COSCO IV

Multicore Programming I (OpenMP)

d8141104 Akram Ben Ahmed
m5171114 Kazuki Kobayashi
s1190106 Kosuke Hongo
s1190209 Yusuke Sato

Outline

- Background
- Parallel computing platform
- Understanding performance
- Multi threading and multicore platform
- API for multicore programming
- Demonstration using OpenMP
- Conclusion

Background

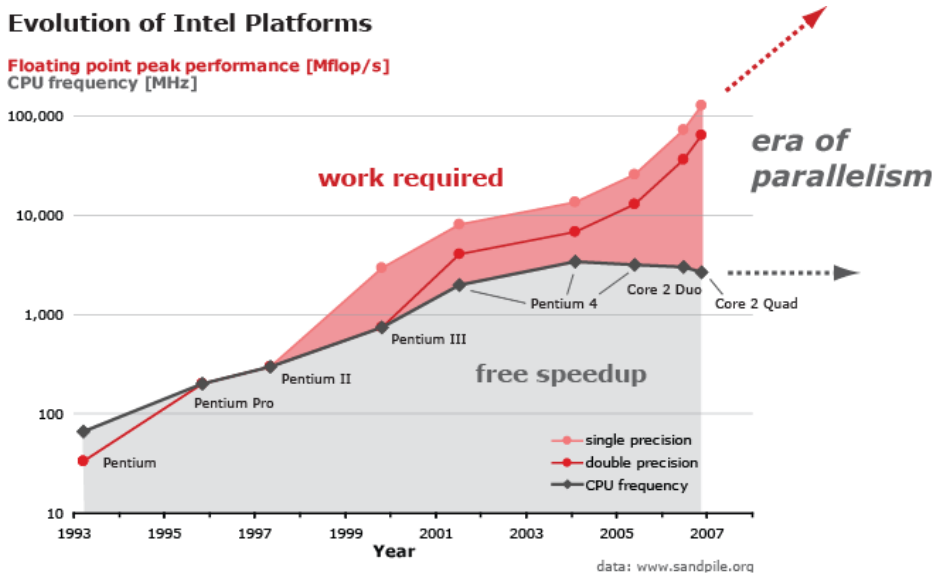
From 1980s to early 2000s

realize increase in performance by increasing clock speeds

Limitation of Moore's law

Increase of

- power consumption
- amount of heat generation

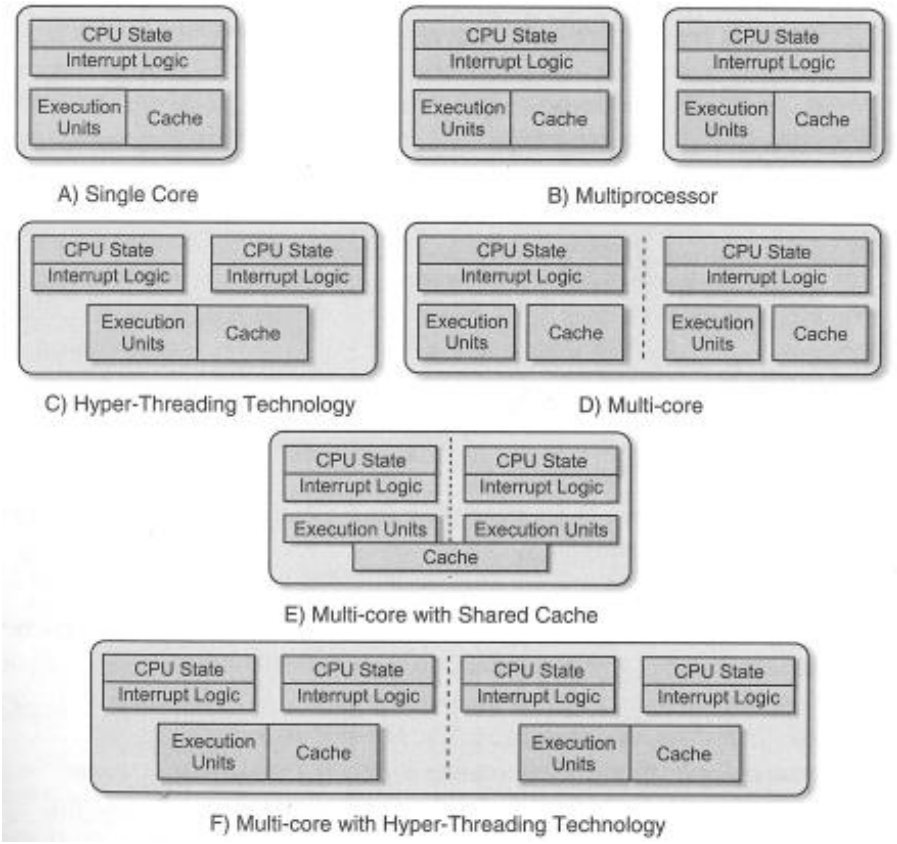


Background

- Limitation of performance increase to increase clock speeds
- Era of parallel computing
  - Hyper threading on single core platform
  - Multithreading on multicore platform

Parallel computing platform

Comparison of single core, multicore, and multi-processor



Parallel computing platform

Differences between multicore and hyper threading

- Multicore platform
  - Embed two or more independent execution cores
- Hyper threading
  - Scheduled multiple threads to logical processors as they would on multiprocessor systems
  - Single execution core shared by multiple thread

