COMS3008A: Parallel Computing Lecture 3: Parallel Algorithm Design

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

March 5, 2025

Contents

WITS UNIVERSITY

Outline

Objectives

- Learn how to design a parallel program given a problem (often starting from a serial solution)

Outline

Outline

Shared memory vs distributed memory

- Physical memory in parallel computers is either shared or local.
- Shared memory parallel computer: parallel computers that share memory space
- Distributed memory parallel computer: parallel computers that do not share memory space

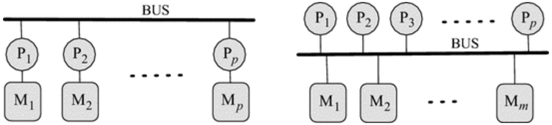


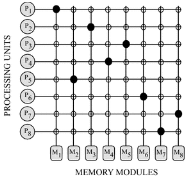
Figure: Bus based parallel computers.

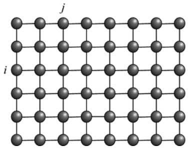
Shared memory vs distributed memory

- From programming point of view: shared memory programming and distributed memory programming
  - Shared memory programming: All PEs have equal access to shared memory space. Data shared among PEs are via shared variables.
  - Distributed memory programming: A PE has access only to its local memory. Data (message) shared among PEs are communicated via the communication channel, i.e., interconnection network.

Shared memory vs distributed memory cont.

- In terms of programming, programming shared memory parallel computers is easier than programming distributed memory systems.
- In terms of scalability, shared memory systems and distributed memory systems display different performance and cost characteristics. For example,
  - Bus, scalable in terms of cost, but not performance
  - Fully connected crossbar switch, scalable in terms of performance , but not cost
  - Hypercube, scalable both in performance and cost
  - 2D mesh, scalable both in performance and cost
- In terms of problem size, distributed memory systems are more suitable for problems with vast amount of data and computation





Outline

Shared memory

- In shared memory programming, variables have two types: shared variable and private variable.
- Shared variable can be read and write by any thread; and private variables can usually only be accessed by one thread.
- In shared memory systems, communication among threads happen via shared variables — this means communication is implicit.
- Non-determinism in shared memory programming.

Shared memory cont.

- Non-determinism: A computation is non-deterministic if a given input can result in different outputs.
- In a shared memory system, non-determinism arises when multiple threads are executing independently, and at different rates. Then these threads could complete at different orders for different runs, and hence could give different outputs from run to run. This is a typical case of non-determinism.
- Non-determinism can be harmless for some problem, and can cause issues for some problems.

Shared memory cont.

Example 1

Suppose we have two threads in a parallel program, one with thread ID 0, and the other with thread ID 1. Suppose also that each thread has a private variable `my_x`; thread 0 stores a value 5 for `my_x`, and thread 1 a value 9 for its `my_x`.

Thread ID	Variable	Value
0	`my_x`	5
1	`my_x`	9

Now, what will be the output like if both threads execute the following line of code?

```
...
printf("Thread %d > my_x = %d\n", my_ID, my_x);
...
```

Shared memory cont.

Example 1 cont.

The output could be

```
Thread 0 > my_x = 5
Thread 1 > my_x = 9
```

or

```
Thread 1 > my_x = 9
Thread 0 > my_x = 5
```

Shared memory cont.

Example 2

Suppose we have a shared variable `min_x` (with initial value ∞) and two threads in our program. Each thread has a private variable `my_x`, thread 0 stores a value 5 for `my_x`, and thread 1 a value 9 for its `my_x`. What will happen if both threads update `min_x = my_x` simultaneously?

```
/*each thread tries to update variable min_val as
    follows*/
if (my_x < min_x)
  min_x = my_x;
```

Shared memory cont.

Example 2 cont.

- This is the situation where two threads more or less try to update a shared variable (`min_x`) simultaneously.
- **Race condition:** When threads attempt to access a resource simultaneously, and the access can result in error, we often say the program has a race condition.
- In such case, we can serialize the contending activities by setting a critical section where the section can only be accessed by one thread at a time.

