CS528

Algo/App Classification : DAO
&
Multi-threading and OpenMP

A Sahu

Dept of CSE, IIT Guwahati

Outline

- APP classification based on Data access
- Threading
  - Pthread, Locking
- OpenMP
  - AutoParallelization
- MPI

Algorithm Classification and Access Optimization

- O(N)/O(N) : If the # of arithmetic Ops and data transfer (LD/ST) are proportional to Loop Length N
  - Optimization potential is limited
  - Example Scalar Product, vector add, sparse MVM
- Memory bound for large N
- Compiler generated code achieve good perf.
  - Using software pipelining and loop nests

Loop fusion for O(N)/O(N)

```
for(i=0;i<N;i++)
    A[i]=B[i]+C[i];   //Bc=3W/1F
for(i=0;i<N;i++)
    Z[i]=B[i]+E[i];   //Bc=3W/1F
```

```
for(i=0;i<N;i++){
    A[i]=B[i]+C[i];
    Z[i]=B[i]+E[i];
}
```

B_C=5W/2F
No need get to
B[i]again

$O(N^2)/O(N^2)$: OPS/DataTransfer

- Typical two loop nests with loop strip count N
  - $O(N^2)$ operation for $O(N^2)$ loads and stores
- Example: dense MVM, Mat add, MatTrans
- MVM : -> Covert both access to row access

```
for(i=0;i<N;i++) {
      tmp=C[i]
      for(j=0;j<N;j++)   tmp=A[i][j]*B[j]
      C[i]=tmp
}
```

  - Row I of A and vector B
  - Original Bc=2W/2F but → 2W*m/2F
  - m is miss rate of cache for Row access

$O(N^3)/O(N^2)$: OPS/DataTransfer

- Typical three loop nests
  - $O(N^3)$ operation for $O(N^2)$ loads and stores
- Example: dense Matrix Mulltiplication
- Implementation of cache Bound
  - Already studied : loop interchange
  - **Blocking : Strassen multiplication, will be discussed later**

Multiprocessor Programming using Theading

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

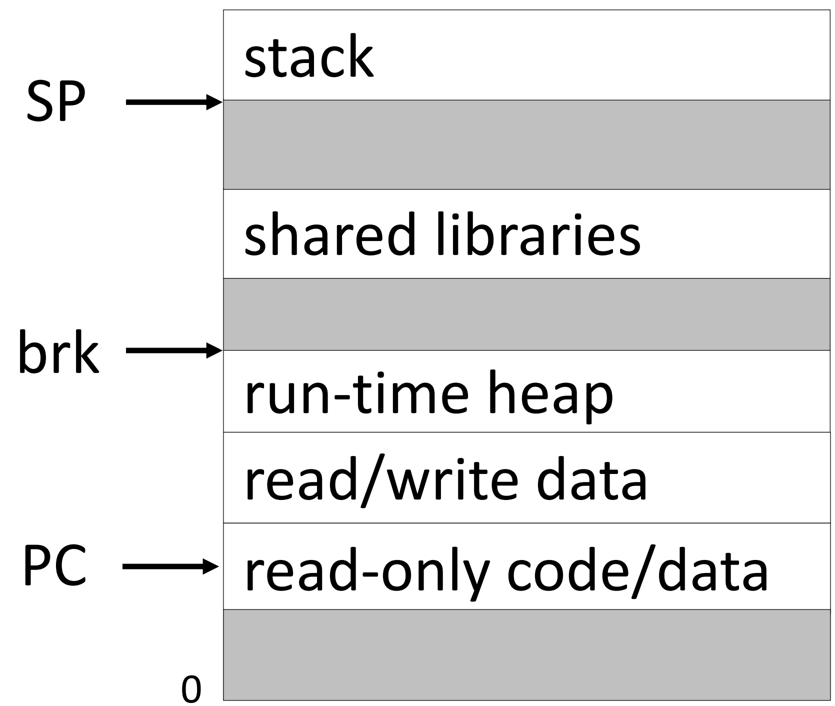
- Process = process context + code, data, and stack