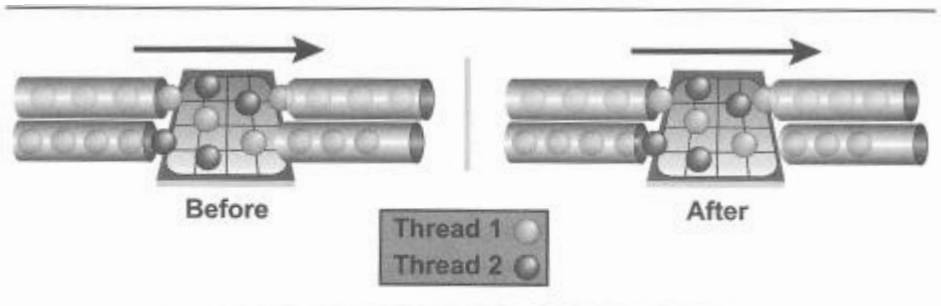


Figure 1.5 Two Threads Executing on a Processor with Hyper-Threading Technology

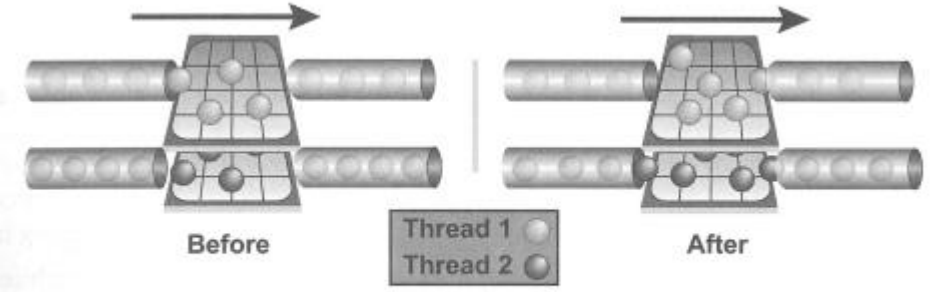


Figure 1.6 Two Threads on a Dual-Core Processor with each Thread Running Independently

