Message Passing Model

Message Passing Model

The parallel program consists of a collection of processes.
The model relies on two mechanisms:

- A method for creating separate processes on remote nodes.

Message Passing Model

The parallel program consists of a collection of processes.
The model relies on two mechanisms:

- A method for creating separate processes on remote nodes.
  - Single Program Multiple Data style. Single executable started *statically* at all processors. Control statements select different parts for each process to execute.

Message Passing Model

The parallel program consists of a collection of processes.
The model relies on two mechanisms:

- A method for creating separate processes on remote nodes.
  - Single Program Multiple Data style. Single executable started *statically* at all processors. Control statements select different parts for each process to execute.
  - Multiple Program Multiple Data style. Potentially separate programs for separate processors. Processes created from a main process: *dynamic* process creation.

Message Passing Model

The parallel program consists of a collection of processes.
The model relies on two mechanisms:

- A method for creating separate processes on remote nodes.
  - Single Program Multiple Data style. Single executable started *statically* at all processors. Control statements select different parts for each process to execute.
  - Multiple Program Multiple Data style. Potentially separate programs for separate processors. Processes created from a main process: *dynamic* process creation.
- The ability to send and receive messages.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

- send(&variable,. . . , P_{pid}): Send one or more primitive variables or arrays to the processor numbered *pid*. The ampersand represents the "address-of" operator (like in C or a reference in Java).

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

- send(&variable,. . . , P_{pid}): Send one or more primitive variables or arrays to the processor numbered *pid*. The ampersand represents the "address-of" operator (like in C or a reference in Java).
- send(&variable,. . . , P_{pid}, TAG): Send one or more primitive variables or arrays to the processor numbered *pid* in an message with tag TAG.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

- `send`(&variable,. . . , P_{pid}): Send one or more primitive variables or arrays to the processor numbered *pid*. The ampersand represents the "address-of" operator (like in C or a reference in Java).
- `send`(&variable,. . . , P_{pid}, TAG): Send one or more primitive variables or arrays to the processor numbered *pid* in an message with tag `TAG`.
- `recv`(&variable,. . . , P_{pid}). Receive a message from the specified process into the specified variable.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

- `send`(&variable,. . . , P_{pid}): Send one or more primitive variables or arrays to the processor numbered *pid*. The ampersand represents the "address-of" operator (like in C or a reference in Java).
- `send`(&variable,. . . , P_{pid}, TAG): Send one or more primitive variables or arrays to the processor numbered *pid* in an message with tag `TAG`.
- `recv`(&variable,. . . , P_{pid}). Receive a message from the specified process into the specified variable.
- `recv`(&variable,. . . , P_{pid}, TAG). Receive a message from the specified process with the specified tag into the specified variable.

Pseudo-code Convention

Synchronous send: waits until the complete message can be accepted by the receiving process before sending the message.
Synchronous recv: wait until the message arrives.

- send(&variable,..., P_{pid}): Send one or more primitive variables or arrays to the processor numbered *pid*. The ampersand represents the "address-of" operator (like in C or a reference in Java).
- send(&variable,..., P_{pid}, TAG): Send one or more primitive variables or arrays to the processor numbered *pid* in an message with tag TAG.
- recv(&variable,..., P_{pid}). Receive a message from the specified process into the specified variable.
- recv(&variable,..., P_{pid}, TAG). Receive a message from the specified process with the specified tag into the specified variable.
- **Wild cards**. Use P_{ANY} for any processors and *ANY_TAG* for any tag.

Pseudo-code Convention (contd.)

Asynchronous sends and recvs: These primitives do not wait for the actions to complete before returning. Usually requires buffering by library and/or local Operating Systems for messages. Or buffering could be done in-place using the variables (then we cannot modify the variables used until the message has transferred).

Pseudo-code Convention (contd.)

Asynchronous sends and recvs: These primitives do not wait for the actions to complete before returning. Usually requires buffering by library and/or local Operating Systems for messages. Or buffering could be done in-place using the variables (then we cannot modify the variables used until the message has transferred).

- async_send(&variable,..., P_{pid}, TAG, &request):
  Start an asynchronous send. The request is filled in by an unique identifier.

Asynchronous sends and recvs: These primitives do not wait for the actions to complete before returning. Usually requires buffering by library and/or local Operating Systems for messages. Or buffering could be done in-place using the variables (then we cannot modify the variables used until the message has transferred).

