Parallel Programming Models, Languages and Compilers

Points to be covered

- **Parallel Programming Models**-

  Shared-Variable Model, Message-Passing Model, Data-Parallel Model, Object Oriented Model, Functional and Logic Models
- **Parallel Languages and Role of Compilers-**

  Language Features for Parallelism, Parallel Language Constructs, Optimizing Compilers for Parallelism
- **Code Optimization and Scheduling-**

  Scalar Optimization with Basic Blocks, Local and Global Optimizations, Vectorization and Parallelization Methods, Code Generation and Scheduling, Trace Scheduling Compilation

10.1 Parallel Programming Model

- Programming model->simplified and transparent view of computer hardware/software system.
- Parallel Programming Model are specifically designed for multiprocessors, multicomputer or vector/SIMD computers.

Classification

- We have 5 programming models-:

✓ Shared-Variable Model

✓ Message-Passing Model

✓ Data-Parallel Model

✓ Object Oriented Model

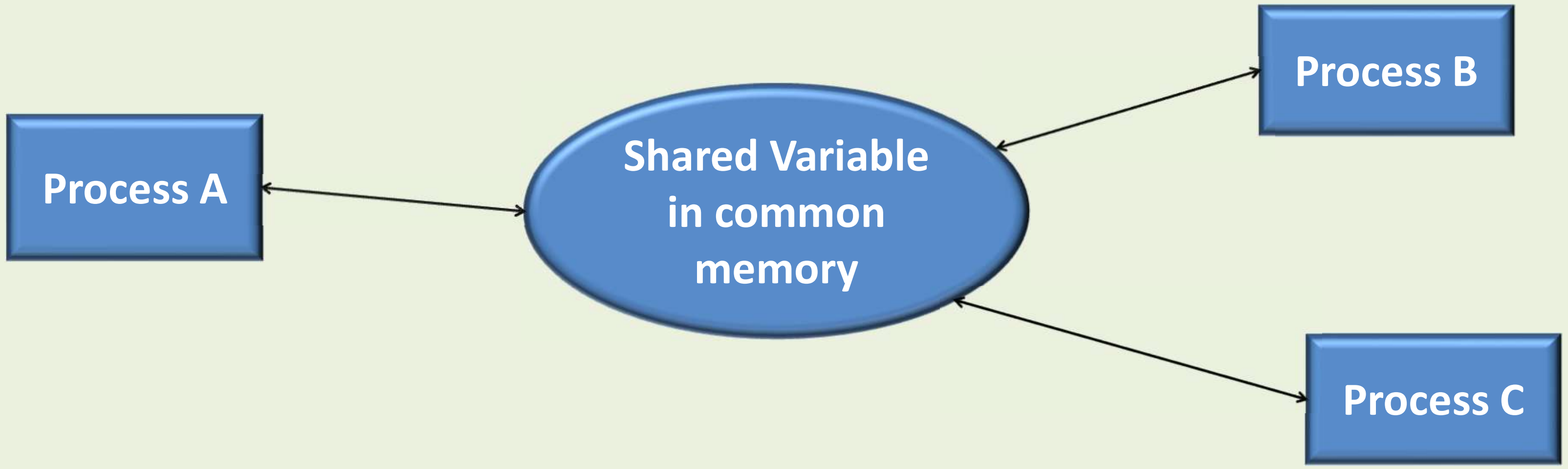
✓ Functional and Logic Models

Shared Variable Model

- In all programming system, processors are **active resources** and memory & IO devices are **passive resources**.
- Program is a collection of processes.
- Parallelism depends on how IPC(Interprocess Communication) is implemented.
- Process address space is shared.
- To ensure orderly IPC ,a mutual exclusion property requires that shared object must be shared by only 1 process at a time.

Shared Variable communication

- Used in multiprocessor programming
- Shared variable IPC demands use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.



Critical Section

- Critical Section(CS) is a code segment accessing shared variable, which must be executed by only one process at a time and which once started must be completed without interruption.

Critical Section Requirements

- It should satisfy following requirements-:

✓ Mutual Exclusion

At most one process executing CS at a time.

✓ No deadlock in waiting

No circular wait by 2 or more process.

✓ No preemption

No interrupt until completion.