Understanding performance

Runtime= S+P

- S is the time spent in serial code
- P is the time spent in parallel code

$$Runtime = S + \frac{P}{N}$$

- N is the number of threads

Understanding performance

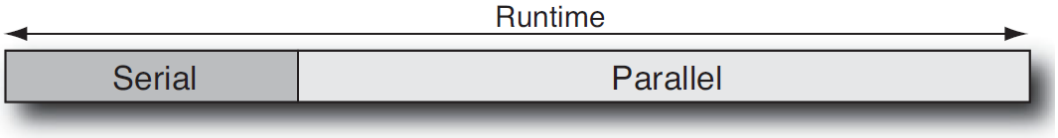


Figure 3.7 Single-threaded runtime

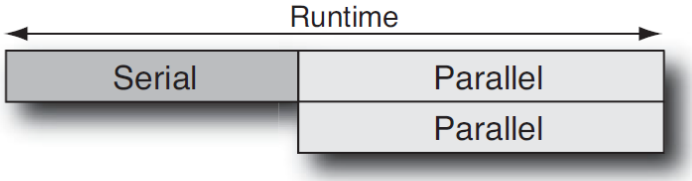


Figure 3.8 Runtime with two threads

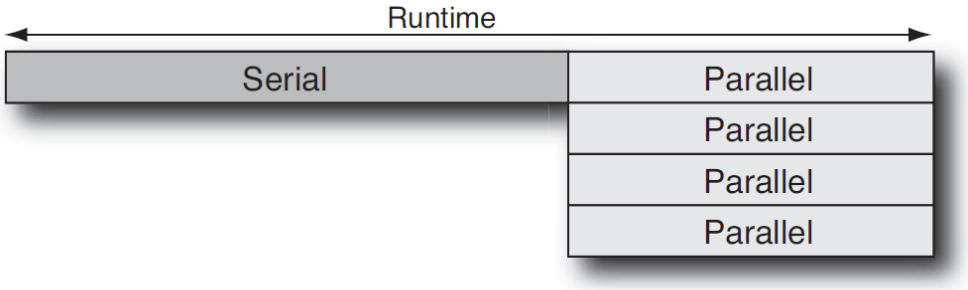


Figure 3.9 Runtime with four threads

Understanding performance

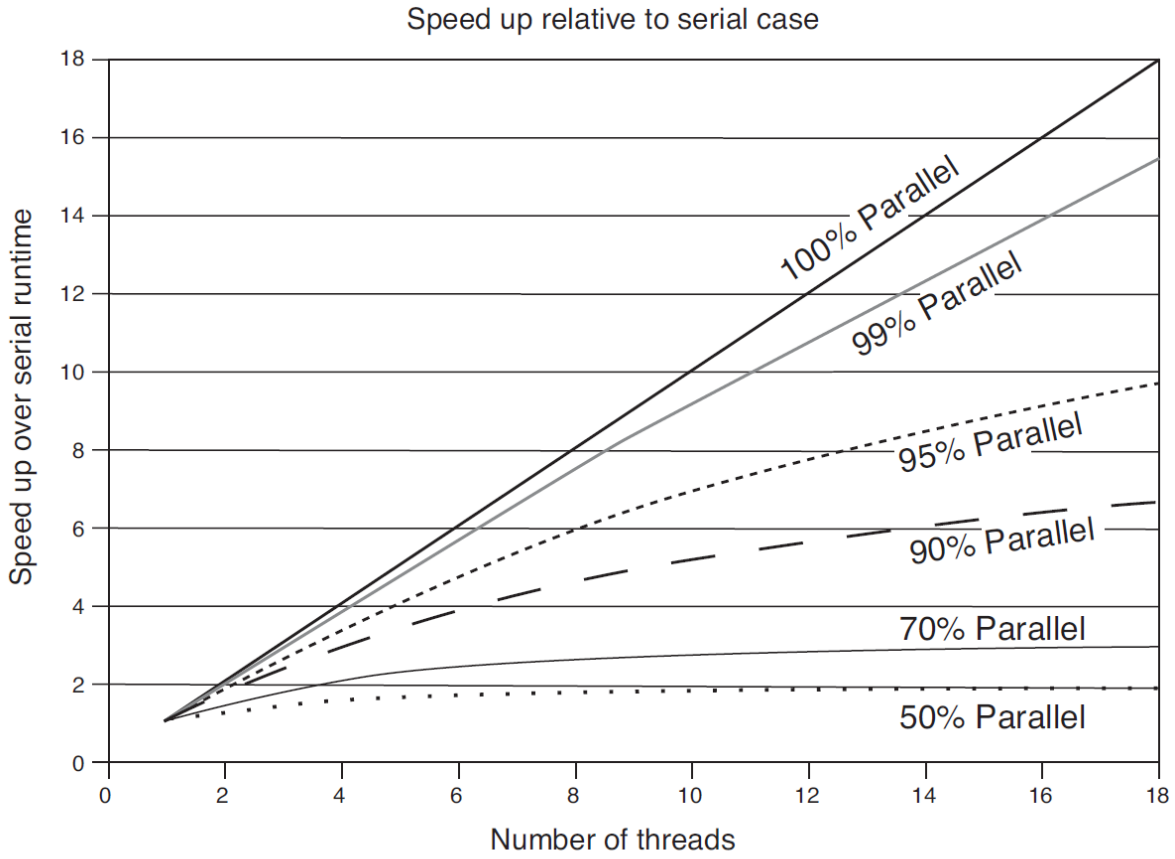


Figure 3.10 Scaling with diminishing parallel regions

Understanding performance

$$\text{Runtime}_N = S + \frac{P}{N}$$

$$\text{Acceptable runtime} = S * (1 + T)$$

$$S * (1 + T) = \left(S + \frac{P}{N} \right)$$

$$\Rightarrow \quad N = \frac{P}{ST} = \frac{P}{(1 - P)T}$$

- T is the tolerance within which users cease to care about a difference in performance.
- It is usually 10%: if the performance that they get is within 10% of the best possible, then it is acceptable.

Understanding performance

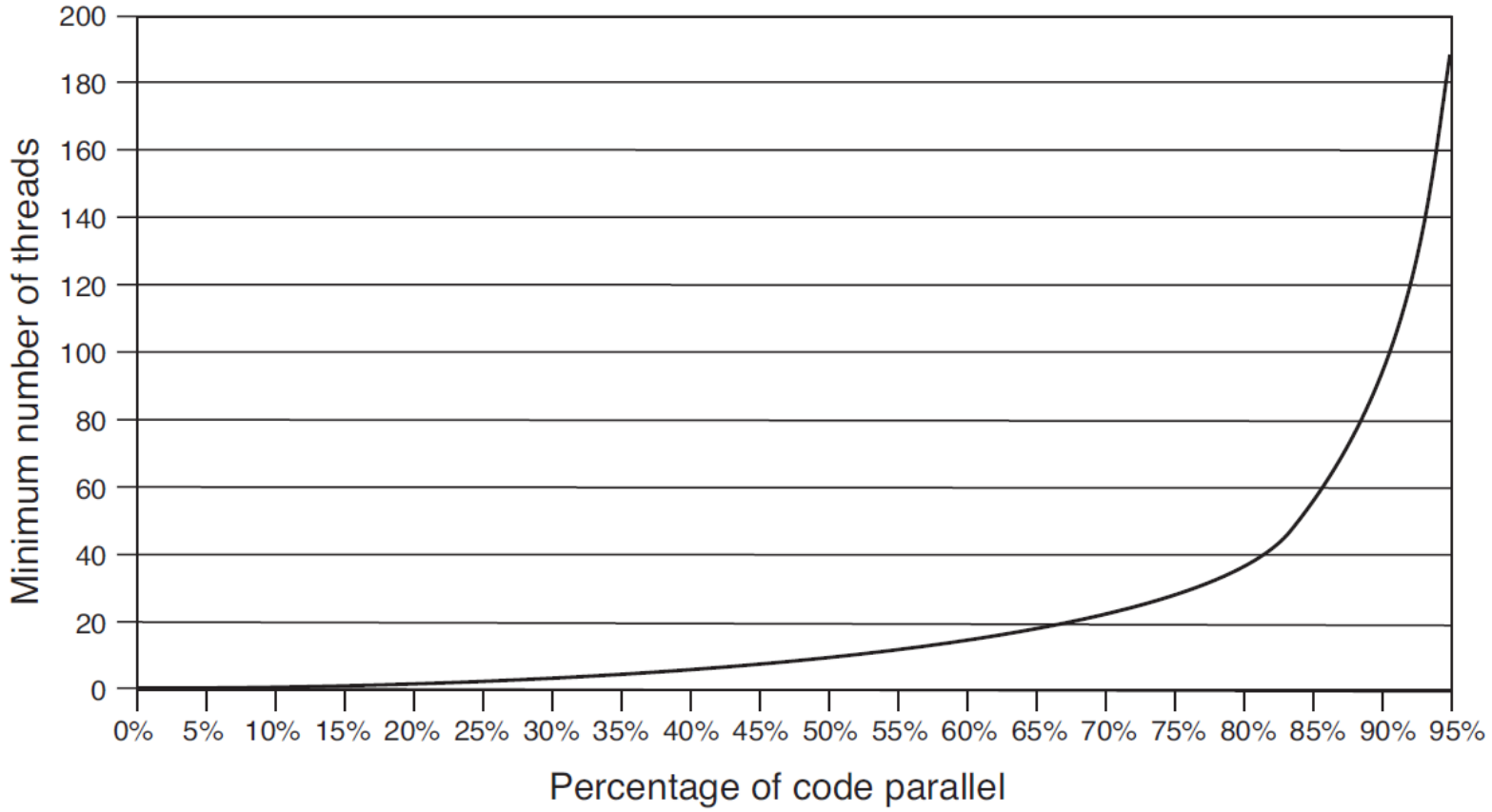


Figure 3.11 Minimum number of threads required to get 90% of peak performance

Understanding performance

- At the presence of multiple threads, a synchronization delay is needed to provide the necessary communication between the different threads.
- The number of threads should not exceed a certain number to avoid that the performance degrades.

