Pipelined Computations
Pipeline Techniques

- A technique which is applied to a wide range of problems that are partially sequential in a nature.

- A sequence of steps must be undertaken, hence pipelining can be used to parallelize sequential code.
Pipeline Techniques

- The problem is divided into a series of tasks.
- These tasks will be executed by a separate process or processor.
- Each process is a pipeline stage.
- The tasks must be performed in succession.

Figure 5.1 Pipelined processes
Pipeline Techniques

• A sequential program can be formulated as a pipeline.
• Consider a program that models building a car
Pipeline Techniques

• One pipeline solution can be a separate stage for each statement.
  – construct the frame
  – build the engine and transmission
  – install the engine and transmission
  – install fenders
  – install roof
  – install hood
  – install doors
  – install wheels and tires

• Instead of simple statements, a series of functions could be performed in a pipeline fashion.
Pipeline Techniques

• The pipelined approach can provide increased speed under the following types of computations:
  
  – If more than one instance of the complete problem is to be executed.
  
  – If a series of data items must be processed, each requiring multiple operations.
  
  – If information to start the next process can be passed forward before the process has completed all its internal operations.
Pipeline Techniques

- **Type 1**
  - Happens in simulation runs
  - Space-time diagram in figure 5.2

---

**Figure 5.2** Space-time diagram of a pipeline

Lecture 5 – Pipelined Computations
Pipeline Techniques

• Type 1
  – Assuming $p$ processes and $m$ instances of problem.
  – The total pipeline cycles to execute all $m$ instances is $m+p-1$.
  – The average number of cycles for one instance is $(m+p-1)/m$.
  – For large $m$, this is 1 cycle per instance
  – Very important formula: $m+p-1$
    • this is a general result for all pipelined solutions.
Pipeline Techniques

• Type 2
  – A series of data items must be processed in a sequence.
  – The arrangement can be shown as in figure 5.4.
  – The overall execution time is still $p+m-1$ cycles; assuming they are all equal.
  – The space time diagram is the same as for Type 1

Input Sequence:
$$d_7d_6d_5d_4d_3d_2d_1 \quad \rightarrow \quad P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6 \rightarrow P_7$$

*Figure 5.4 Pipeline processing 10 data elements*
Pipeline Techniques

• Type 3
  – Only one instance of the problem to execute.
  – Each process can pass on information to next process before it has completed.
  – Figure 5.5 shows space-time diagrams for two cases in this type.
Pipeline Techniques

• Type 3

Information transfer sufficient to start next process

(a) Processes with the same execution time

(b) Processes not with the same execution time

Figure 5.5 Pipeline processing where information passes to next stage before end of process
Computing Platform for Pipelined Applications

- Key requirement: ability to send message between adjacent processes.
  - A line or ring will work.
  - It can be embedded on meshes and hypercubes.
  - Networked workstations (simple Ethernet) may not be suitable in this case unless a more complex interconnection structure is used.
  - Here, we will assume the interconnection structure can provide simultaneous transfers between adjacent processes or processors.
Pipeline Program Examples

• Adding Numbers
  – Problem: adding a list of numbers (or performing any associative operation on a sequence of numbers).
  – A pipeline solution has each process in the pipeline add one number to an accumulating sum.
  – One number is held in each process and the partial sum is passed from one process to the next, each process adding its number to the accumulating sum.
Pipeline Program Examples

• Adding Numbers
  – Basic code
    ```
    recv(&accumulation, P_{i-1});
    accumulation += number;
    send(&accumulation, P_{i+1});
    ```
Pipeline Program Examples

- Adding Numbers
  - After considering the first and last process, the program looks like:

```c
if (process > 0) {
    recv(&accumulation, Pi-1);
    accumulation += number;
}
if (process < n-1) {
    send(&accumulation, Pi+1);
}
```
Pipeline Program Examples

- Adding Numbers
  - Two available network structures
    - Ring architecture
      - All data entered into the first process
    - Master – slave organization
      - Data enters a process when it is needed directly from the master.
  - Data partitioning can be useful
    - 1000 numbers, 1000 slave processes?
Pipeline Program Examples

• Adding Numbers
  – Analysis
    • Type 1, only efficient if we have more than one instance of the problem to solve.
    • Assume all processes have simultaneous communication and computation phases.

\[
t_{\text{total}} = (\text{time for one pipeline cycle})(\text{number of cycles})
\]

\[
t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}})(m+p-1)
\]

\[
t_{\text{average}} = \frac{t_{\text{total}}}{m}
\]

\[
t_{\text{comp}} = 1
\]

\[
t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})
\]
Pipeline Program Examples

- Adding Numbers
- Analysis
  - Single Instance (m=1)
    \[ t_{\text{total}} = t_{\text{average}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)p \]
  - Multiple Instance
    \[ t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + 1)(m+p-1) \]
    For a large m (m>>p);
    \[ t_{\text{average}} = \frac{t_{\text{total}}}{m} = 2(t_{\text{startup}} + t_{\text{data}}) + 1 \]
    That is, one pipeline cycle
Pipeline Program Examples

- Adding Numbers
  - Analysis
    - Data Partitioning with Multiple Instances
      - Each stage processing a group of \( d \) numbers.
      - Number of processes is \( p = n/d \)
      - Each computation will be \( d \)

\[
t_{\text{total}} = (2(t_{\text{startup}} + t_{\text{data}}) + d)(m+n/d-1)
\]
Pipeline Program Examples

- **Sorting Numbers**
  - Problem: Reorder a set of numbers in nonincreasing (or nondecreasing) order.
  - Algorithm: process 1 chose and store the largest number for all numbers in the set; process 2 chose and store the second largest number from the rest of the set; and so on.
  - This algorithm is a parallel version of *insertion sort*.
Pipeline Program Examples

• Sorting Numbers
  – The parallel program will look like:

```c
right_pro = n-i-1;
recv(&x, P_{i-1});
for (j=0; j<right_pro; j++) {
    recv(&number, P_{i-1});
    if(number>x) {
        send(&x, P_{i+1});
        x = number;
    } else {
        send(&number, P_{i+1});
    }
}
```
Pipeline Program Examples

- Sorting Numbers
  - This is actually a type 2 example
  - The picture

*Figure 5.7 Pipeline for sorting using insertion sort*
Pipeline Program Examples

- Sorting Numbers

Analysis
- A sequential implementation requires
  \[ t_s = (n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 \]
  Approximate number of \( n^2/2 \).
- Parallel implementation has \( n+n-1 = 2n-1 \) cycles, if there are \( n \) pipeline processes and \( n \) numbers to sort (\( m = p = n \))
  \[ t_p = (t_{\text{comp}} + t_{\text{comm}})(2n-1) \]
  \[ t_{\text{comp}} = 1 \]
  \[ t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}}) \]
- In the parallel version the time cost is linear in \( n \)
Summary

• The pipeline concept and its application areas
• Three cases where the usage of pipeline can improve the performance
• Analysis of pipelines
• Examples showing the potential of pipelining
  – Adding numbers.
  – Insertion Sort.