✓ Eventual Entry

Once entered CS,must be out after completion.

Protected Access

- Granularity of CS affects the performance.
- If CS is too large,it may limit parallism due to excessive waiting by process.
- When CS is too small,it may add unnecessary code complexity/Software overhead.

4 operational Modes

- Multiprogramming
- Multiprocessing
- Multitasking
- Multithreading

Multiprogramming

- Multiple independent programs running on single processor/multiprocessor by time sharing use of system resource.
- When program enters the I/O mode, the processor switches to another program.

Multiprocessing

- When multiprogramming is implemented at the process level on a multiprocessor, it is called multiprocessing.
- 2 types of multiprocessing-:
- ✓ If interprocessor communication are handled at the instruction level, the multiprocessor operates in MIMD mode.
- ✓ If interprocessor communication are handled at the program,subroutine or procedure level, the multiprocessor operates in MPMD mode.

Multitasking

- A single program can be partitioned into multiple interrelated tasks concurrently executed on a multiprocessor.
- Thus multitasking provides the parallel execution of 2 or more parts of single program.

Multithreading

- The traditional UNIX/OS has a single threaded kernal in which 1 process can receive OS kernal service at a time.
- In multiprocessor we extend single kernal to be multithreaded.
- The purpose is to allow multiple threads of light weight processes to share same address space.

Partitioning and Replication

- Goal of parallel processing is to exploit parallelism as much as possible with lowest overhead.
- Program partitioning is a technique for decomposing a large program and data set into many small pieces for parallel execution by multiple processors.
- Program partitioning involves both programmers and compilers.

Partitioning and Replication

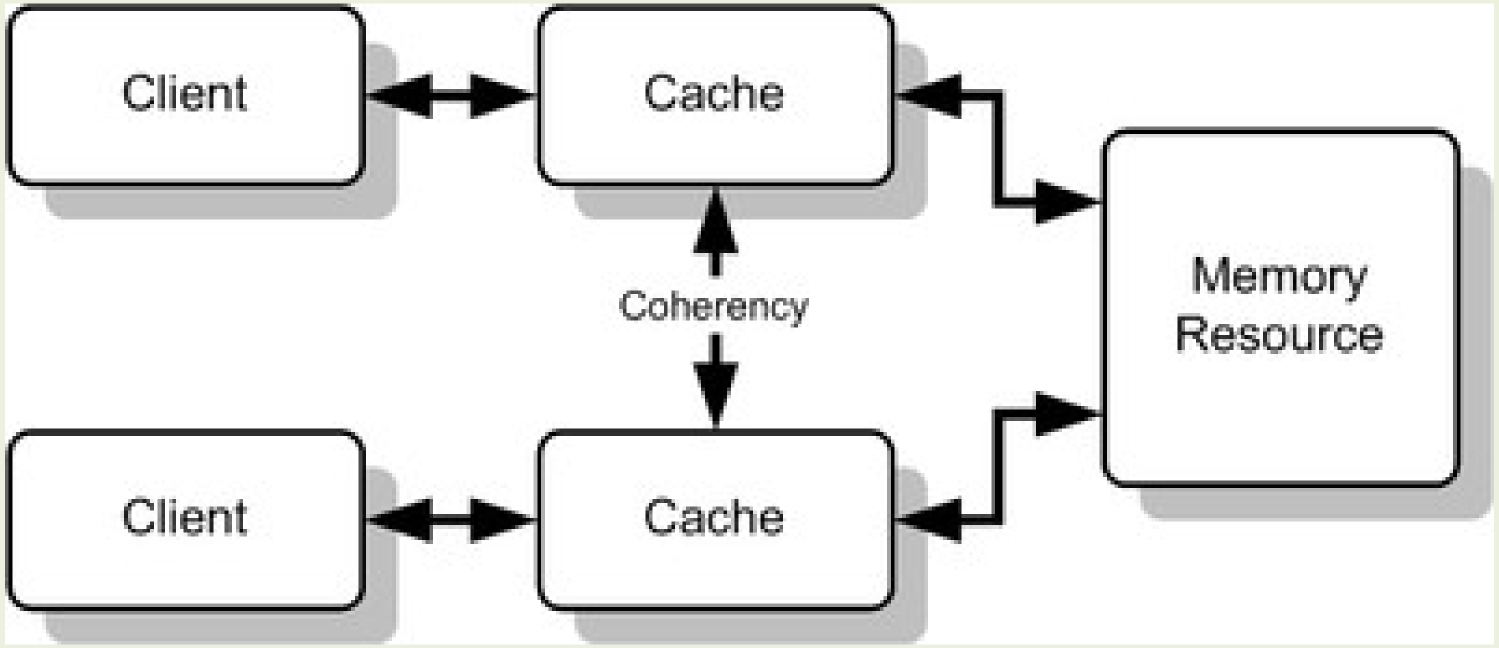
- Program replication refers to duplication of same program code for parallel execution on multiple processors over different data sets.