Understanding performance

- A function $F_{(N)} = \mathrm{K} \times \ln(N)$ is used to denote the synchronization cost
  - K is constant which depends on the memory or cache latency of the system
  - N is the number of threads

$$Runtime = S + \frac{P}{N} + \mathrm{K} \times \ln(N)$$

$$\Rightarrow \quad \mathrm{N} = \frac{P}{K}$$

Understanding performance

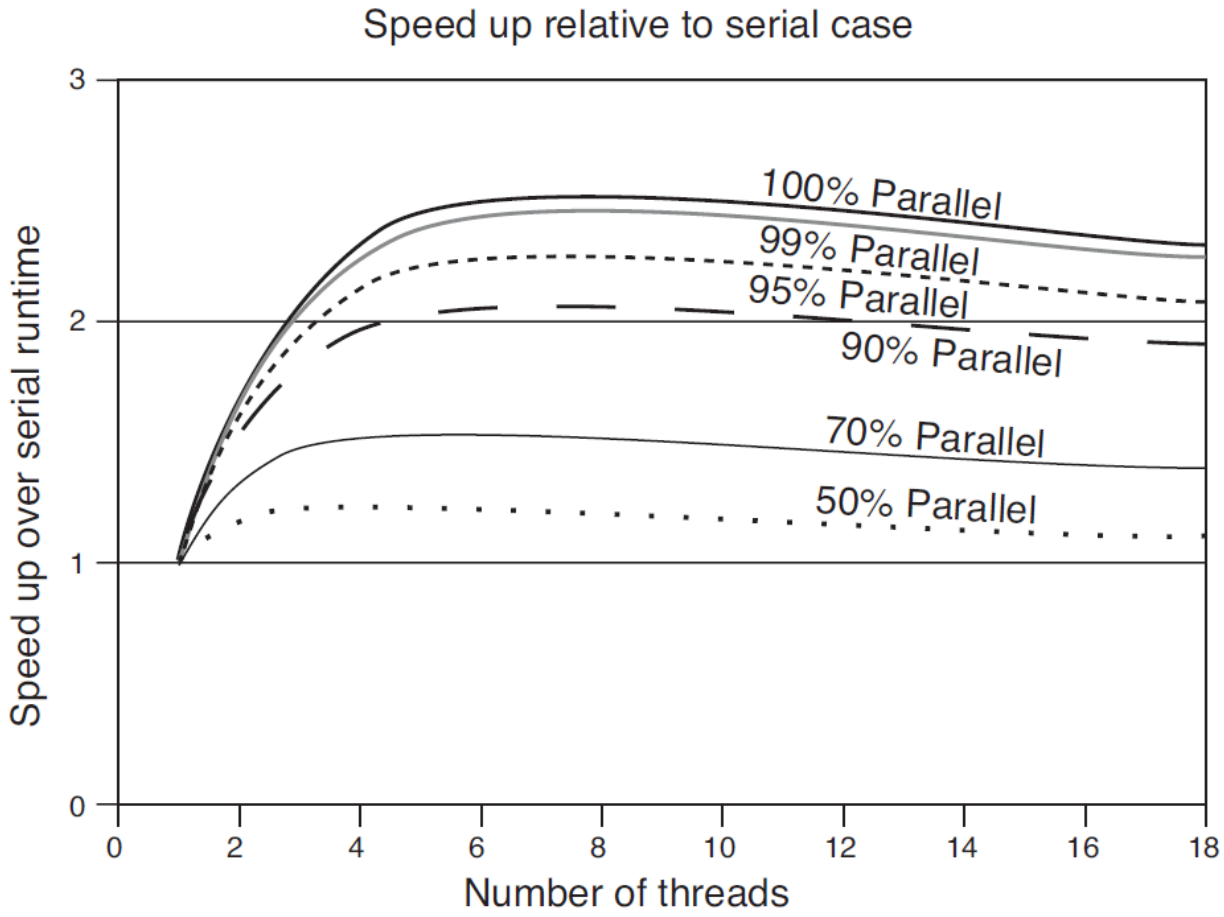


Figure 3.12 Scaling with exaggerated synchronization overheads

Multi threading and multicore platform

- Some problems to synchronize each core to process multiple threads
  - Critical section
  - Deadlocks
- Synchronization primitive to solve these problems
  - Semaphores
  - Locks

Synchronization

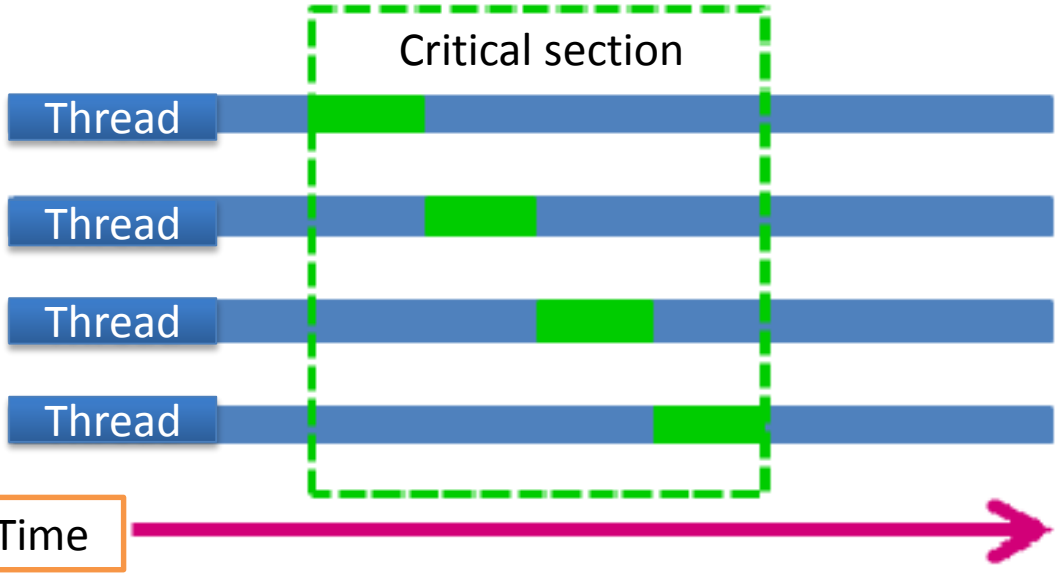
- Synchronization is used to coordinate the activity of multiple threads.
  - Ensure that shared resources are not accessed by multiple threads simultaneously.
  - All work on those resources is complete before new work starts.
- Most operating systems provide a rich set of synchronization primitives.

