Cluster Computing

Main Approaches to Parallelizing an application with threads

- Static orchestration
- Dynamic orchestration
- Fork-Join
- Specialized Workers
- Higher level tools

Cluster Computing

Static Orchestration

- Static Orchestration is the most common approach to parallel orchestration
- Divide the data into equal sized blocks and issue one block per processor
- Pitfalls
  - A static division is not always possible
  - Evenly sized blocks of data don’t necessarily mean evenly distributed workload

Cluster Computing

Dynamic Orchestration

- Used if static orchestration is not possible due to uneven work on different data-blocks
- Used if we have heterogeneous computing resources
- Used if we need to share the CPUs

Cluster Computing

Fork-Join

- Also known as loosely synchronous parallelism
- Usually used in combination with static orchestration
- Usage:
  - Fork *n* workers before a loop and let each worker perform part of the loop
  - All workers must terminate before the instruction after the loop can be executed

Specialized Worker

- Used if the data is not easily divided but the instructions can
- Much harder way to parallelize
- Usually limits the degree of parallelism to very few threads

Cluster Computing

Challenges in programming with threads

- Getting the right data
- Avoiding race-conditions
- Synchronizing execution
- Allowing threads to run Asynchronously
- Reusing threads
- Considering your memory layout

Cluster Computing

Volatile

- Things are not always as they may appear

```
i++; =    read a,i
          inc a
          write i,a
```

or `inc a` ?

- Solution: declare `i` as volatile

Cluster Computing

Mutual Exclusion

- We must protect critical regions

A: i++;
B: i++;

```
read a,i      read a,i      read a,i
inc a         read a,i      inc a
write i,a     inc a         read a,i
read a,i      inc a         write i,a
inc a         write i,a     inc a
write i,a     write i,a     write i,a
```

Cluster Computing

Mutual Exclusion

- Critical Region in Java

```
class counter {
  int cnt;
  public void counter(){cnt=0;}
  synchronized public int inc(){return ++cnt;}
  synchronized public int reset(){return cnt=0;}
}
```

Cluster Computing

Mutual Exclusion

- Critical Region in PThreads

```
pthread_mutex_lock(&mutex);
i++;
pthread_mutex_unlock(&mutex);
```

Cluster Computing

Barriers

- When multiple threads iterates over the same data block we often need to agree when an iteration begins (or ends)
- Barriers comes in three shapes
  - Spinning
  - Blocking
  - Mixed

Cluster Computing

Spinning Barrier

```
class counter {
  int cnt;
  public void counter(){cnt=0;}
  synchronized public int inc(){return ++cnt;}
  synchronized public int reset(){return cnt=0;}
}

class Barrier {
 counter c;
 int members;
 volatile boolean spin;
 public Barrier(int num){members=num; spin=true; c=new counter();}
 public void meet(){
    boolean state=spin;
    if(c.inc()==members){
      c.reset();
      spin=!spin;
    }
    while(spin==state);
 }
}
```

