       S=0;
       for(k=0;k<Size;k++)
          S=S+A[i][k]*B[k][j];
       C[i][j]=S;
    }
 }
```

Matrix Pthreaded: RowWise

```
void * DoMatMulThread(void *arg){
    int i,j,k,S,LB,UB, TID, ThrdSz;
    TID=*((int *)arg);ThrdSz=Size/NumThrd;
    LB=TID*ThrdSz;UB=LB+ThrdSz;
    for(i=LB;i<UB;i++)
        for(j=0;j<Size;j++){
        S=0;
        for(k=0;k<Size;k++)
          S=S+A[i][k]*B[k][j];
        C[i][j]=S;
        }
 }
```

Matrix Pthreaded: RowWise

```
int main(){
    pthread_t thread[NumThread];
    int t;
    Initialize();
     for(t=0; t<NumThread; t++)
          pthread_create(&thread[t], NULL,
          DoMatMulThread, &t);
     for(t=0; t<NumThread; t++)
          pthread_join(thread[t], NULL);
     TestResult();
     return 0;
}
```

Estimating π using Monte Carlo

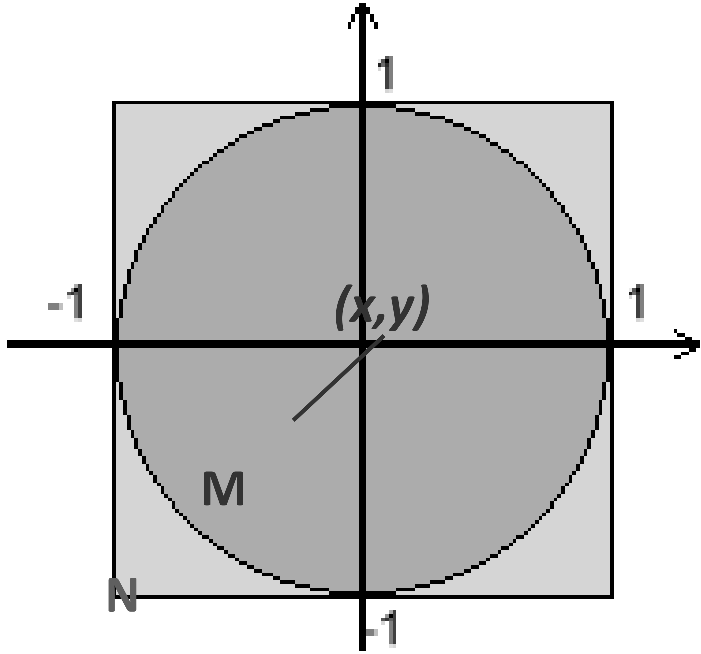
- **The probability of a random point lying inside the unit circle:**

$$\mathbf{P}\left(x^2+y^2<1\right)=\frac{A_{circle}}{A_{square}}=\frac{\pi}{4}$$

- **If pick a random point *N* times and *M* of those times the point lies inside the unit circle:**

$$\mathbf{P}^{\circ}\left(x^2+y^2<1\right)=\frac{M}{N}$$

- **If *N* becomes very large,** $P=P^0$



$$\pi=\frac{4\cdot M}{N}$$

Value of PI: Monte-Carlo Method

```
void MontePI(){
    int count=0,i;
    double x,y,z;
    for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        z = x*x+y*y;
        if (z<=1) count++;
        }
    pi=(double)count/niter*4;
}
```

PI- Multi-threaded

- 1 thread you are able to generate N points
  - Suppose M points fall under unit circle
  - PI=4M/N
- With 10 thread generate 10XN points and calculate more accurately
  - Each thread calculate own value of PI (or M)
  - Average later on (or recalculate PI from collective M)

Value of PI: Pthreaded

```
int main(){
   pthread_t thread[NumThread]; double pi;
   int t, at[NumThread], count, TotalIter;
    for(t=0; t<NumThread; t++)
      pthread_create(&thread[t], NULL,
          DoLocalMC_PI, &t);
    for(t=0; t<NumThread; t++)
      pthread_join(thread[t], NULL);
    for(t=0;t<NumThread;t++) count+=LCount[t];
    TotalIter=niter*NumThread;
    pi=((double)count/TotalIter)*4;
    return 0;
}
```

Value of PI: Pthreaded

