# Chapter 2:
Perspectives on Parallel Programming

Module 2.1
Parallel Programming Models

Programming Models

- What is programming model?
  - An abstraction provided by the hardware to programmers
  - Determines how easy/difficult for programmers to express their algorithms into computation tasks that the hardware understands

- Uniprocessor programming model
  - Based on program + data
  - Bundled in Instruction Set Architecture (ISA)
  - Highly successful in hiding hardware from programmers

- Multiprocessor programming model
  - Much debate, still searching for the right one…
  - Most popular: shared memory and message passing

Shared Mem vs. Msg Passing

Shared Memory Model

P   P   P   ...   P

st   ld

Memory

Message Passing Model

P   P   P   ...   P

M   M   M   M

send   recv
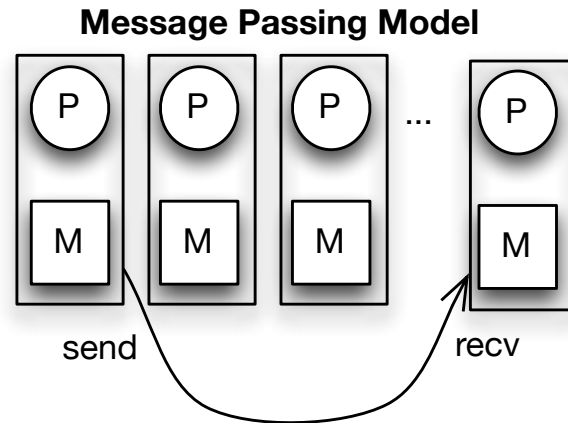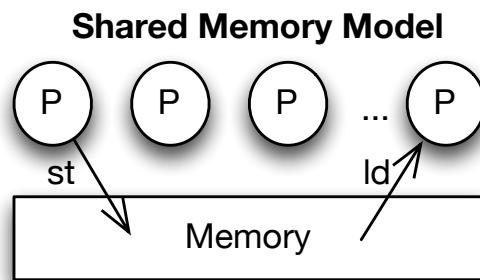
- Shared Memory / Shared Address Space
  – Each memory location visible to all processors
- Message Passing
  – Each memory location visible to 1 processor

Thread/process – Uniproc analogy

Process: share nothing

```
if (fork() == 0)
    printf("I am the child process, my id is %d", getpid());
else
    printf("I am the parent process, my id is %d", getpid());
```

-heavyweight => high thread creation overhead
-The processes share nothing => explicit communication using
  socket, file, or messages

Thread: share everything

```
void sayhello() {
  printf("I am child thread, my id is %d", getpid());
}

printf("I am the parent thread, my id is %d", getpid());
clone(&sayhello,<stackarg>,<flags>,())
```

+ lightweight => small thread creation overhead
+ The processes share addr space => implicit communication

Thread communication analogy

```
int a, b, signal;
…
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf("child thread> sum is %d", a + b);
  signal = 0;  // my work is done
}

void main() {
  a = 5, b = 3;
  signal = 0;
  clone(&dosum,…)          // spawn child thread
  signal = 1;              // tell child to work
  while (signal == 1) {}   // wait until child done
  printf("all done, exiting\n");
}
```

- Shared memory in multiproc provides similar memory sharing abstraction

Message Passing Example

```
Int a, b;
…
void dosum() {
  recvMsg(mainID, &a, &b);
  printf("child process> sum is %d", a + b);
}

Void main() {
  if (fork() == 0)  // I am the child process
    dosum();
  else {            // I am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
    wait(childID);
    printf("all done, exiting\n");
  }
}
```

Differences with shared memory:
• Explicit communication
• Message send and receive provide automatic synchronization

Quantitative Comparison

Table 2.1: Comparing shared memory and message passing programming models.