Process context

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

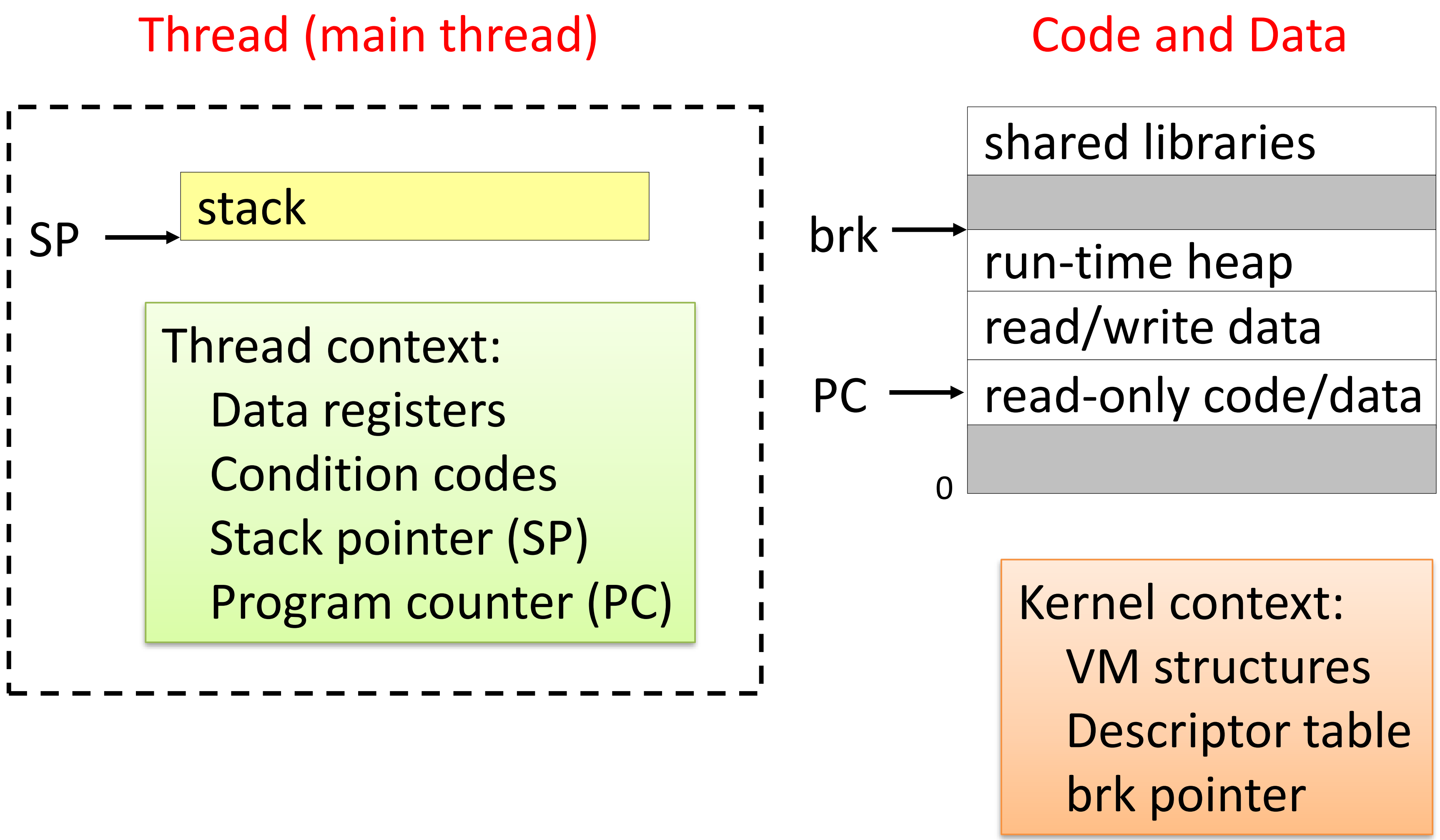
Kernel context:
- VM structures (VMem)
- Descriptor table
- brk pointer

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

Thread 1 (main thread)

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Shared code and data

shared libraries
run-time heap
read/write data
read-only code/data

0

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Thread 2 (peer thread)

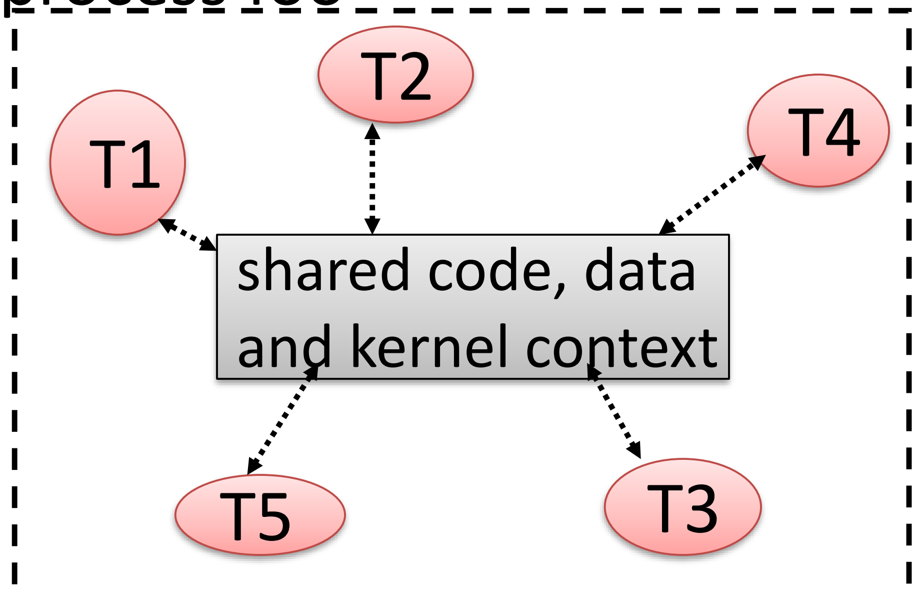
stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