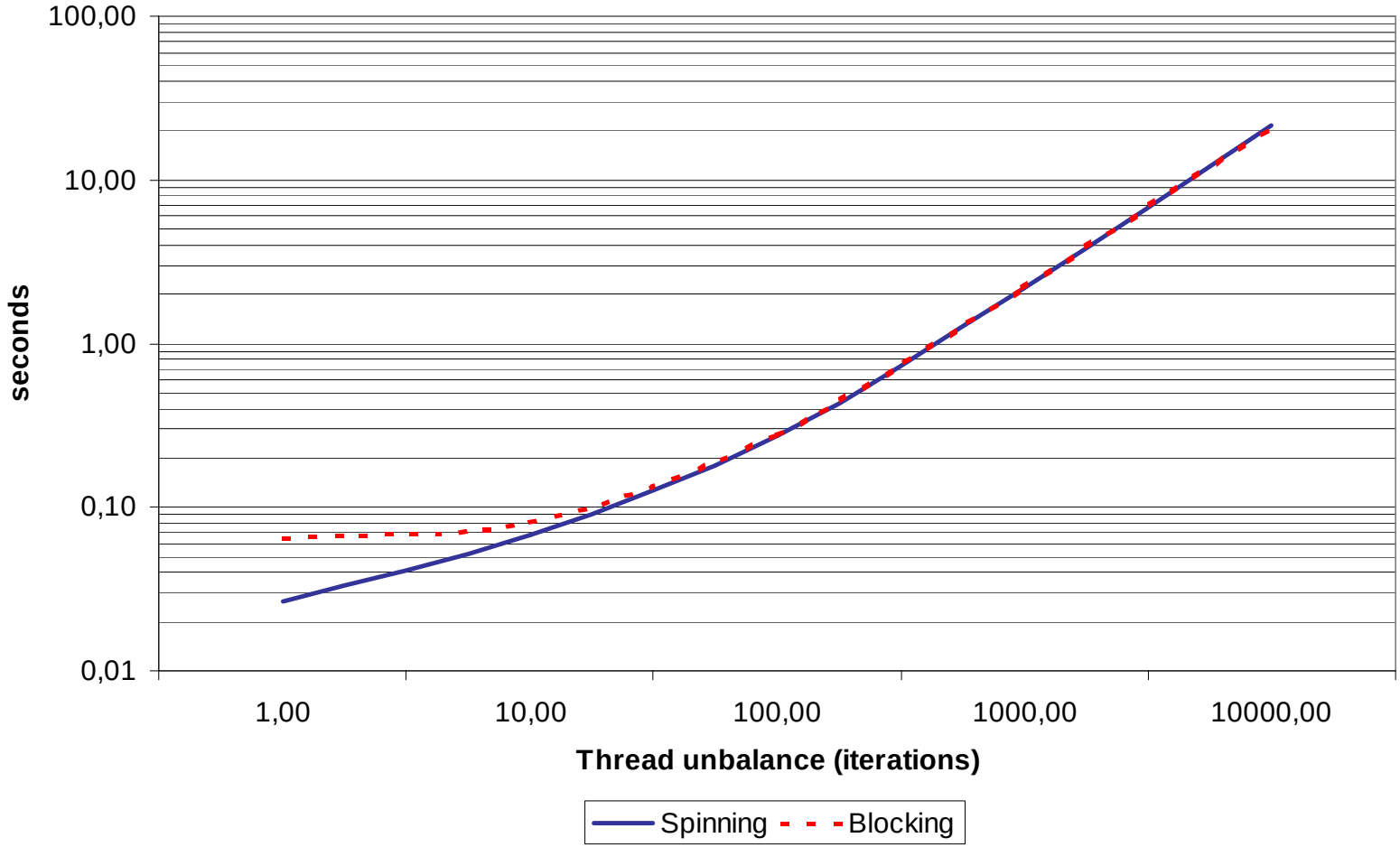
Cluster Computing

Blocking Barrier

```
class Barrier {
  int cnt, members;
  public Barrier(int members){this.members=members;}
  public synchronized void meet() {
    if(++cnt==members){
      cnt=0;
      notifyAll();
    }
    else try {wait();} catch(Exception e){};
  }
}
```

Cluster Computing

Spin or block?



Cluster Computing

Bounded Buffers

- We often wish to be able to execute the threads with more loose synchrony
- In these cases we often have a producer-consumer setup
- Rather than having a handover rendezvous a bounded-buffer is preferable

Cluster Computing

Thread Pools

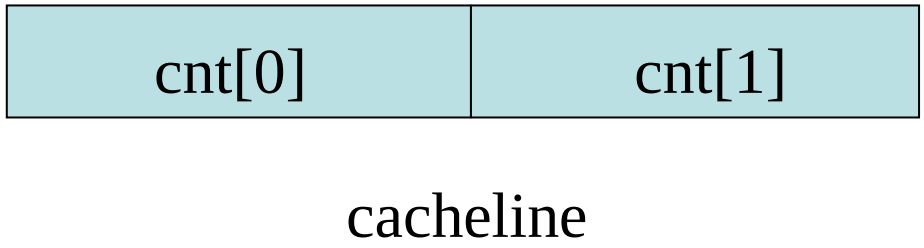
- Since threads are expensive to create and since some applications create and destroy threads frequently it is a real alternative to have a pool of threads
  - On create one first checks if a thread already exists in the pool if so this is chosen otherwise a new is created
  - On destroy the thread is put into the pool

Cluster Computing

False Sharing

- Unfortunate memory layout may cause performance problems due to ’false sharing’

```
Int cnt[2];
A: cnt[0]++;
B: cnt[1]++;
```



Cluster Computing

False Sharing: A test!

```
volatile int count[2];

#ifdef FALSE
worker(void * arg)
{
  int i,index=(int)arg;
  for(i=0;i<100000000; i++)
    count[index]++;
}
#else
worker(void * arg)
{
  int i,index=(int)arg;
  int temp=0;

  for(i=0;i<100000000; i++)
    temp++;

    count[index]+=temp;
}
#endif
```