Outline

Distributed memory

- In distributed memory programming, the processes can only access their own private (or local) memory. Message-passing programming model (or APIs) is most commonly used. For example, MPI.
- In message passing, each process is identified by its rank.
- Processes communicate with each other by explicitly sending and receiving messages.
- Message passing APIs often provide basic send and receive functions, they also provide more powerful communication functions such as broadcast and reduction.
  - Broadcast, a single process transmits the same data to all processes;
  - Reduction, the results computed by individual processes are combined to a single results, e.g., summation.

Outline

Parallel program design

The following steps are often taken to solve a problem in parallel [1]

- Decomposition (or partitioning) of the computation into tasks;
- Assignment of tasks to processes;
- Orchestration of the necessary data access, communication, synchronization among processes;
- Mapping of processes to processors.

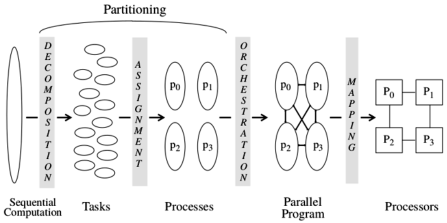


Figure: Steps in parallelization, relationship between PEs, tasks and processors.

[1] https://dl.acm.org/doi/pdf/10.5555/2821564, in Chapter 2

Parallel program design cont.

- Decomposition involves decomposing a problem into finer tasks such that these tasks could be executed in parallel.
- The major goal of decomposition is to expose enough concurrency to keep processes busy all the time, yet not so much that overhead of managing tasks become substantial.
- Assignment involves assigning fine tasks to available processes, such that each process has approximately similar workload. Load balancing is an challenging issue in parallel programming.
- The primary performance goals of assignment are to achieve balanced workload, reduce the runtime overhead of managing assignment.

Parallel program design cont.

- The third step involves necessary orchestration, could involve communication among processes and synchronization.
- The major performance goals in orchestration:
  - reducing the cost of communication and synchronization
  - preserving locality of data reference
  - scheduling tasks
  - reducing the overhead of parallelism management

Parallel program design cont.

- Finally, the mapping is to map processes to processors. The number of processes and the number of processors are not necessarily need to be matched. That is, for example, you can have 8 processes on a 4 processor machines. In such a case, a processor may handle more than one processes by techniques such as space sharing and time sharing.
- The mapping process can be taken care of by OS in order to optimize resource allocation and utilization. The program may also control the mapping of course.

Parallel program design — simple example

Example 3

Consider a simple example program with two phases.

- In the first phase, a single operation is performed independently on all points of a 2-dimensional n by n grid;
- in the second phase, the sum of n^2 grid point values is computed.

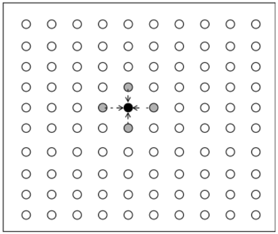


Figure: 2D grid points.

Parallel program design — simple example cont.

Example 3 cont.

If we have p processes,

- we can assign n^2/p points to each process and complete the first phase in time n^2/p.
- In the second phase, each process can add each of its assigned n^2/p values to a global sum variable.

Parallel program design — simple example cont.

Example 3 cont.

- Issue: the second phase is in serial where it takes n^2 time regardless of multiple processes. The total time is $n^2/p + n^2$. Then the speedup, compared to the sequential time $2n^2$, is

$$s = \frac{2n^2}{\frac{n^2}{p} + n^2} = \frac{2}{\frac{1}{p} + 1},$$

  which is at most 2, even if a large number of processes is used.
- Can we expose more concurrency?

WITS UNIVERSITY

Parallel program design — simple example cont.

Example 3 cont.

We can improve the performance:

- We can first compute local sums of n^2/p values on all processes simultaneously.
- After n^2/p time, we have p number of local sums.
- We then add these p local sums into a global sum one at a time, which takes p units of time.
- Now the total time is $n^2/p + n^2/p + p$. The speedup is

$$s = \frac{2n^2}{\frac{2n^2}{p} + p} = p \times \frac{2n^2}{2n^2 + p^2}.$$

This speedup is almost linear in p, the number of processors used, when n is large compared to p.