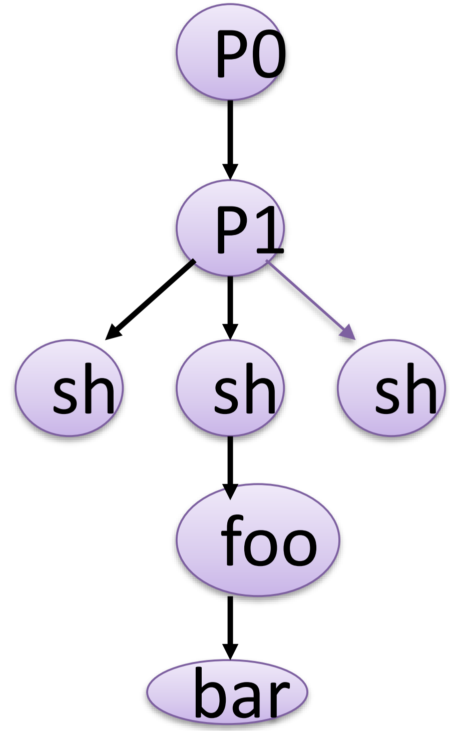
Logical View of Threads

- 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

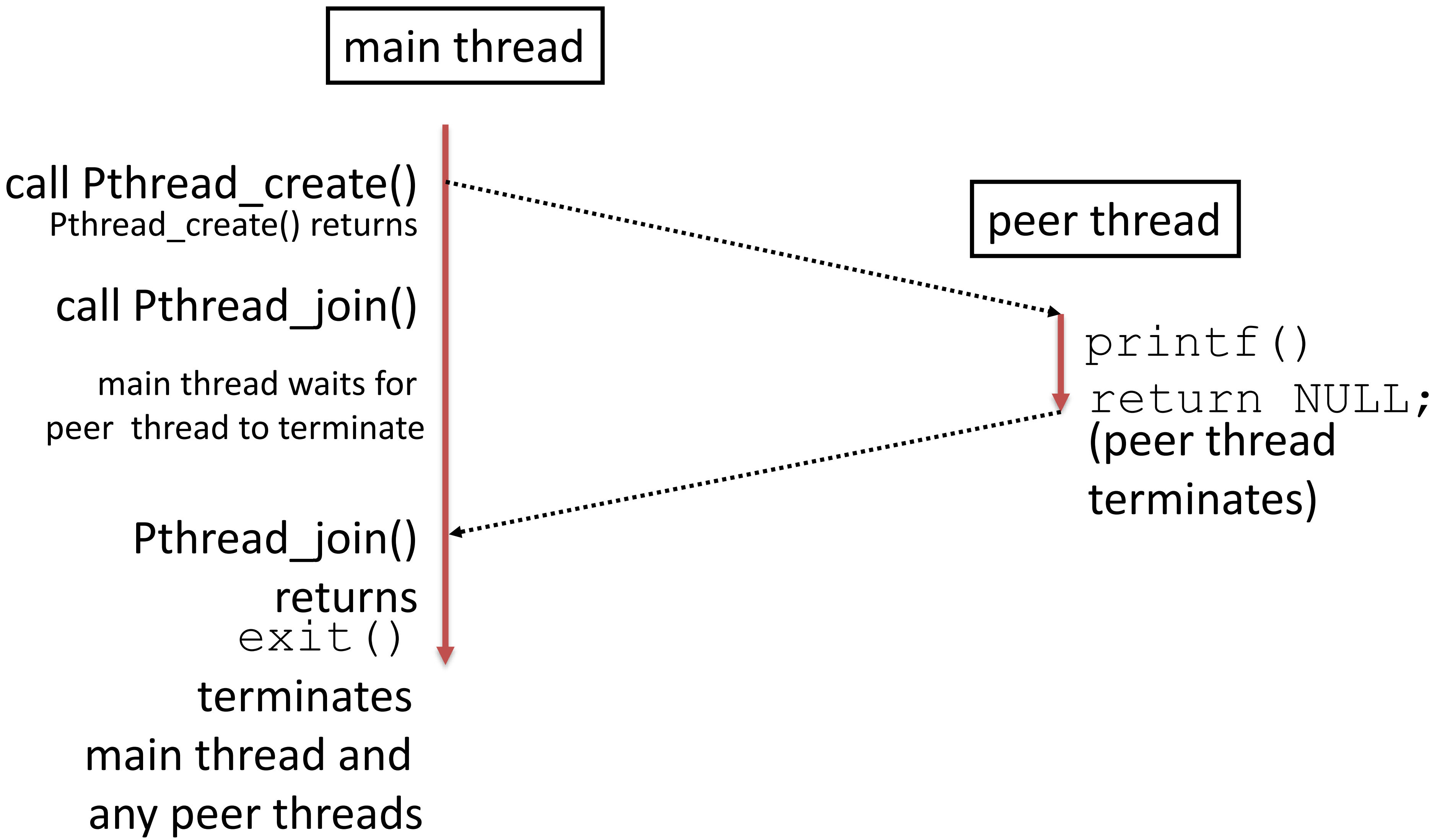
```
/* thread routine */
void *HelloW(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}

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

