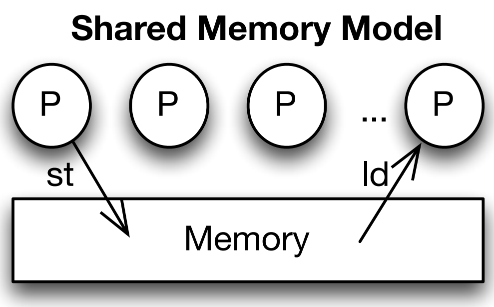
Chapter 2: Parallel Programming Models

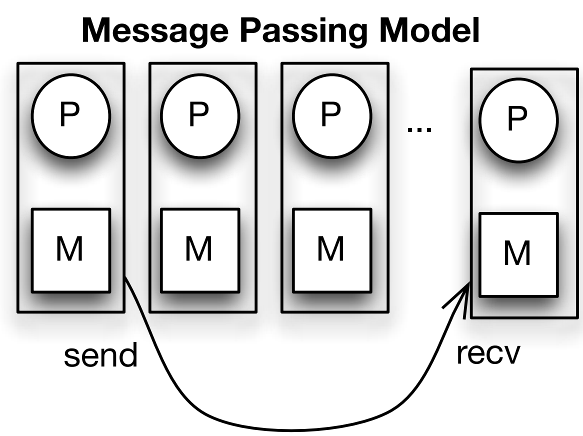
Module 2.1 Parallel Programming Models

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 / 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

Prog Model vs. Architecture

Was:

Shared Mem PM

Shared Memory Hardware

Msg Passing PM

Distributed Memory Hardware

Now:

Shared Mem PM | Msg Passing PM

Shared Cache/Memory Hardware

Shared Mem PM

SVM layer | Msg Passing PM

Distributed Memory Hardware

- Msg passing programs benefit from shared memory architecture
  - Sending a message achieved by passing a pointer to msg buffer
- Shared mem programs need software virtual memory (SVM) layer on distributed memory computers

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);
```