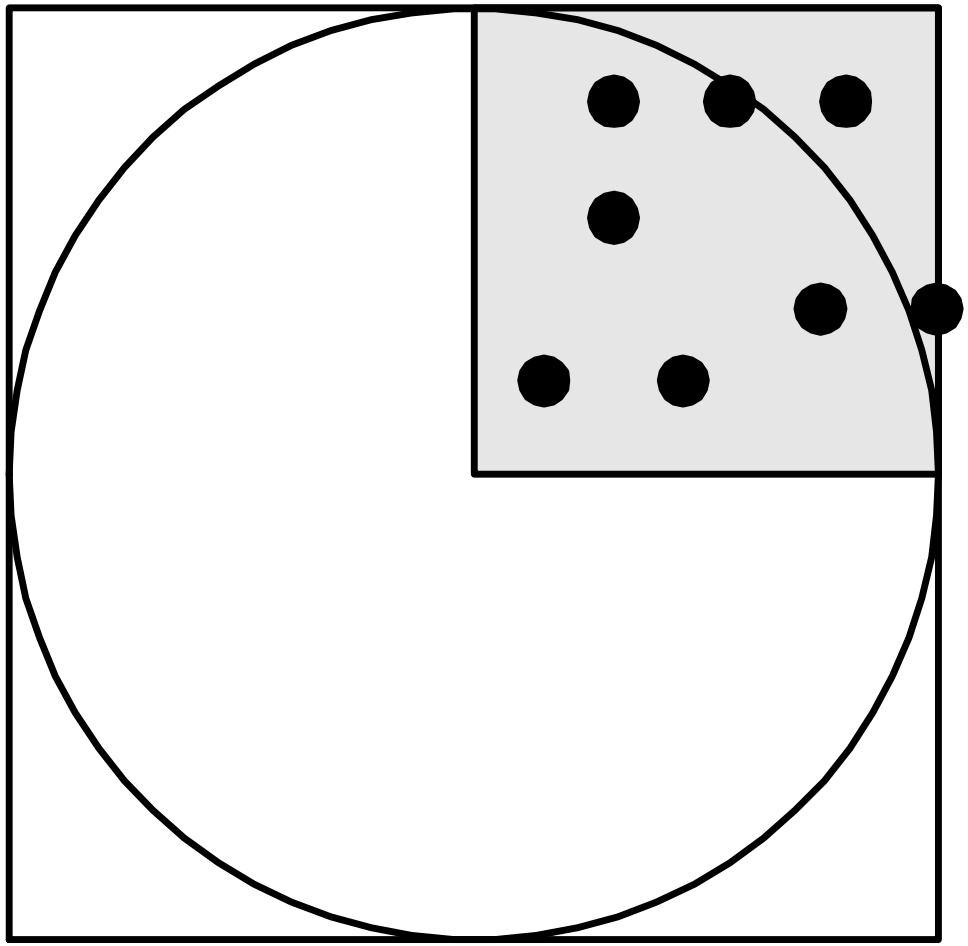
```
main()
{
  pthread_t t;

  pthread_create(&t,NULL,worker,NULL);
  worker((void *)1);

  printf("%d %d\n",count[0],count[1]);

}
```

False Sharing
13.190u 0.020s 0:06.79 194.5% 0+0k 0+0io 245pf+0w

No False Sharing
2.690u 0.000s 0:01.36 197.7% 0+0k 0+0io 245pf+0w

Cluster Computing

Monte Carlo PI



Cluster Computing

Monte Carlo Pi - sequential

```
import java.util.Random;
public class mcpi_sequential {
  public static void main(String argv[]){
    final int LIM=10000000; //We throw 10E7 darts
    int i,count=0;
    double x,y;
    Random r=new Random();

    for(i=1;i<=LIM;i++) {
      x=r.nextDouble();
      y=r.nextDouble();
      if((x*x+y*y)<=1.0)count++;
    }
    System.out.println("PI is " + 4.0*count/LIM);
  }
}
```

```
time java PI
PI is 3.141338
0.100u 0.010s 1:51.26 0.0%      0+0k 0+0io 4739pf+0w
```

Cluster Computing

Monte Carlo PI – static scheduled

```
public class PI {
  public static void main(String argv[]){
    Worker w1,w2;
    data cnt;
    int LIM=10000000;
    cnt=new data();
    w1=new Worker(0,LIM/2,cnt); w2=new Worker(1,LIM/2,cnt);
    w1.start(); w2.start();
    try { w1.join();} catch (InterruptedException e) {};
    try { w2.join();} catch (InterruptedException e) {};
    System.out.println("PI is " + 4.0*cnt.count/LIM);
  }
```

