Programming Paradigms

- Imperative Programming – Fortran, C, Pascal
- Functional Programming – Lisp
- Object Oriented Programming – Simula, C++, Smalltalk
- Logic Programming - Prolog

1

Parallel Programming

A misconception occurs that parallel programs are difficult to write as compared to sequential programmes. Consider the situations:

- ✓ Write a sequential programme and its multiple copies on a parallel computer. Parallelism remains transparent to the user.
- ✓ Write an Oracle application. Oracle is implicitly parallel. Run it on a parallel machine. Similarly parallel C and FORTRAN90.

2

Parallel Programming

A parallel computer should be flexible and easy to use. This will depend upon its architecture and the way we write a parallel program on it.

Let us consider various Parallel Programming paradigms:

3

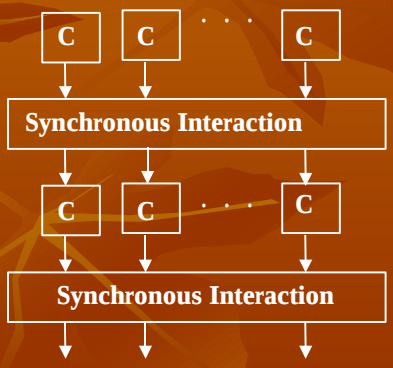
Parallel Programming Paradigm

- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool
- Remark :

The parallel program consists of number of super steps, and each super step has two phases : *computation phase and interaction phase*

4

Phase Parallel Model



- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each has two phases.
- In a computation phase, multiple processes each perform an independent computation *C*.
- In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
- Then next superstep is executed.

5

Phase Parallel Model

This paradigm is also known as Loosely Synchronous Paradigm or the Agenda Paradigm.

6

Phase Parallel Model Shortcomings

- Interaction is not overlapped with computation
- It is difficult to maintain balanced workload amongst processors.

7

Phase Parallel Model

A special case of Phase-Parallel Paradigm is Synchronous Iteration Paradigm where the supersteps are a sequence of iterations in a loop.

Consider the example of computing x=f(x) where x is an n-dimensional vector.

8

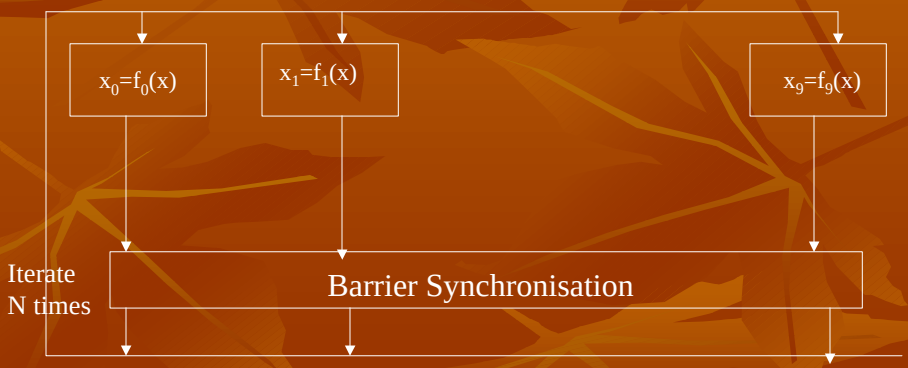
Synchronous Iteration Paradigm

```
parfor (i=0; i<n; i++)   //create n processes
                         //each executing a for loop
{
  for (j=0; j<N; j++)
    {
      x[i] = f_i(x);
      barrier;
    }
  }
```

9

Synchronous Iteration Paradigm

For n = 9 we have



10

Asynchronous Iteration Paradigm

```
parfor (i=0; i<n; i++)
{
  for (j=0; j<N; j++)
    x[i] = f_i(x);
  }
```

It allows a process to proceed to the next iteration, without waiting for the remaining processes to catch up.

11

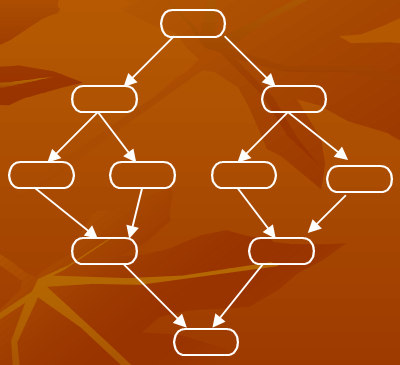
Asynchronous Iteration Paradigm

The above code could be indeterminate, because when a process is computing x[i] in the jth iteration, the x[i-1] value used could be computed by another process in iteration j-1.

However, under certain conditions, an asynchronous iteration algorithm will converge to the correct results and is faster than the synchronous iteration algorithm.