Aspects	Shared Memory	Message Passing
Communication	implicit (via loads/stores)	explicit messages
Synchronization	explicit	implicit (via messages)
Hardware support	typically required	none
Development effort	lower	higher
Tuning effort	higher	lower

Development vs. Tuning Effort

- Easier to develop shared memory programs
  - Transparent data layout
  - Transparent communication between processors
  - Code structure little changed
  - Parallelizing compiler, directive-driven compiler help
- Harder to tune shared memory programs for scalability
  - Data layout must be tuned
  - Communication pattern must be tuned
  - Machine topology matters for performance

Fundamentals of Parallel Computer Architecture - Chapter 2

More Shared Memory Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

+ Communication directly through memory.

+ Requires less code modification

- Requires privatization prior to parallel execution

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;     // necessary

end parallel // kill the child thread
Print sum;
```

More Message Passing Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

+ Communication only
  through messages

- Message sending and
  receiving overhead

- Requires algo and
  program
  modifications

```
// parent and child already spawned
id = getpid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4..7]);
else
  recv_msg (P0, &b[4..7], &c[4..7]);

for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0)
    local_sum = local_sum + a[i];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}
else
  send_msg (P0, local_sum);
```
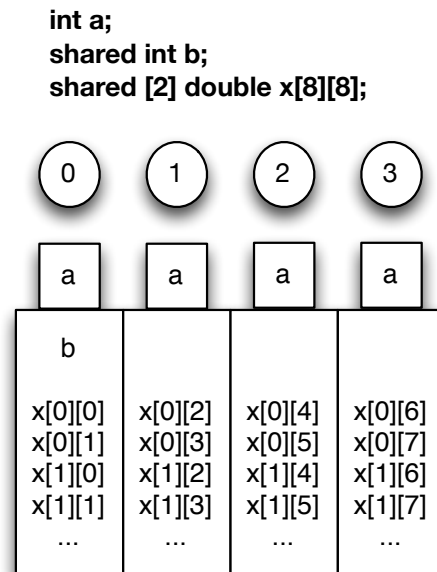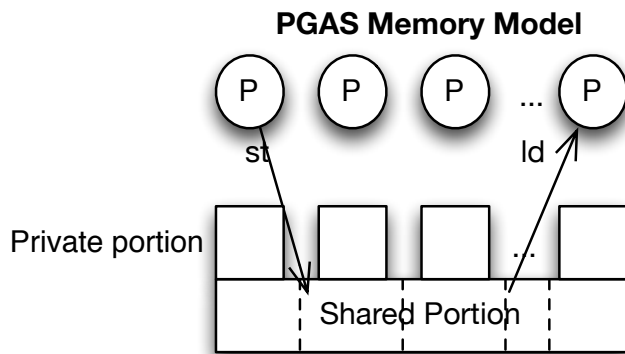
Other Programming Models

- PGAS
  - Partitioned Global Address Space
- Data Parallel
- MapReduce
- Transactional Memory

PGAS

- Shared memory model too simple for NUMA
  - Data layout is hidden from programmers
  - But thread-data proximity is important for performance
- PGAS provides shared & private address space

PGAS Memory Model