Outline

- Critical section
- Deadlocks
- Synchronization primitive
  - Semaphores
  - Locks
- API for Multicore programming
  - MPI and OpenMP

Critical section

- Critical Section
  - A section of a code block is where shared variables have dependency among multiple threads.
  - Deferent synchronization primitives are used to keep critical sections safe.



Critical section

- Critical Section
  - The critical sections use if multiple threads perform mutually exclusive operation for critical section and do not use critical sections simultaneously.
  - Each critical section has an entry and an exit point.

C言語

```
int myTask, nextTask;
nextTask = 1;
#pragma omp parallel Critical
{
    myTask = nextTask;
    nextTask = nextTask + 1;
}
```

Deadlocks

- Deadlocks
  - *Deadlock*, where two or more threads cannot make progress because the resources that they need are held by the other threads.

Thread 1

```
void update1()
{
   acquire(A);
   acquire(B); <<< Thread 1
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

Thread 2

```
void update2()
{
   acquire(B);
   acquire(A); <<< Thread 2
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order allowing the 2 thread to make progress

Synchronization Primitives

- Synchronization is typically performed by three types of primitives.
  1. Semaphores
  2. Locks
     1. Mutexes
     2. Recursive Locks
     3. Readers-Writers Locks
     4. Spin Locks

Synchronization Primitives

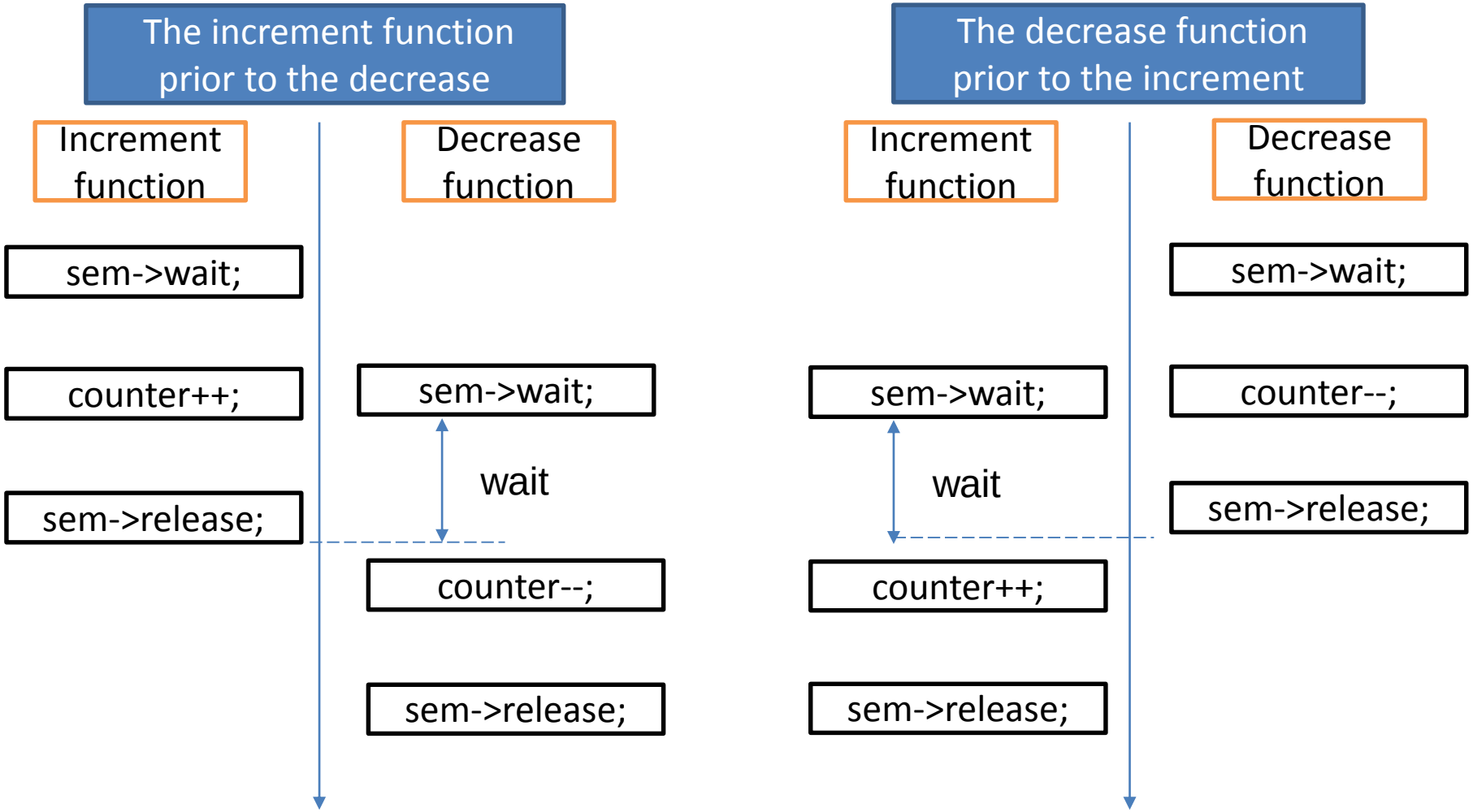
- Semaphores
  - *Semaphores* are counters that can be either incremented or decremented. They can be used in situations where there is a finite limit to a resource and a mechanism is needed to impose that limit.
  - When used for a pool of resources, a semaphore tracks only *how many* resources are free; it does not keep track of *which* of the resources are free.

```
semaphore sem;

void Increment()
{
  while(1){
    sem->wait();
    counter++;
    sem->release();
  }
}

void Decrease()
{
  while(1){
    sem->wait();
    counter--;
    sem->release();
  }
}
```