```
int LCount[NumThread];
void *DoLocalMC_PI(void *aTid){
   int tid, count, i; double x,y,z;
    tid= *((int *)aTid);
    count=0;LCount[tid]=0;
    for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        z = x*x+y*y; if (z<=1) count++;
        }
    LCount[tid]=count;
}
```

Locking Shared Variable

Parallel Counter: without Lock

```
#define NITERS 100
int cnt = 0; /* shared */
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, count, NULL);
  pthread_create(&tid2, NULL, count, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  if(cnt!=NITERS*2) printf("BOOM! cnt=%d",cnt);
    else printf("OK cnt=%d\n", cnt);
}
```

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++) cnt++;
}
```

```
$./badcnt
BOOM! cnt=196
$./badcnt
BOOM! cnt=184
```

cnt should be 200

What went wrong?!

Thread Safety

- Functions called from a thread must be *thread-safe*
- There are four (non-disjoint) classes of thread-unsafe functions:
  - **Class 1: Failing to protect shared variables : L/UL**
  - Class 2: Relying on persistent state across invocations
  - Class 3: Returning pointer to static variable
  - Class 4: Calling thread-unsafe functions

Class 1: Failing to protect shared variables

- Fix: Use Lock and unlock semaphore operations
- Issue: Synchronization operations will slow down code
- Example: `goodcnt.c`

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++)
      pthread_mutex_lock(&LV);
      cnt++;
      pthread_mutex_unlock(&LV);
} // LV is lock variable
```

Class 2: Relying on persistent state across multiple function invocations

- Random number generator relies on static state
- Fix: Rewrite function so that caller passes in all necessary state, ➔ Maintain Thread Specific State

```
int rand() {
    static uint next = 1;
    next = next*1103515245 + 12345;
    return (uint)(next/65536)% 32768;
}
void srand(uint seed) {
    next = seed;
}
```

Class 3: Returning pointer to `static` variable

- Fixes: 1. Rewrite code so caller passes pointer to `struct`, Issue: Requires changes in caller and callee
- *Lock-and-copy :* Issue: Requires only simple changes in caller (and none in callee), However, caller must free memory

```
struct hostent  *gethostbyname(
                              char* name){
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

Class 3: Returning pointer to `static` variable

```
struct hostent *gethostbyname_ts(char *p) {
  struct hostent *q = Malloc(...);
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;    /* copy */
  V(&mutex);
  return q;
}
```

```
hostp = malloc(...));
gethostbyname_r(name, hostp);
```

Class 4: Calling thread-unsafe functions

- Calling one thread-unsafe function makes an entire function thread-unsafe
- Fix: Modify the function so it calls only thread-safe functions

C++ Thread:atomic

```
atomic_uint AtomicCount;
void DoCount(){
   int j, timesperthrd;
   timesperthrd=(TIMES/NUM_THREADS);
   for(j=0;j<timesperthrd;j++) AtomicCount++;
}

main(){
 thread T[N_THRDS]; int i;
 for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
 for(i=0;i<N_THRDS;i++) T[i].join();
}
```

Improved

```
atomic_uint AtomicCount;
void DoCount(){
  int j, timesperthrd,localcount=0;
  timesperthrd=(TIMES/NUM_THREADS);
  for(j=0;j<timesperthrd;j++)localcount++;
            AtomicCount+=localcount;
}
main(){
     thread T[N_THRDS]; int i;
  for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
  for(i=0;i<N_THRDS;i++) T[i].join();
}
```

Java Thread:synchronized

```
void run()
        synchronized(sender)  {
      // synchronizing the sender object //only one thread at time
          sender.send(msg);
       }
}


synchronized void printTable(int n){//synchronized method
                  …….
  }
```