Parallel program design — simple example cont.

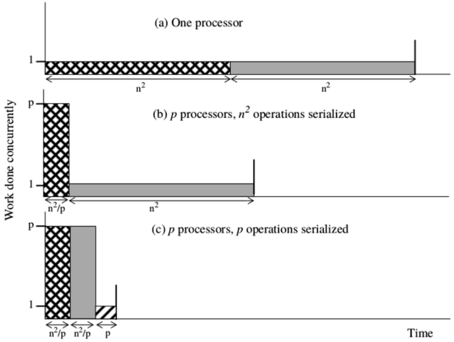


Figure: Impact of limited concurrency.

Parallel program design — simple example cont.

Example 4

Suppose we have an array with large quantities of floating point data stored in it. In order to have a good feeling of the distribution of the data, we can find the histogram of the data. To find the histogram of a set of data, we can simply divide the range of data into equal sized subintervals, or bins, determine the number of values in each bin, and plot a bar graphs showing the sizes of the bins.

Parallel program design — simple example cont.

Example 4 cont.

As a very small example, suppose our data are

$$A = [1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, \\ 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9]$$

- For A, $A_{min} = 0.3, A_{max} = 4.9, A_{count} = 20$.
- Let's set the number of bins to be 5, and the bins are the $[0, 1.0), [1.0, 2.0), [2.0, 3.0), [3.0, 4.0), [4.0, 5.0)$, $bin_{width} = (5.0 - 0)/5 = 1.0$.
- Then $bin_{count} = [6, 3, 2, 3, 6]$ is the histogram — the output is an array the number of elements of data that lie in each bin.

Parallel program design — simple example cont.

Example 4 cont.

```
for (i = 0; i < data_count; i++){
    bin = Find_bin(data[i], ...);
    bin_count[bin]++;
}
```

The `Find_bin` function returns the bin that `data[i]` belongs to.

WITS UNIVERSITY

Parallel program design — simple example cont.

Example 4 cont.

- Now if we want to parallelize this problem, first we can decompose the dataset into subsets. Given the small dataset, we divide it into 4 subsets, so that each subset has 5 elements.
  - Identify tasks (decompose): i) finding the bin to which an element of data belongs; ii) increment the corresponding entry in `bin_count`.
  - The second task can only be done once the first task has been completed.
  - If two processes or threads are assigned elements from the same bin, then both of them will try to update the count of the same bin. Assuming the `bin_count` is shared, this will cause **race condition**.

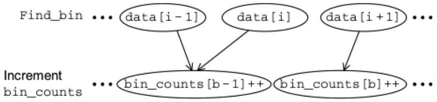


Figure: Tasks and their communications.

Parallel program design — simple example cont.

Example 4 cont.

- A solution to race condition in this example is to create local copies of `bin-count`, each process updates their local copies, and at the end add these local copies into the global `bin_count`.

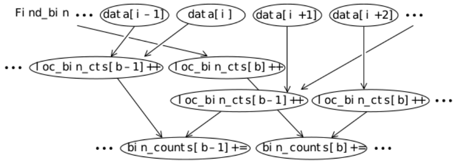


Figure: Tasks and communications.

Parallel program design — simple example cont.

Example 4 cont.

- In summary, the parallelization approach is to
  - Elements of data are assigned to processes/threads so that each process/thread gets roughly the same number of elements;
  - Each process/thread is responsible for updating its `local_bin_counts` array based on the assigned data elements.
  - The local `local_bin_counts` are needed to be aggregated into the global `bin_counts`.
    - If the number of bins is small, the final aggregation can be done by a single process/thread.
    - If the number of bins is large, we can apply parallel addition in this step too.

Summary

- Aspects of parallel program design
  - Decomposition to create concurrent tasks
  - Assignment of works to workers
  - Orchestration to coordinate processing of tasks by processes/threads
  - Mapping to hardware

  We will look more into making good decisions in these aspects in the coming lectures.