12

Divide and Conquer



- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- The child processes then compute their workload in parallel and the results are merged by the parent.
- The dividing and the merging procedures are done recursively.
- This paradigm is very natural for computations such as quick sort. Its disadvantage is the difficulty in achieving good load balance.

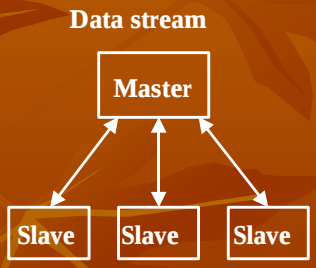
13

Pipeline

- In pipeline paradigm, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.

14

Process Farm



- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.

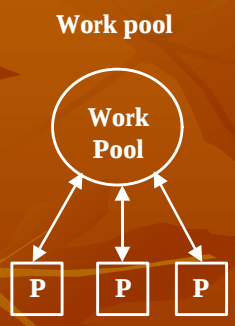
15

Process Farm Disadvantage

The master can become a bottleneck.

16

Work Pool



- This paradigm is often used in a shared variable model.
- A pool of works is realized in a global data structure.
- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool.
- The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.

17

Work Pool

Implementing the work pool to allow efficient accesses by multiple processes is not easy especially in a message passing model. The work pool may be an unordered set, a queue or a priority queue.

18

Programmability Issues

We define programmability as a combination of

- ✓ Structuredness
- ✓ Generality
- ✓ Portability

19

Parallel Programming Models

Implicit parallelism

- If the programmer does not explicitly specify parallelism, but let the compiler and the run-time support system automatically exploit it.

Explicit Parallelism

- It means that parallelism is explicitly specified in the source code by the programming using special language constructs, complex directives, or library cells.

20

Implicit Parallel Programming Models

Implicit Parallelism: Parallelizing Compilers

- Automatic parallelization of sequential programs
  - Dependency Analysis
  - Data dependency
  - Control dependency

Remark

Users belief is influenced by the currently disappointing performance of automatic tools (Implicit parallelism) and partly by a theoretical results obtained

21

Implicit Parallel Programming Models

Effectiveness of Parallelizing Compilers

- **Question :**
  - **Are parallelizing compilers effective in generalizing efficient code from sequential programs?**
    - **Some performance studies indicate that may not be a effective**
    - **User direction and Run-Time Parallelization techniques are needed**

22

Implicit Parallel Programming Models

Implicit Parallelism

- Bernstein's Theorem
  - It is difficult to decide whether two operations in an imperative sequential program can be executed in parallel
  - An implication of this theorem is that there is no automatic technique, compiler time or runtime that can exploit all parallelism in a sequential program

23

Implicit Parallel Programming Models

To overcome this theoretical limitation, two solutions have been suggested

- The first solution is to abolish the imperative style altogether, and to use a programming language which makes parallelism recognition easier
- The second solution is to use explicit parallelism

24

Explicit Parallel Programming Models

Three dominant parallel programming models are :

- Data-parallel model
- Message-passing model
- Shared-variable model

25

Explicit Parallel Programming Models

Main Features	Data-Parallel	Message-Passing	Shared-Variable
Control flow (threading)	Single	Multiple	Multiple
Synchrony	Loosely synchronous	Asynchronous	Asynchronous
Address space	Single	Multiple	Multiple
Interaction	Implicit	Explicit	Explicit
Data allocation	Implicit or semiexplicit	Explicit	Implicit or semiexplicit

26

Explicit Parallel Programming Models

Message – Passing

- Message passing has the following characteristics :
  - Multithreading
  - Asynchronous parallelism (MPI reduce)
  - Separate address spaces (Interaction by MPI/PVM)
  - Explicit interaction
  - Explicit allocation by user

27

Explicit Parallel Programming Models

Message – Passing

- **Programs are multithreading and asynchronous requiring explicit synchronization**
- **More flexible than the data parallel model, but it still lacks support for the work pool paradigm.**
- **PVM and MPI can be used**
- **Message passing programs exploit large-grain parallelism**

28

Explicit Parallel Programming Models

Shared Variable Model

- It has a single address space (Similar to data parallel)
- It is multithreading and asynchronous (Similar to message-passing model)
- Data resides in single shared address space, thus does not have to be explicitly allocated
- Workload can be either explicitly or implicitly allocated
- Communication is done implicitly through shared reads and writes of variables. However synchronization is explicit

29

Explicit Parallel Programming Models

Shared variable model

- The shared-variable model assumes the existence of a single, shared address space where all shared data reside
- Programs are multithreading and asynchronous, requiring explicit synchronizations
- Efficient parallel programs that are loosely synchronous and have regular communication patterns, the shared variable approach is not easier than the message passing model

30

Other Parallel Programming Models

- Functional programming
- Logic programming
- Computing by learning
- Object oriented programming

31