Scheduling and Synchronization

- Scheduling further classified-:
  - ✓ Static Scheduling
  - It is conducted at post compile time.
  - Its advantage is low overhead but shortcomings is a possible mismatch with run time profile of each task.
  - ✓ Dynamic Scheduling
  - Catches the run time conditions.
  - Requires fast context switching,premption and much more OS support.
  - Advantage include better resource utilization at expense of highest scheduling overhead.

Cache Coherence & Protection

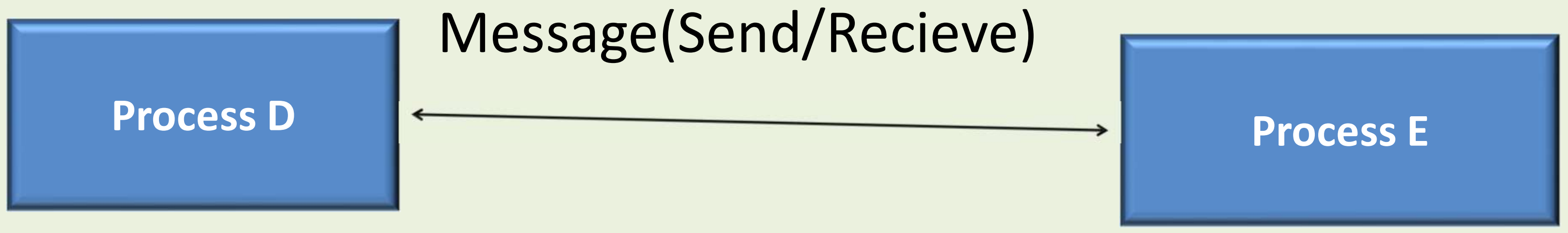
- Multicache coherance problem demands an invalidation or update after each write operation.



Message Passing Model

- Two processes D and E residing at different processor nodes may communicate with each other by passing messages through a direct network.
- The messages may be instructions, data, synchronization or interrupt signals etc.
- Multicomputers are considered loosely coupled multiprocessors.

IPC using Message Passing



Synchronous Message Passing

- No shared Memory
- No mutual Exclusion
- Synchronization of sender and reciever process just like telephone call.
- No buffer used.
- If one process is ready to cummunicate and other is not, the one that is ready must be blocked.

Asynchronous Message Passing

- Does not require that message sending and receiving be synchronised in time and space.
- Arbitrary communication delay may be experienced because sender may not know if and when the message has been received until acknowledgement is received from receiver.
- This scheme is like a postal service using mailbox with no synchronization between senders and recievers.

Data Parallel Model

- Used in SIMD computers
- Parallelism handled by hardware synchronization and flow control.
- Fortran 90 ->data parallel lang.
- Require pre-distributed data sets.

Data Parallelism

- This technique used in array processors(SIMD)
- Issue->match problem size with machine size.

Array Language Extensions

- Various data parallel language used
- Represented by high level data types
- CFD for Illiac 4,DAP fortran for Distributed array processor, C* for Connection machine
- Target to make the number of PE’s of problem size.

Object Oriented Model

- Objects dynamically created and manipulated.
- Processing is performed by sending and receiving messages among objects.

Concurrent OOP

- Need of OOP because of abstraction and reusability concept.
- Objects are program entities which encapsulate data and operations in single unit.
- Concurrent manipulation of objects in OOP.

Actor Model

- This is a framework for Concurrent OOP.
- Actors->independent component
- Communicate via asynchronous message passing.
- 3 primitives->create, send to and become.