- async_send(&variable,..., P_{pid}, TAG, &request):
  Start an asynchronous send. The request is filled in by an unique identifier.
- async_recv(&variable,..., P_{pid}, TAG, &request):
  Attempt an asynchronous recv.

Pseudo-code Convention (contd.)

Asynchronous sends and recvs: These primitives do not wait for the actions to complete before returning. Usually requires buffering by library and/or local Operating Systems for messages. Or buffering could be done in-place using the variables (then we cannot modify the variables used until the message has transferred).

- async_send(&variable,..., P_{pid}, TAG, &request): Start an asynchronous send. The request is filled in by an unique identifier.
- async_recv(&variable,..., P_{pid}, TAG, &request): Attempt an asynchronous recv.
- async_wait(&request): Wait for asynchronous request to finish.

Pseudo-code Convention (contd.)

Asynchronous sends and recvs: These primitives do not wait for the actions to complete before returning. Usually requires buffering by library and/or local Operating Systems for messages. Or buffering could be done in-place using the variables (then we cannot modify the variables used until the message has transferred).

- async_send(&variable,. . . , P_{pid}, TAG, &request): Start an asynchronous send. The request is filled in by an unique identifier.
- async_recv(&variable,. . . , P_{pid}, TAG, &request): Attempt an asynchronous recv.
- async_wait(&request): Wait for asynchronous request to finish.
- async_test(&request): Test if asynchronous request has finished. Returns TRUE or FALSE.

Pseudo-code Convention (contd.)

Group operations

- bcast(&variable, . . . , P_{source}): Broadcast one or more primitive variables or arrays to all processes from the process P_{source}. All processes call the bcast method.

Pseudo-code Convention (contd.)

Group operations

- bcast(&variable,. . . , P_{source}): Broadcast one or more primitive variables or arrays to all processes from the process P_{source}. All processes call the bcast method.
- reduce(&data, &result, operation, P_{dest}): Reduce the value of the variable data across all processes to a single value using the specified operation. All processes call this method. The operation must be commutative.

Pseudo-code Convention (contd.)

Group operations

- bcast(&variable,..., P_{source}): Broadcast one or more primitive variables or arrays to all processes from the process P_{source}. All processes call the bcast method.
- reduce(&data, &result, operation, P_{dest}): Reduce the value of the variable data across all processes to a single value using the specified operation. All processes call this method. The operation must be commutative.
- scatter(&srcArray, &destVariable, P_{source}): Scatter the ith element of the source array on the source process P_{source} to the ith process. All processes call this method.

Pseudo-code Convention (contd.)

Group operations

- bcast(&variable,. . . , P_{source}): Broadcast one or more primitive variables or arrays to all processes from the process P_{source}. All processes call the bcast method.
- reduce(&data, &result, operation, P_{dest}): Reduce the value of the variable data across all processes to a single value using the specified operation. All processes call this method. The operation must be commutative.
- scatter(&srcArray, &destVariable, P_{source}): Scatter the ith element of the source array on the source process P_{source} to the ith process. All processes call this method.
- gather(&srcVariable, &destArray, P_{dest}): Gather the ith element of the destination array on the destination process P_{dest} from the source variable on the ith process. All processes call this method.

We will introduce more primitives later.

Parallel Sum Example-code

- There are p processes with process ids: $0 \leq pid \leq p-1$.
- Assume that the n elements are distributed across the p processes evenly such that each process has n/p elements.
- The sum is to be computed at process 0.

Parallel Sum Pseudo-code

```
parallel_sum(A, pid)
//p processes, process number pid is 0 ≤ pid ≤ p − 1
//Input: A[0 . . . n/p] on each process
//Output: sum on process 0
sum ← 0
for (i=0; i<n/p; i++)
do sum ← sum + A[i]

if (pid ≠ 0)
    send(&sum, &pid, P_0)
else
    partial_sums[0] ← sum
    for (i=1; i<p; i++)
    do    recv(&sum, &source, P_ANY)
          partial_sums[source] ← sum
    sum ← 0
    for (i=0; i<p; i++)
    do    sum ← sum + partial_sums[i]
    return sum
```

Evaluating Parallel Programs

We need to estimate the computation time as well as the communication overhead.

$$T_p(n) = t_{comp}(n) + t_{comm}(n)$$

Evaluating Parallel Programs

We need to estimate the computation time as well as the communication overhead.

$$T_p(n) = t_{comp}(n) + t_{comm}(n)$$

- Computational time: In general, this would be the longest computational time for the processes running the parallel program.