Cluster Computing

Monte Carlo Pi – static (i)

```
class data {
  public int count;
  data(){ count=0; }
}

class Worker extends Thread {
private int id,LIM;  data cnt;
  public Worker(int id, int LIM, data cnt)
    {this.id=id; this.LIM=LIM; this.cnt=cnt;}
  public void run() {
    int i;
    double x,y;
    Random r=new Random();
    for(i=0;i<LIM;i++) {
      x=r.nextDouble(); y=r.nextDouble();
      if((x*x+y*y)<=1.0)cnt.count++;
    }
  }
}
```

```
PI is 3.1221796
118.910u 0.090s 0:59.73 199.2%  0+0k 0+0io 7109pf+0w
```

Cluster Computing

Monte Carlo PI – static (ii)

```
class data {
  public int count;
  public void data(){ count=0;}
  synchronized void inc() { count++;}
}
```

```
PI is 3.1424512
139.530u 14.160s 1:21.66 188.2% 0+0k 0+0io 7109pf+0w
```

Cluster Computing

Monte Carlo PI – static (iii)

```
class data {
  public int count;
  public void data(){ count=0;}
  synchronized void inc(int by) {count+=by;}
}
```

```
PI is 3.1424888
117.180u 0.200s 0:59.04 198.8%  0+0k 0+0io 7109pf+0w
```

Cluster Computing

Monte Carlo Pi - Dynamic

```
class data {
  public int count;
  private int LIM,task;
  public data(int LIM, int task)
    {this.LIM=LIM; this.task=task; count=0;}
  synchronized int get_job() {
    if(LIM>0){LIM-=task; return task;}
    return 0;
  }
  synchronized void inc(int by){ count+=by; }
}
....//Worker
 LIM=cnt.get_job();
    while(LIM>0){
....//Main
cnt=new data(LIM,100);
```

```
PI is 3.14221
121.790u 0.830s 1:01.70 198.7%  0+0k 0+0io 7109pf+0w
```

Cluster Computing

Monte Carlo Pi – Specialized Worker

```
public class bounded_buffer {
  random_coor [] buf;
  int size, head, tail;
  bounded_buffer(int len){
    size=len; buf=new random_coor[len];
    head=0; tail=len-1;
  }
  synchronized void put(random_coor c){
    if(tail==head)notify();
    buf[head]=c;
    head=(head+1)%size;
    if(head==tail) try{wait();}catch(Exception e){};
  }
  synchronized random_coor get(){
    if(tail==head)notify();
    tail=(tail+1)%size;
    if(head==tail) try{wait();}catch(Exception e){};
    return buf[tail];
  }
}
```

Cluster Computing

Monte Carlo Pi – Specialized Worker

```
class Producer extends Thread {
  int LIM;
  random_coor c;
  bounded_buffer b;
  Producer(bounded_buffer b, int LIM)
    {this.b=b; this.LIM=LIM;}
  public void run(){
    int i;
    Random r=new Random();
    for(i=0;i<LIM;i++) {
      c=new random_coor();
      c.x=r.nextDouble(); c.y=r.nextDouble();
      b.put(c);
    }
  }
}
```

Cluster Computing

Monte Carlo Pi – Specialized Worker

```
class Consumer extends Thread {
  int LIM, hit;
  random_coor c;
  bounded_buffer b;
  Consumer(bounded_buffer b, int LIM)
    {this.b=b; this.LIM=LIM;}
  public void run(){
    int i;
    random_coor c;
    for(i=0;i<LIM;i++){
      c=b.get();
      if(c==null)
        System.out.println("Strange NULL returned as "+i);
      if((c.x*c.x+c.y*c.y)<=1.0)hit++;
    }
  }
}
```