Parallelism in COOP

- 3 common patterns for parallelism-:

1)Pipeline concurrency

overlapped enumeration of successive solutions and concurrent testing of solutions

2)Divide and conquer

concurrent elaboration of different subprograms and combining of their solutions to produce a overall problem solution

3)Cooperative Problem Solving

aims at mutually-supported agreements, improved relationships, and continued **problem**-**solving** capacity among the parties.

Functional and logic Model

- Functional Programming Language->

Lisp,Sisal and Strand 88.

Logic Programming Language->

Concurrent Prolog and Parlog

Functional Programming Model

- Should not produce any side effects.
- No concept of storage, assignment and branching.
- Single assignment and data flow language functional in nature.

Logic Programming Models

- Used for knowledge processing from large database.
- Supports implicitly search strategy.
- AND-parallel execution and OR-Parallel Reduction technique used.
- Used in artificial intelligence

10.2 Parallel Language and Compilers

- Programming environment is collection of s/w tools and system support.
  - Parallel Software Programming environment needed.
- Users still forced to focus on hardware details rather than parallelism using high level abstraction.

10.2.1 Language Features For Parallelism

- Optimization Features
- Availability Features
- Synchronization/communication Features
- Control Of Parallelism
- Data Parallelism Features
- Process Management Features

Optimization Features

- Theme->Conversion of sequential Program to Parallel Program.
- The purpose is to match s/w parallelism with hardware parallelism.

- Software in Practice-:

1)Automated Parallelizer

Express C automated parallelizer and Allaint FX Fortran compiler.

2)Semiautomated Parallizer

Needs compiler directives or programmers interaction.

3) Interactive restructure support

static analyzer, run-time statistics and code translator for restructuring.

Availability Features

- Theme-:Enhance user friendliness, make language portable for large no of parallel computers and expand the applicability of software libraries.

1)Scalability

Language should be scalable to number of processors and independent of hardware topology.

2)Compatibility

Compatible with sequential language.

3)Portability

Language should be portable to shared memory multiprocessor, message passing or both.

Synchronization/Communication Features

- Shared Variable (locks) for IPC.
- Single assignment language.
- Send/receive for message passing.
- Logical shared memory such as the row space in Linda.
- Remote procedure call.
- Data flow languages such as id.
- Mailbox, Semaphores, Monitors

Control Of Parallelism

- Coarse, Medium and fine grain
- Explicit vs implicit parallelism
- Global Parallelism
- Loop Parallelism
- Task Parallelism
- Divide and Conquer Parallelism

Data Parallelism Features

Theme-:how data is accessed and distributed in either SIMD and MIMD computers.

1)Runtime automatic decomposition

Data automatically distributed with no user interaction.

2)Mapping Specification

User specifies patterns and input data mapped to hardware.

3) Virtual Processor Support

Compilers made statically and maps to physical processor.

4) Direct Access to shared data

Shared data is directly accessed by operating system.

Process Management Features

Theme-:

Support efficient creation of parallel process, implementation of multithreading or multitasking, program partitioning and replication and dynamic load balancing at run time.

1)Dynamic Process Creation at Run Time.

2)Creation of lightweight processes.

3)Replication technique.

4)Partitioned Networks.