Synchronization Primitives



Synchronization Primitives

- Semaphores
  - A semaphore is a way to limit the number of tasks that can simultaneously operate on a shared (protected) resource.
  - A semaphore is similar to a mutex. A mutex is a binary semaphore (1,0)

Implementation

wait(): Decrements the value of semaphore variable by 1

signal(): Increments the value of semaphore variable by 1

Synchronization Primitives

- Mutexes and Critical Regions
  - The simplest form of synchronization is a mutually exclusive (mutex) lock.
  - Only one thread at a time can acquire a mutex lock to ensure that the data structure is modified by only one thread at a time.

```
int counter;

mutex_lock mutex;

void Increment()
{
   acquire( &mutex );
   counter++;
   release( &mutex );
}

void Decrement()
{
   acquire( &mutex );
   counter--;
   release( &mutex );
}
```

Synchronization Primitives

- Mutexes and Critical Regions
  - The region of code between the acquisition and release of a mutex lock is called a *critical section*, or *critical region*.
  - Code in this region will be executed by only one thread at a time.

```
int counter;

mutex_lock mutex;

void Increment()
{
   acquire( &mutex );
   counter++;
   release( &mutex );
}

void Decrement()
{
   acquire( &mutex );
   counter--;
   release( &mutex );
}
```

Synchronization Primitives

- Recursive Locks

- Recursive locks are locks that may be repeatedly acquired by the thread that currently owns the lock without causing the thread to deadlock.

- No other thread may acquire a recursive lock until the owner releases it once for each time the owner acquired it.

```
Recursive_Lock L;
void recuresiveFunction(int counter){
  L->acquire();
  if (counter>0){
    counter = counter - 1;
    recursiveFunction(counter);
  }
  L->release();
}
```

Synchronization Primitives

- Readers-Writer Locks
  - Multiple threads reading the shared data do not present a problem. Read-only data does not, therefore, need protection with some kind of lock.
  - Data that is read-only needs to be updated. A *readers-writer lock* allows many threads to read the shared data but can then lock the readers threads out to allow one thread to acquire a writer lock to modify the data.

Listing 4.9 Using a Readers-Writer Lock

```
int readData( int cell1, int cell2 )
{
  acquireReaderLock( &lock );
  int result = data[cell] + data[cell2];
  releaseReaderLock( &lock );
  return result;
}

void writeData( int cell1, int cell2, int value )
{
  acquireWriterLock( &lock );
  data[cell1] += value;
  data[cell2] -= value;
  releaseWriterLock( &lock );
}
```

Synchronization Primitives

- Spin Locks
  - *Spin locks* are essentially mutex locks. The difference between a mutex lock and a spin lock is that a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping.
  - The advantage of using spin locks is that they will acquire the lock as soon as it is released, whereas a mutex lock will need to be woken by the operating system before it can get the lock.

Acquiring a Spin Lock

pthread_spin_lock()

release a Spin Lock

pthread_spin_unlock

API for Multicore programming

- MPI
  - designed for high performance on both massively parallel machines and on workstation clusters.
- OpenMP(Open Multi-Processing)
  - allows for parallel programming in a shared memory machine.

API for Multicore programming

	MPI	**OpenMP**
Advantage	• Both of distributed system and shared memory system are available • Controlling data placement is simple	• Parallelization is easy • High bandwidth and low latency • Dynamic load balancing is possible
Disadvantage	• Debugging and program development is difficult • Low bandwidth and high latency • Dynamic load balancing is difficult	• Only shared memory system • Node size is the limitation of scalability • Fine controlling of threads are difficult • Data placement may become the problem

Environment

- CPU
  - Use 4 cores PC
- Cygwin
  - 4.8.3-1 gcc-core
    - OpenMP included

How to parallelize

- Use commands on Cygwin
  - Choose threads number

```
s1190106@s1190106-PC /cygdrive/c/Users/s1190106/
% export OMP_NUM_THREADS=“number”
```

  - Compile to “FILE_NAME” on single thread

```
%gcc “FILE_NAME”
```

  - Compile to “FILE_NAME” on multi threads

```
%gcc –fopenmp –lgomp “FILE_NAME”
```

Introduction

- Introduce to use OpenMP code
  - Display the “Hello world” of 10 times in this program

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

int main(){
        int i;

#pragma omp parallel for
        for( i=0 ;i<10 ; i++){
        printf("Hello World%d!¥n",i);
        }
}
```

Introduction

- Explain the code
  - #include<omp.h>
    - Need to use OpenMP
  - #pragma omp parallel
    - Parallel processing in block
  - For(written after the "parallel")
    - Must be written in parallel block
    - Separate For processing

Compare Time

- Compare to single thread and multi threads
  - Execute the For loop of 10000000 times in a program
  - Resut table

Threads numbers	Single thread	2-threads	3-threads	4-threads	5-threads
Time[ms]	238	156	140	171	171

Problems and Solutions

- Loop-carried dependence
- Data-race conditions
- Managing shared and private data
- Loop Scheduling and partitioning
- Effective use of reductions

Loop-carried dependence

- Calculate to use previous results
- Difficult separate loop in this problem

Statement A or B	Iterator k-1	Iterator k
Statement A	Read x[k-1]	
Statement B		Write y[k]
Statement A		Write x[k]
Statement B	Read y[k-1]	

Loop-carried dependence

- Introduce Program
  - Repeat to add 1 to y[k-1] and 2 to x[k-1] in this program(correct result x = 149, y = 149)

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

main()
{
int k;
int x[100];
int y[100];
```

```
x[0] = 0;
y[0] = 1;

#pragma omp parallel for
for ( k = 1; k < 100; k++){
        x[k] = y[k-1] + 1;//S1
        y[k] = x[k-1] + 2;//S2
printf("X = %d, Y = %d, k = %d¥n",x[k],y[k],k );
                }
}
```