```
int a;
shared int b;
shared [2] double x[8][8];
```

Example

```
1 shared [N*P /THREADS] double A[N][P];  // distribute rows of matrix A
2 shared [N*P /THREADS] double C[N][M];  // distribute rows of matrix C
3 shared [M /THREADS] double B[P][M] ;   // distribute columns of matrix B
4
5 void main(void) {
6   ... // matrices are initialized
7
8   upc_forall(i=0;i<N;i++;&A[i][0]
9     for (j=0; j<M; j++) {
10      C[i][j] = 0;
11      for(l=0; l< P; l++)
12        C[i][j] +=A[i][l]*B[l][j];
13    }
14   upc_barrier;
15
16   ...
17 }
```
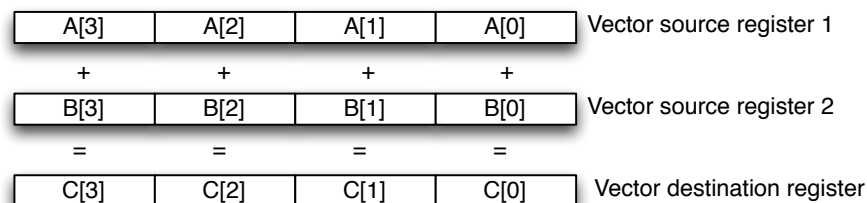
- Every node has N*P/THREADS rows of A and C
- &A[i][0] in upc_forall assigns iteration i to thread where A[i][0] is located

Fundamentals of Parallel Computer Architecture - Chapter 2

Data Parallel Model

- Data parallel = programming model for SIMD

- Either vector or scalar with lanes
  - Requires packing data into a wide register

	register width (128 bits)
	2x 64-bit data
	4x 32-bit data
	8x 16-bit data
	16x 8-bit data

  - Once packed, can compute multiple data items at once

A[3]	A[2]	A[1]	A[0]	Vector source register 1
+	+	+	+	
B[3]	B[2]	B[1]	B[0]	Vector source register 2
=	=	=	=	
C[3]	C[2]	C[1]	C[0]	Vector destination register

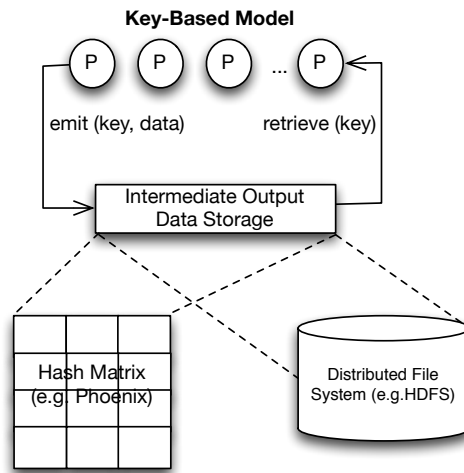Fundamentals of Parallel Computer Architecture - Chapter 2

Example

```
1 // A 128-bit vector struct with four 32-bit floats
2 struct Vector4
3 {
4    float x, y, z, w;
5 };
6
7 // Add two constant vectors and return the resulting vector
8 Vector4 SSE_Add ( const Vector4 &Operand1, const Vector4 &Operand2
9 {
10   Vector4 Result;
11
12   __asm
13   {
14     MOV EAX Operand1       // Load pointers into CPU regs
15     MOV EBX, Operand2
16
17     MOVUPS XMM0, [EAX]     // Move unaligned vectors to SSE regs
18     MOVUPS XMM1, [EBX]
19
20     ADDPS XMM0, XMM1       // Vector addition
21     MOVUPS [Result], XMM0 // Save the return vector
22   }
23   return Result;
24 }
```

Fundamentals of Parallel Computer Architecture - Chapter 2

MapReduce

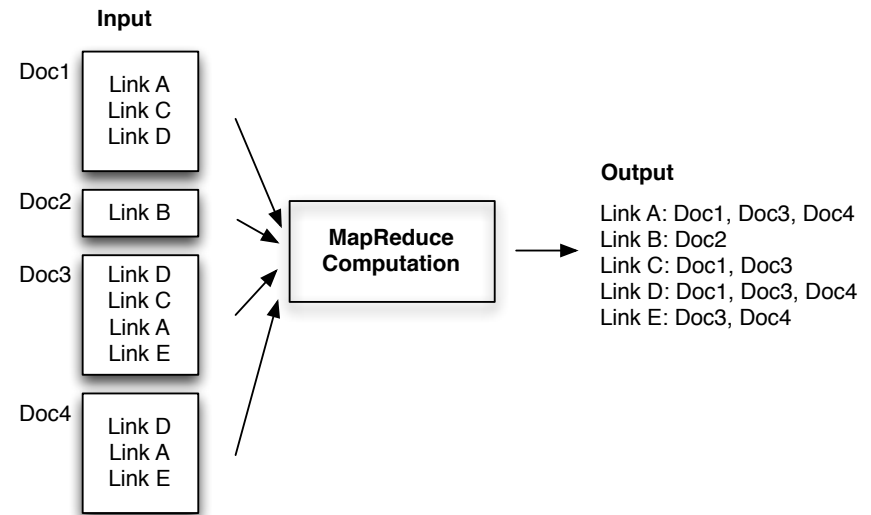- Data accessed through key rather than location
  - Can be implemented on shared memory or message passing models
- Two steps: Map (produce <key,val> pairs) and Reduce (aggregate values for each key)

Key-Based Model

emit (key, data)　　retrieve (key)

Intermediate Output Data Storage

Hash Matrix (e.g. Phoenix)

Distributed File System (e.g.HDFS)

Fundamentals of Parallel Computer Architecture - Chapter 2

Example: Inverted Index

- Files distributed over map workers

- Each map worker produces <LinkX,DocY> when it encounters LinkX on DocY

- Reduce worker aggregates all Docs having the same Link

Input

Doc1
| Link A
| Link C
| Link D

Doc2
| Link B

Doc3
| Link D
| Link C
| Link A
| Link E

Doc4
| Link D
| Link A
| Link E

MapReduce Computation

Output

Link A: Doc1, Doc3, Doc4
Link B: Doc2
Link C: Doc1, Doc3
Link D: Doc1, Doc3, Doc4
Link E: Doc3, Doc4

Transactional Memory

- Transaction = code region with ACID property (Atomicity, Consistency, Isolation, Durability)

- Transactional Memory (TM) adopts Atomicity and Isolation

  – It allows us to remove some locks and worries over deadlocks

Lock Example