5)Automatic Load Balancing

10.2.3 Optimizing Compilers for Parallelism

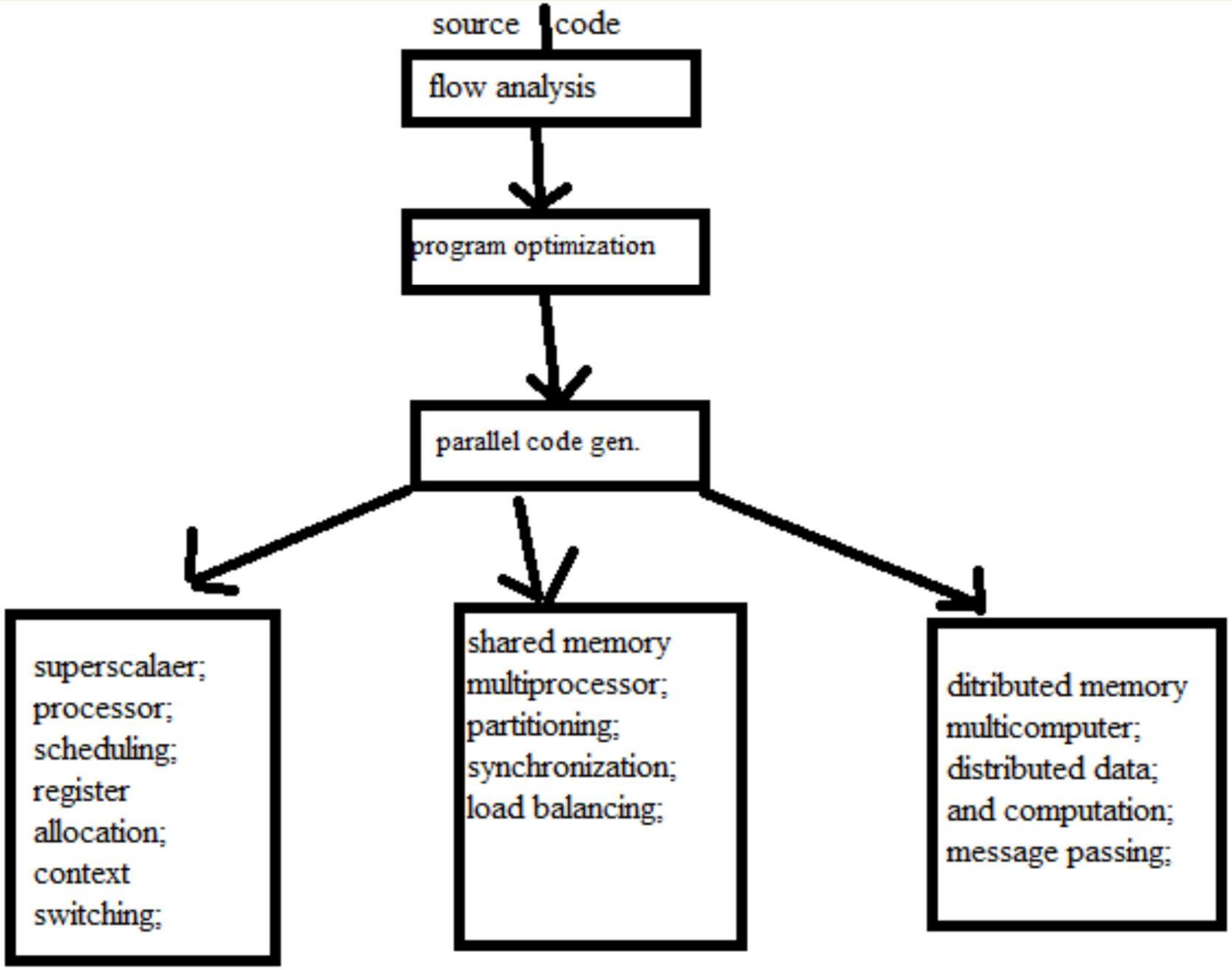
- Role of compiler to remove burden of optimization and generation.

3 Phases-:

1)Flow analysis

2)Optimization

3)Code Generation


source code
flow analysis
program optimization
parallel code gen.
superscalaer; processor; scheduling; register allocation; context switching;
shared memory multiprocessor; partitioning; synchronization; load balancing;
ditributed memory multicomputer; distributed data; and computation; message passing;

Flow Analysis

- Reveals design flow patters to determine data and control dependencies.
- Flow analysis carried at various execution levels.

1)Instruction level->VLSI or superscaler processors.

2)Loop level->Simd and systolic computer

3)Task level->Multiprocessor/Multicomputer

Program Optimization

- Transformation of user program to explore hardware capability.
- Explores better performance.
- Goal to maximize speed of code execution.
- To minimize code length.
- Local and global optimizations.
- Machine dependent Transformation

Parallel Code Generation

- Compiler directive can be used to generate parallel code.
- 2 optimizing compilers-:

1)Parafrase and Parafrase 2

2)PFC and Parascope

Parafrase and Parafrase2

- Transforms sequential programs of fortran 77 into parallel programs.
- Parafrase consists of 100 program that are encoded and passed.
- Pass list indentifies dependencies and converts it to concurrent program.
- Parafrase2 for c and pascal in extension to fortran.