```
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];
}
```

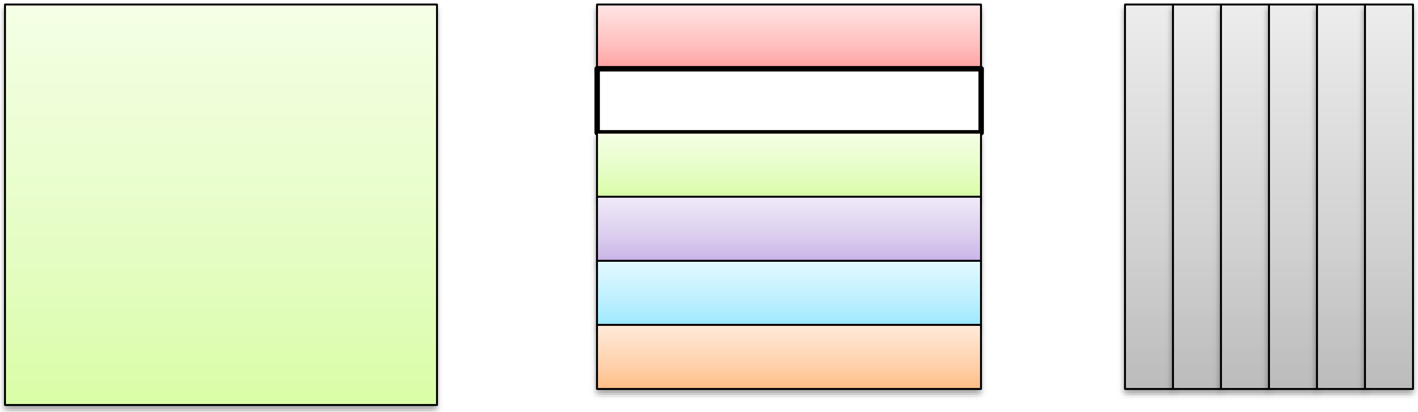
VectorSum Parallel

```
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

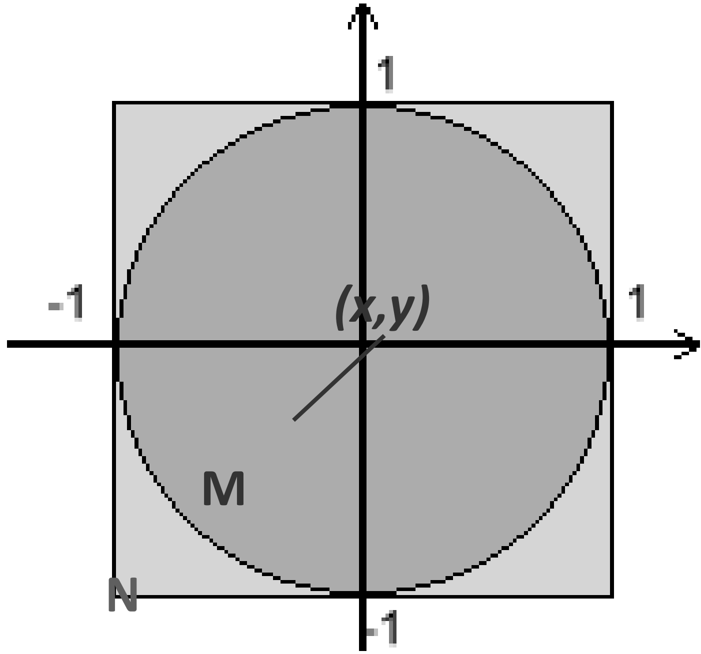
- The probability of a random point lying inside the unit circle:

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

- 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;
}
```