Evaluating Parallel Programs

We need to estimate the computation time as well as the communication overhead.

$$T_p(n) = t_{comp}(n) + t_{comm}(n)$$

- Computational time: In general, this would be the longest computational time for the processes running the parallel program.
- Communication time: To send n data words in one message, we will assume that the time taken is:

$$t_{startup} + n \times t_{data},$$

where $t_{startup}$ is time to send a message with no data and t_{data} is the transmission time per data word. Both these are assumed constants.

Parallel Sum Analysis with Communication Overhead

- Steps 1–3 are done by all processes and take $\Theta(n/p)$ computation time.

Parallel Sum Analysis with Communication Overhead

- Steps 1–3 are done by all processes and take $\Theta(n/p)$ computation time.
- Steps 4–10 involve $p-1$ processes sending partial sums to process 0. There are p separate messages with one data word each. Thus the communication time is:

$$\Theta(pt_{startup} + pt_{data})$$

Parallel Sum Analysis with Communication Overhead

- Steps 1–3 are done by all processes and take $\Theta(n/p)$ computation time.
- Steps 4–10 involve $p-1$ processes sending partial sums to process 0. There are p separate messages with one data word each. Thus the communication time is:

$$\Theta(pt_{startup} + pt_{data})$$

- Process 0 adds the partial sums up in Steps 11–13. This takes $\Theta(p)$ computation time.

Parallel Sum Analysis with Communication Overhead

- Steps 1–3 are done by all processes and take $\Theta(n/p)$ computation time.
- Steps 4–10 involve $p-1$ processes sending partial sums to process 0. There are p separate messages with one data word each. Thus the communication time is:

$$\Theta(pt_{startup} + pt_{data})$$

- Process 0 adds the partial sums up in Steps 11–13. This takes $\Theta(p)$ computation time.

Thus, the total time is:

$$\Theta(n/p + p + pt_{startup} + pt_{data})$$
$$= \Theta(n/p + p(1 + t_{startup} + t_{data}))$$
$$= \Theta(n/p + p)$$

Parallel Sum Analysis with Communication Overhead

- Steps 1–3 are done by all processes and take $\Theta(n/p)$ computation time.
- Steps 4–10 involve $p-1$ processes sending partial sums to process 0. There are p separate messages with one data word each. Thus the communication time is:

$$\Theta(pt_{startup} + pt_{data})$$

- Process 0 adds the partial sums up in Steps 11–13. This takes $\Theta(p)$ computation time.

Thus, the total time is:

$$\begin{aligned} &\Theta(n/p + p + pt_{startup} + pt_{data}) \\ &= \Theta(n/p + p(1 + t_{startup} + t_{data})) \\ &= \Theta(n/p + p) \end{aligned}$$

In this case, the startup time didn't make a significant difference but in some cases it does. Practically speaking, the startup time does cause overhead so sending fewer, larger messages will give faster times and better efficiency.

NetPipe 3.7.2 Benchmark Details

What does the startup overhead look like in real life?

- The tests were done on two nodes of the onyx cluster. Each node has one Gigabit Ethernet PCI Express network card and has a quad-core Intel 64-bit i5 3.1 GHz processor, 8 GB RAM and running 3.15.8-200.fc20.x86_64 Fedora Linux kernel. The version of the gcc compiler used was 4.8.3 20140624.
Test	startup time	data time
TCP	78 usec	0.001127 usec (887 MBits/sec)
- Note that we can send around 80,000 data words in one startup time!
- The commands that were used are shown below.

```
(on node01) NPtcp -h node02 -I -b 262144
(on node02)   NPtcp -I -b 262144
```

Exercises

1. Write pseudo-code for the unordered search problem in parallel. Use the following function prototype:

   parallel_search(A, pid)
   //p processes, process number pid is $0 \leq pid \leq p - 1$
   //Input: $A[0 \ldots n/p]$ on each process (unsorted)
   //Output: (pid,index) if found, otherwise -1

2. Write pseudo-code for two processes that play ping pong with a message!
3. Write pseudo-code for a a set of processes that pass a message around in a ring forever.