PFC and ParaScope

- Translates fortran 77 to fortran 90 code.
- PFC package extended to PFC + for parallel code generation on shared memory multiprocessor.

- PFC performs analysis as following steps below-:

1)Interprocedure Flow analysis using call graph

2)Transformation (do-loop normalization etc)

3)dependence analysis

4)Vector Code Generation

MODULE 5

SOFTWARE FOR PARALLEL PROGRAMMING

Parallel programming models

There are six parallel programming models:

1) Shared-variable model
2) Message passing model
3) Data parallel model
4) Object oriented model
5) Functional and logic model

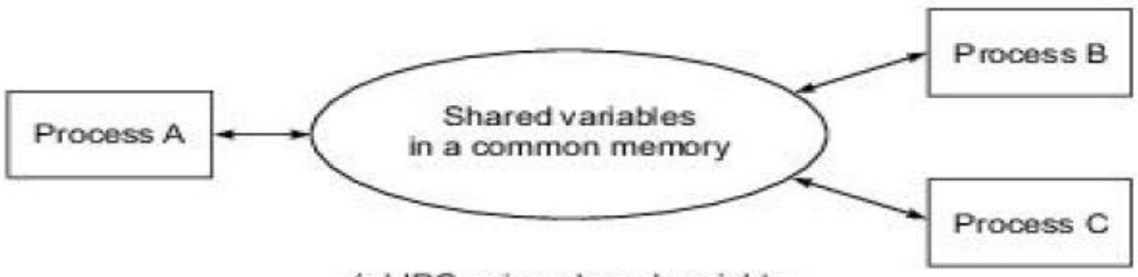
1. Shared-Variable Model

In all programming systems, we consider processors are active resources and memory and I/O devices passive resources.The basic computational units in a parallel program are processes corresponding to operations performed by related code segments.A program is a collection of processesParallelism depends on how interprocess communication (IPC) is implemented

- Fundamental issues in parallel programming are centered around the specification, creation, suspension, reactivation, migration, termination, and synchronisation of concurrent processes residing in the same or different processors

Shared-variable communication

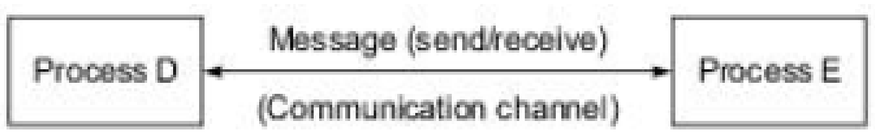
- Multiprocessor programming is based on the use of shared variables in a common memory for IPC.Shared-variable IPC demands the use of shared memory and mutual e



(a) IPC using shared variable

among multiple processes accessing the same set of variables .

- The main issues in using this model include protected access of critical sections, memory consistency, atomicity of memory operations, fast synchronization, shared data structures, and fast data movement techniques



(b) IPC using message passing

Critical Section

- A Critical Section (CS) is a code segment accessing shared variables, which must be executed by only one process at a time and which, once started, must be completed without interruption. In other words, a CS operation is indivisible and satisfies the following requirements:
- Mutual exclusion- At most one process executing the C S at a time.
- No deadlock in waiting- No circular wait by two or more processes trying to enter the CS; at least one will succeed
- Nonpreemption- No interrupt until completion, once entered the CS.
- Eventual entry- A process attempting to enter its CS will eventually succeed

Protected Access

- The main problem associated with the use of a CS is avoiding race conditions where concurrent processes executing in different orders produce different results.

Four operational modes used in programming multiprocessor systems are:

- Multiprogramming
- Multiprocessing
- Multitasking
- Multithreading

Partitioning and Replication

- The goal of parallel processing is to exploit parallelism as much as possible with the lowest overhead
- Program partitioning is a technique for decomposing a large program and data set into many small pieces for parallel execution by multiple processors
- Program replication refers to duplication of the same program code for parallel execution on multiple processors over different data sets
- Partitioning is often practiced on a shared-memory multiprocessor system, while replication is more suitable for distributed-memory message-passing multicomputers

Scheduling and Synchronization

- Scheduling of divided program modules on parallel processors is much more complicated than scheduling of sequential programs on a uniprocessor

Static scheduling is conducted at post-compile time.