```
1   image = Image_Read(fn_input);

2

3   for (i=0; i<image->row; i++) {
4     for (j=0; j<image->col; j++) {
5       lock();
6       histoRed[image->red[i][j]]++;
7       histoGreen[image->green[i][j]]++;
8       histoBlue[image->blue[i][j]]++;
9       unlock();
10    }
11  }
```

```
1   image = Image_Read(fn_input);
2   lock_t redLock[256], greenLock[256], blueLock[256];
3
4   for (i=0; i<image->row; i++) {
5     for (j=0; j<image->col; j++) {
6       lock(&redLock(image->red[i][j]));
7       histoRed[image->red[i][j]]++;
8       unlock(&redLock(image->red[i][j]));
9
10      lock(&greenLock(image->red[i][j]));
11      histoGreen[image->red[i][j]]++;
12      unlock(&greenLock(image->red[i][j]));
13
14      lock(&blueLock(image->red[i][j]));
15      histoBlue[image->blue[i][j]]++;
16      unlock(&blueLock(image->red[i][j]));
17    }
18  }
```

TM Example

```
1   image = Image_Read(fn_input);
2
3   for (i=0; i<image->row; i++) {
4     for (j=0; j<image->col; j++) {
5       atomic {
6         histoRed[image->red[i][j]]++;
7         histoGreen[image->green[i][j]]++;
8         histoBlue[image->blue[i][j]]++;
9       }
10    }
11  }
```

- Transaction enclosed by atomic {...}

- Hardware or software ensures atomicity and isolation
  - SW inflexible and slow
  - HW expensive

- No locks needed and no locking overheads are incurred

Closing Comments

- Many programming models out there
- Most are based on shared memory or message passing, and build on top of them
- Trade offs between control vs. complexity

- Overall, parallel programs still require a lot of tuning despite the programming model used

Module Review Questions

- What are two basic parallel programming models?

- What are key advantages shared memory model have over message passing model?

- What primitives are necessary for supporting shared memory parallel programming?

- In what way Transactional Memory simplify parallel programming?

Module Review Questions

- What are two basic parallel programming models?
  - Shared memory and message passing
- What are key advantages and disadvantages shared memory model have over message passing model?
  - Pluses: implicit communication, lower development effort, finer communication, Minuses: explicit synchronization, higher tuning effort, requires hardware support
- What primitives are necessary for supporting shared memory parallel programming?
  - Variable scope (shared vs. private), synchronization primitives
- In what way Transactional Memory simplify parallel programming?
  - Higher abstraction (simpler coding and reasoning), removing lock-related problems

Fundamentals of Parallel Computer Architecture - Chapter 2