Cluster Computing

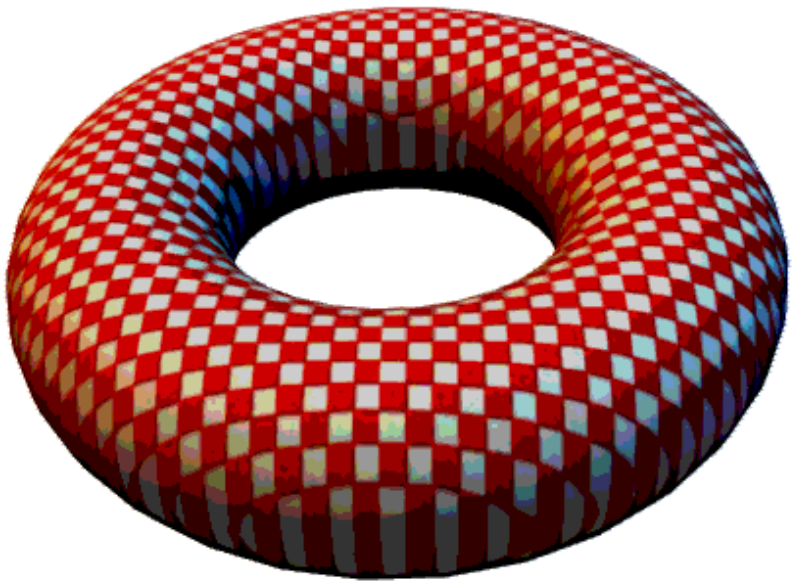
Monte Carlo Pi – Specialized Worker

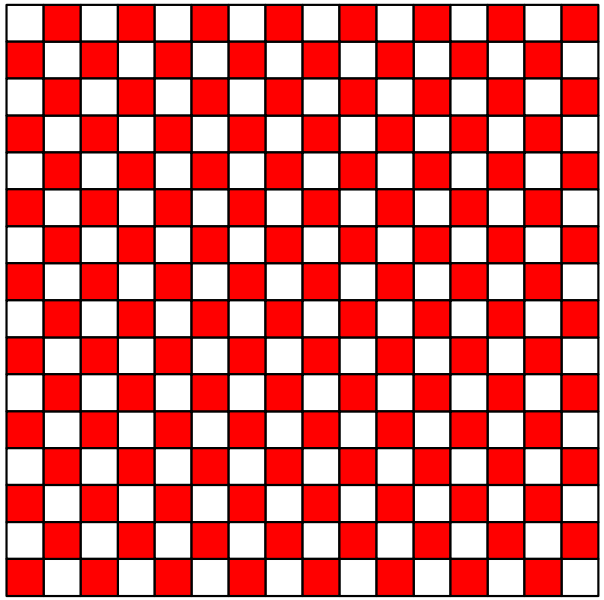
```
public class special_worker_mcpi {
  public static void main(String argv[]){
    int LIM=10000000;
    bounded_buffer b=new bounded_buffer(128);
    Producer p=new Producer(b,LIM);
    Consumer c=new Consumer(b,LIM);
    p.start(); c.start();
    try {p.join(); c.join();} catch (InterruptedException e) {};
    System.out.println("PI is " + 4.0*c.hit/LIM);
  }
}
```

Output needed here!

Cluster Computing

WATOR





Cluster Computing

Parallel WATOR

- Is WATOR embarrassingly parallel like Monte Carlo Pi?
  - Why not?
  - Can we solve this?

Cluster Computing

WATOR Sequential

```
main() {
  int i;
  initworld();
  for(i=0; i< TIMESTEPS; i++){
    movefish();
    movesharks();
  }
}
```

Cluster Computing

WATOR – static scheduled

```
typedef struct {int from, to} job;

void * worker(job *myjob){
        int i;
  for(i=0;i<myjob->itt; i++{
    move_fish(myjob->from, i<myjob->to);
    barrier();
    move_sharks(myjob->from, i<myjob->to);
    barrier();
    return (void *)1;
}
```

Cluster Computing

WATOR – static scheduled

```
main() {
  int blocksize=worldsize/nthreads;
  job jobs[WORKERS];
  pthread_t t[WORKERS];
  for (i=0; i<nthreads; i++){
     jobs[i].from=i*blocksize;
     jobs[i].to=(i+1)*blocksize;
}
  jobs[nthreads-1].to=worldsize;
  initworld();
  for(j=0;j<nthreads;j++)
    pthread_create(&(t[j], NULL, worker,(void *)(&jobs[j]));
  for(j=0;j<nthreads;j++)
    pthread_join(t[j]);
}
```

Cluster Computing

WATOR Fork-join

```
main(){
  int blocksize=worldsize/nthreads;
  job jobs[WORKERS];
  pthread_t t[WORKERS];
  for (i=0; i<nthreads; i++){
    jobs[i].from=i*blocksize;
    jobs[i].to=(i+1)*blocksize;
  }
  jobs[nthreads-1].to=worldsize;
  initworld();
  for(i=0; i<timesteps; i++){
    for(j=0;j<nthreads;j++)
      pthread_create(&(t[j], NULL, move_fish,(void *)(&jobs[j]));
    for(j=0;j<nthreads;j++)
      pthread_join(t[j]);
    for(j=0;j<nthreads;j++)
      pthread_create(&(t[j], NULL, move_sharks,(void *)(&jobs[j]));
    for(j=0;j<nthreads;j++)
      pthread_join(t[j]);
}
```

Cluster Computing

WATOR OpenMP

```
move_fish(){
//Local declarations here
#pragma omp parallel for
for(i=0; i<WORLDSIZE; i++)
  for(j=0;j<WORLDSIZE; j++)
    //Handle each point (i,j)
```