- Its advantage is low overhead but the shortcoming is a possible mismatch with the runtime profile of each task and therefore potentially poor resource utilization

Dynamic scheduling catches the run-time conditions.

- However, dynamic scheduling requires fast context switching, preemption, and much more OS support.
- The advantages of dynamic scheduling include better resource utilization at the expense of higher scheduling overhead

Cache coherence and protection

- The multicache coherence problem demands an invalidation or update after each write operation
- These coherence control operations require special bus or network protocols for implementation
- A memory system is said to be coherent if the value returned on a read instruction is always the value written by the latest write instruction on the same memory location
- Sequential consistency model demands that all memory accesses be strongly ordered on a global basis

- A processor cannot issue an access until the most recently shared writable memory access has been globally perforinecl
- Weak consistency model enforces ordering and coherence at explicit synchronization points only

2. Message Passing Model

- Two processes D and E residing at different processor nodes may communicate with each other by passing messages through a direct or indirect network. The messages may be instructions, data, synchronization, or interrupt signals, etc. The communication delay caused by message passing is much longer than that caused by accessing shared variables in a common memory

Synchronous Message Passing

- Since there is no shared memory, there is no need for mutual exclusion. Synchronous message passing must synchronize the sender process and the receiver process in time and space, just like a telephone call using circuit-switched lines.In general, no buffers are used in the communication channels
- That is why synchronous communication can be blocked by channels being busy or in error since only one message is allowed to be trasnmittted via a channel at a time
- In a synchronous paradigm, the passing of a message must synchronize the sending process and the receiving process in time and space. Besides having a time connection, the sender and receiver must also be linked by physical communication channels in space. A path of channels must be ready to enable the message passing between them
- If one process is ready to communicate and the other is not, the one that is ready must be blocked (or wait). In this sense, synchronous commnunication has been also called a blocking communication scheme

Asynchronous Message Passing

- Asynchronous communication does not require that message sending and receiving be synchronized in time and space. Buffers are often used in channels, which results in nonblocking in message passing
- However, arbitrary communication delays may be experienced because the sender may not know if and when the message has been received until acknowledgment is received from the receiver. Nonblocking can be achieved by asynchronous message passing in which two processes do not have to be synchronized either in time or in space.
- The sender is allowed to send a message without blocking, regardless of whether the receiver is ready or not

- Asynchronous communication requires the use of buffets to hold the messages along the path of the connecting channels. Since channel buffers are finite, the sender will eventually be blocked

3. **Data-parallel model**

With the lockstep operations in SIMD computers, the data-parallel code is easier to write and to debug because parallelism is explicitly handled by hardware synchronization and flow control.Data-parallel languages are modified directly from standard serial programming languages. Data-parallel programs require the use of pre-distributed data sets. Thus the choice of parallel data structures makes a big difference in data-parallel programming.Interconnected data structures are also needed to facilitate data exchange operations

Data Parallelism

Ever since the introduction of the llliac IV computer, programming SIMD array processors has been a challenge for computational scientists. The main difficulty in using the llliac IV had been to match the problem size with the fixed machine size.

In other words, large arrays or matrices had to be partitioned into 64-element segments before they could be effectively processed by the 64 processing elements (PEs) in the llliac IV machine

Synchronous SIMD programming differs from asynchronous MIMD programming in that all PEs in an SIMD computer operate in a lockstep fashion, whereas all processors in an MIMD computer execute different instructions asynchronously.In an SIMD program, scalar instructions are directly executed by the control unit.

Vector instructions are broadcast to all processing elements. Vector operands are loaded into the PEs from local memories simultaneously using a global address with different offsets in local index registers.A masking pattern (binary vector) can be set under program control so that PEs can be enabled or disabled dynamically in any instruction cycle.

4. **Object-Oriented Model**

In this model, objects are dynamically created and manipulated. Processing is performed by sending and receiving messages among objects

Concurrent OOP

The popularity of OOP is attributed to three application demands:

- First, there is increased use of interacting processes by individual users, such as the use of multiple windows.
- Second, workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving.