OpenMP

OpenMP

- Compiler directive: Automatic parallelization
- Auto generate thread and get synchronized

```
#include <openmp.h>
main(){
#pragma omp parallel
#pragma omp for schedule(static)
{
   for (int i=0; i<N; i++) {
        a[i]=b[i]+c[i];
   }
}
}
```

```
$ gcc –fopenmp  test.c
$ export OMP_NUM_THREADS=4
$./a.out
```

OpenMP: Parallelism

Sequential code

```
for (int i=0; i<N; i++)
    a[i]=b[i]+c[i];
```

OpenMP: Parallelism

(Semi) manual parallel

```
#pragma omp parallel
{
 int id =omp_get_thread_num();
 int Nthr=omp_get_num_threads();
 int istart = id*N/Nthr
 int iend= (id+1)*N/Nthr;
 for (int i=istart;i<iend;i++){
        a[i]=b[i]+c[i];
   }
}
```

OpenMP: Parallelism

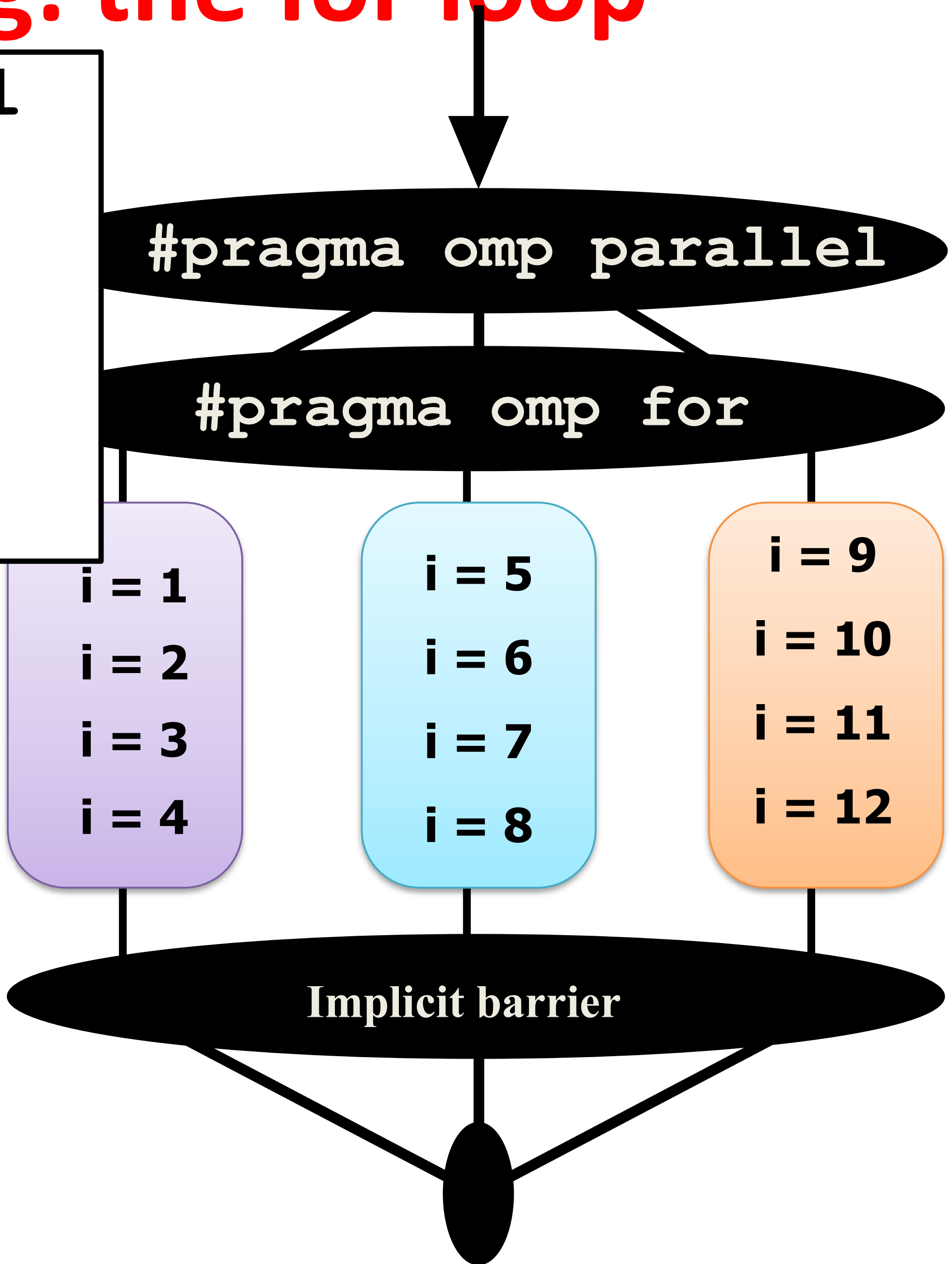
Auto parallel for loop

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++){
    a[i]=b[i]+c[i];
  }
}
```

Work-sharing: the for loop

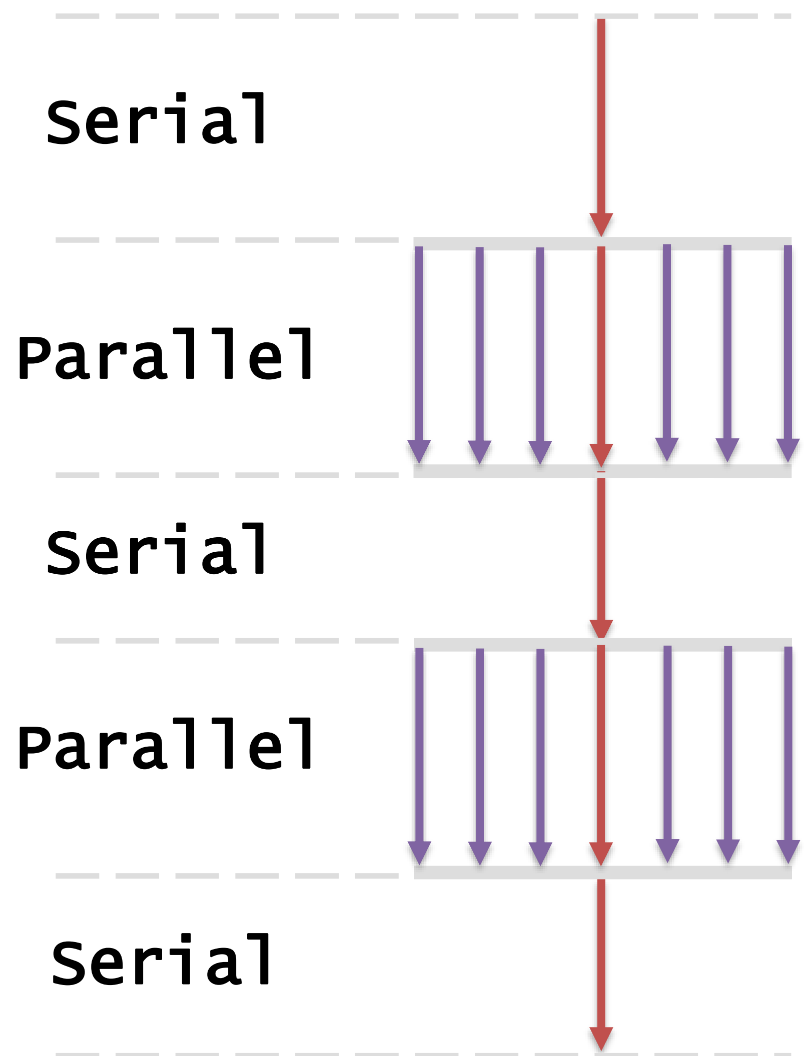
```
#pragma omp parallel
#pragma omp for
{
   for(i=1;i<13;i++)
      c[i]=a[i]+b[i];
}
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



OpenMP Fork-and-Join model

```
printf("begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("done\n");
```



AutoMutex: Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++) {
        lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```