Loop-carried dependence

- Reason to fail
  - "x[k]" depend on "y[k-1]"
  - "y[k]" depend on "x[k-1]"
  - can't depend on to separate threads
  - Automatically separate the threads by "parallel for"
    - $Separation\ loop\ number = \frac{maximum\ loop\ number}{Threads\ number}$

Loop-carried dependence

- Solution
  - Separate the loop by self
    - Initialize at separation point

```
#include<stdio.h>
#include<omp.h>
main()
{
int k;
int m;
int x[100],y[100];

x[0] = 0;  y[0] = 1;
x[49] = 74; y[49] = 74;
```

```
#pragma omp parallel for private(m,k)
for( m = 0; m < 2; m++){
        for ( k = m*49+1 ; k < m*50+50; k++){
        x[k] = y[k-1] + 1;//S1
        y[k] = x[k-1] + 2;//S2
        printf("X = %d, Y = %d, k = %d¥n",x[k],y[k],k );
                }
        }
}
```

Loop-carried dependence

- Another solution
  - Use “sections” pragma

```
#include<stdio.h>
#include<omp.h>
main(){
            int k, x[100],y[100];
#pragma omp parallel sections private(k)
{{
 x[0] = 0; y[0] = 1;
 for ( k = 1; k < 49; k++){
   x[k] = y[k-1] + 1;//S1
   y[k] = x[k-1] + 2;//S2
   printf("X = %d, Y = %d, k =
%d¥n",x[k],y[k],k );}}
```

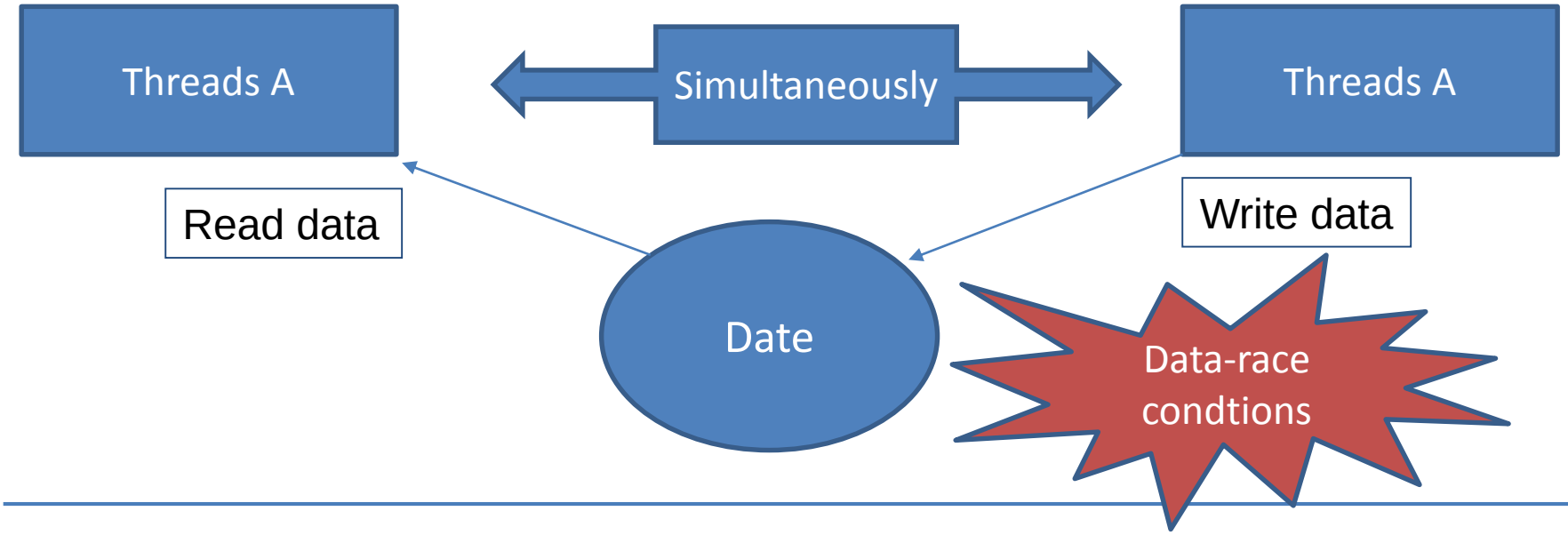
```
#pragma omp section
{
 x[49] = 74; y[49] = 74;
 for ( k = 50; k < 100; k++){
   x[k] = y[k-1] + 1;//S3
   y[k] = x[k-1] + 2;//S4
   printf("X = %d, Y = %d, k =
%d¥n",x[k],y[k],k );}}
            }
}
```

Loop-carried dependence

- Explain the code
  - "sections"
    - Execute one section block on one thread
  - "private(x)"
    - Have x (independent variable) in each the threads

Data-race conditions

- Simultaneously write data and read data
  - Get to not expect result
    - Write incorrect data
    - Read incorrect data



Data-race conditions

- Introduce Program
  - Display to 1 to 4 for two times.