Threads wait their turn; only one thread at a time executes the critical section

Reduction Clause

Shared variable

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
{
  for (i=0; i<N; i++) {
    sum = sum + A[i];
  }
}
```

OpenMP Schedule

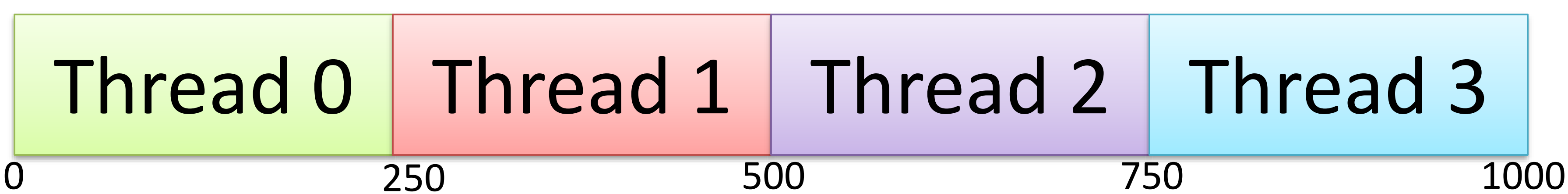
- Can help OpenMP decide how to handle parallelism

  schedule(type [,chunk])

- **Schedule Types**
  - **Static** – Iterations divided into size chunk, if specified, and statically assigned to threads
  - **Dynamic** – Iterations divided into size chunk, if specified, and dynamically scheduled among threads

Static Schedule

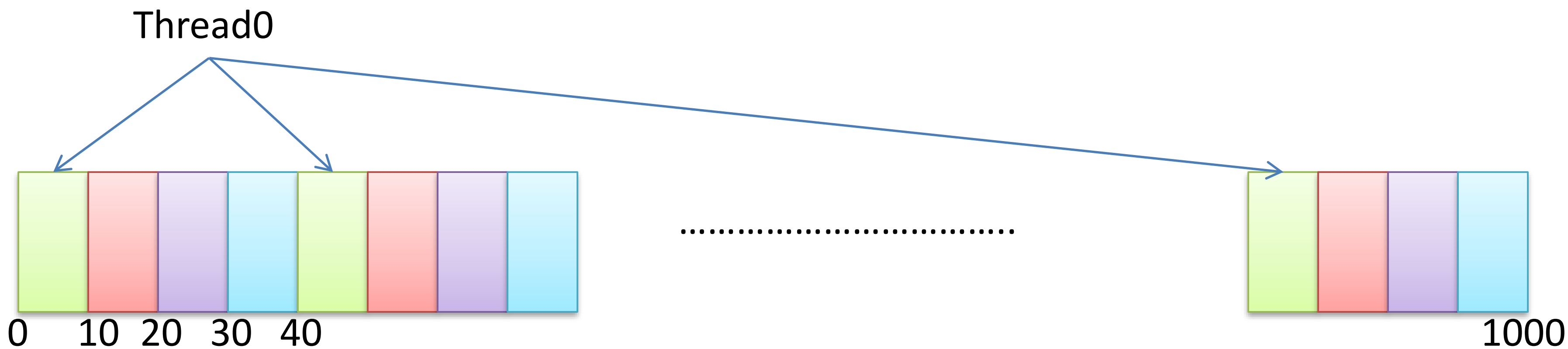
- Although the OpenMP standard does not specify how a loop should be partitioned
- Most compilers split the loop in N/p (N #iterations, p #threads) chunks by default.
- This is called a static schedule (with chunk size N/p)
  - *For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:*



Static Schedule with chunk

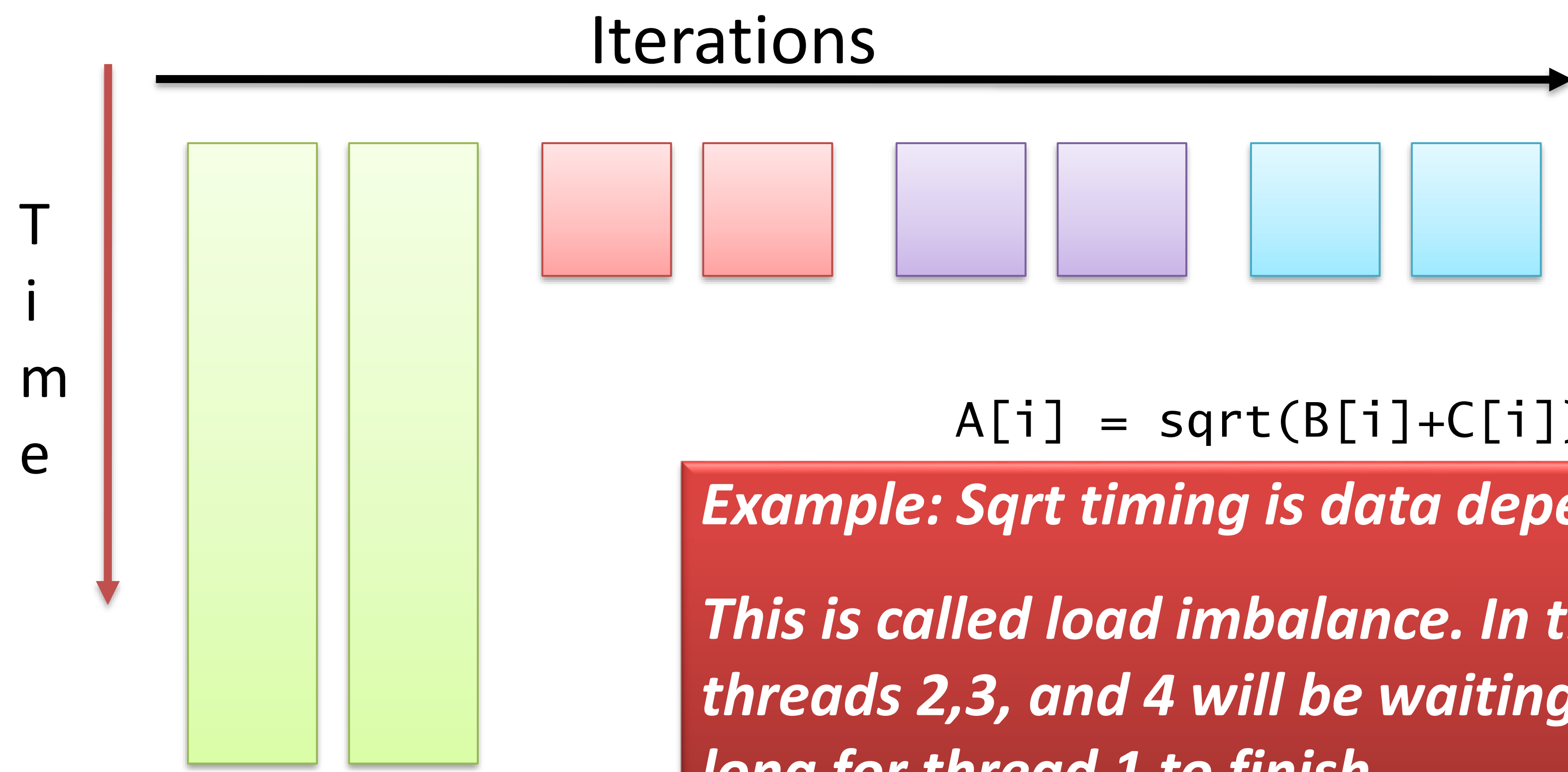
- A loop with 1000 iterations and 4 omp threads. Static Schedule with Chunk 10

```
#pragma omp parallel for schedule (static, 10)
{
for (i=0; i<1000; i++)
    A[i] = B[i] + C[i];
}
```



Issues with Static schedule

- With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations).
- This is not always the best way to partition. Why is This?




```
A[i] = sqrt(B[i]+C[i]);
```

Example: Sqrt timing is data dependent…

This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish

Dynamic Schedule

- With a dynamic schedule new chunks are assigned to threads when they come available.
- SCHEDULE(DYNAMIC,n)
  - Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.

Dynamic Schedule

- SCHEDULE(GUIDED,n)
  - Similar to DYNAMIC but chunk size is relative to number of iterations left.
- Although Dynamic scheduling might be the prefered choice to prevent load inbalance
  - In some situations, there is a significant overhead involved compared to static scheduling.

More Examples on OpenMP

- http://users.abo.fi/mats/PP2012/examples/OpenMP/