Correct results	
X=0, i= 0	#include<stdio.h>
X=1, i= 0	#include<omp.h>
X=0, i= 1	
X=1, i= 1	main(){
X=0, i= 2	int x = 0,y=0,i=0;
X=1, i= 2	#pragma omp parallel for
X=0, i= 3	for (i = 0; i < 5; i++){
X=1, i= 3	for(x = 0; x < 2; x++){
X=0, i= 4	printf("X = %d, i = %d¥n", x, i);
X=1, i= 4	}}}

Data-race conditions

- Reason to fail
  - Simultaneously write data and read data
    - happen data-race condition at “i++” or “x++” in this program
- Solution
  - Assign a variable into a each thread
  - Use “private()” pragma

Data-race conditions

- Solution program
  - Add private pragma after for pragma

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

main(){
int x = 0,y=0,i=0;
#pragma omp parallel for private(i,x)
for ( i = 0; i < 5; i++ ){
   for(x = 0; x < 2; x++){
   printf( "X = %d,  i = %d¥n", x, i );
              }}}
```

Managing shared and private data

- Manage shared variable and private variable
- Use Managing shared pragma
  - "shared(x)"
    - Declare to share variable x
  - "default(shared or none)"
    - Declare to share variable for all variable in parallel execution(choose shared)
    - Declare no type
    - Choose variable type in "private", "shared", "firstprivate", "lastprivate"

Managing shared and private data

- Use managing private pragma
  - "private(x)"
    - Have x (independent variable) in each the threads
  - "firstprivate(x)"
    - Synchronize x value(master thread) in all threads in the beginning
  - "lastprivate(x)"
    - Synchronize x value(master thread) in all threads in the end

Managing shared and private data

- Introduce Program
  - Display array 0 to 3(each array content is 0,1,2,3)

```
#include<stdio.h>
#include<omp.h>
int do_work(int);
main(){
int k,x =0;
int array[4];
#pragma omp parallel for
for(k = 0; k < 2; k++){
        for(x = 0; x < 2; x++){
        array[2*x+k] = do_work(2*x+k);
```

```
printf("x = %d, array[%d] = %d¥n",
x, 2*x+k, array[2*x+k] );
                        }}}

int do_work(int a){
        return a;
}
```

Managing shared and private data

- Reason to fail
  - happen loop-carried dependence or data-race condition
  - happen data-race condition in this program
- Solution
  - Use “private” pragma

Managing shared and private data

- Solution program
  - Add private pragma after for pragma

```
#include<stdio.h>
#include<omp.h>
int do_work(int);
main(){
int k,x =0;
int array[4];
#pragma omp parallel for pivate(k,x)
for(k = 0; k < 2; k++){
        for(x = 0; x < 2; x++){
        array[2*x+k] = do_work(2*x+k);
```

```
printf("x = %d, array[%d] = %d¥n",
x, 2*x+k, array[2*x+k] );
                        }}}

int do_work(int a){
        return a;
}
```

Managing shared and private data

- Another Solution

```
#include<stdio.h>
#include<omp.h>
int do_work(int);
main(){
int k,x =0;
int array[4];
#pragma omp parallel for
for(k = 0; k < 2; k++){
  int x;
  for(x = 0; x < 2; x++){
  array[2*x+k] = do_work(2*x+k);
```

```
printf("x = %d, array[%d] = %d¥n",
x, 2*x+k, array[2*x+k] );
                              }}}

int do_work(int a){
            return a;
}
```

Loop Scheduling and partitioning

- Have 4 kind Scheduling in OpenMP
  - Use to scheduling
    - Static(default)
    - Dynamic
    - Guided
    - Runtime
  - How to use
    - “#pragma omp for schedule(kind, chunk-size)”

Loop Scheduling and partitioning

- Chunk
  - Loop count in each threads

Schedule Type	Description
Static	Default chunk-size = (all loop count) / (thread number) Assign chunk to each threads before execution
Dynamic	Default chunk-size = 1 Assign chunk to free thread in execution
Guided	Default chunk-size = 1 Assign chunk to free thread, chunk size automatically is assign
Runtime	Default chunk-size = (all loop count) / (thread number) Uses the OMP_SCHEDULE environment variable

Loop Scheduling and partitioning

- Sample program
  - Display twice k (0<k<100)
  - Run the loops 8 times on each threads
  - Repeat the loops to the last on each threads.
- Explain code
  - schedule(dynamic, 8)
    - Assign chunk to free thread in execution
    - Chunk size is 8
  - omp_get_thread_num()
    - obtain to use the thread number.

Loop Scheduling and partitioning

- Sample program

```
#include<stdio.h>
#include<omp.h>
main(){
int k, x;
#pragma omp parallel for schedule(dynamic, 8)
for(k = 0; k < 100; k++) {
    x = k + k;
    printf("thread number %d, k = %d, x = %d¥n", omp_get_thread_num(),k,x);
            }
}
```

Effective use of reductions

- "reducution" pragma
  - Use to calculate faster in parallel calculation.
  - Should use to get correct calculation result in parallel execution

Effective use of reductions

- Introduce Program
  - Add 0 to 100(correct answer is 4590)

```
#include<stdio.h>
#include<omp.h>
main(){
int k, sum= 0;
#pragma omp parallel for
for(k = 0; k < 100; k++){
  sum += k;
  }
printf("%d¥n",sum );
}
```

Effective use of reductions

- Reason to fail
  - Get to correct answer in this program
  - May to output to the incorrect calculation result
- Solution
  - Use “reduction(operator:variavle)” pragma
  - Able to use the following operator
    - +, -, *, &, ^, |, &&, ||

Effective use of reductions

- Solution program

```
#include<stdio.h>
#include<omp.h>
main(){
int k, sum= 0;
#pragma omp parallel for reduction(+:sum)
for(k = 0; k < 100; k++){
  sum += k;
  }
printf("%d¥n",sum );
}
```

Conclusion

- some problems and solutions in the multicore programing
- Chose OpenMP in parallel program in this time
- Explained how to use OpenMP