- Third, multiprocessor technology in several variants has advanced to the point of providing supercomputing power at a fraction of the traditional cost

Program abstraction leads to program modularity and software reusability as is commonly experienced with OOP

An Actor Model

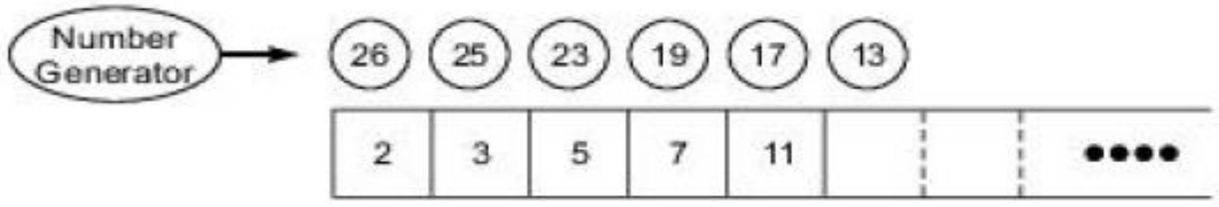
- COOP must support patterns of reuse and classification, for example, through the use of inheritance which allows all instances of a particular class to share the same property.Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing.
- In an actor model, message passing is attached with semantics.

Basic actor primitives include:

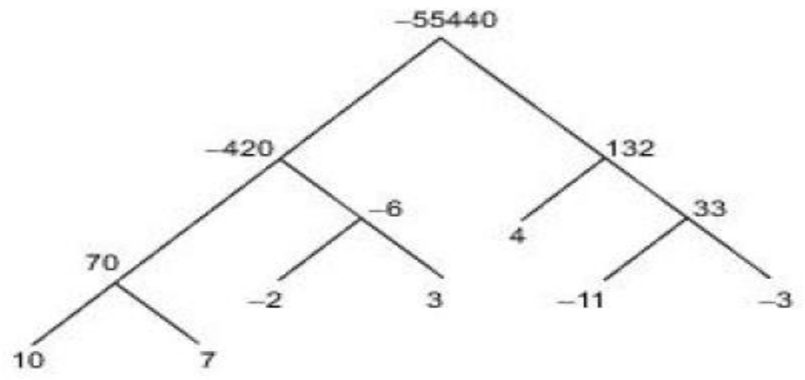
- Create - Creating an actor from a behaviour description and a set of parameters
- Send to - Sending a message to another actor
- Become - An actor replacing its own behaviour by a new behaviour.
- State changes are specified by behaviour replacement

Parallelism in COOP

- Pipeline concurrency
- Divide and conquer
- Cooperative problem solving



(a) Pipeline concurrency



(b) Divide-and-conquer concurrency

5. **Functional and Logic Models**

Two language-oriented programming models for parallel processing are:

- Functional programming model
- Logic programming model

- **Functional programming model**

A functional programming language emphasizes the functionality of a program and should not produce side effects after execution.There is no concept of storage, assignment, and branching in functional programs

In other words, the history of any computation performed prior to the evaluation of a functional expression should be irrelevant to the meaning of the expression

PARALLEL LANGUAGES AND COMPILERS

The environment for parallel computers is much more demanding than that for sequential computers. A programming environment is a collection of software tools and system software support.

Users should not have to spend a lot of time programming hardware details; they should focus instead on program parallelism using high-level abstractions

Language Features for Parallelism

Six categories according to the functionality

- Optimization features
- Availability Features
- Synchronization /Communication Features

- Control of Parallelism
- Data parallelism Features
- Process management Features

Optimization features

These features are used for program restructuring and compilation directives in convening sequentially coded programs into parallel forms.

- The purpose is to match the software parallelism with the hardware parallelism in the target machine
- Automated parallelizer
- Semi-automated parallelizer
- Interactive restructure support

Availability Features

These are features that enhance the user-friendliness, make the language portable to a large class of parallel computers, and expand the applicability of software libraries

- Scalability
- Compatibility
- Portability

Synchronization /Communication Features

- Single-assignment languages
- Shared variables (locks) for IPC
- Logically shared memory such as the tuple space in Linda
- Send/receive for message passing
- Rendezvous in Ada
- Remote procedure call
- Dataflow languages such as Id
- Barriers, mailbox, semaphores, monitors

Control of Parallelism

Listed below are features involving control constructs for specifying parallelism in various forms:

- Coarse, medium or fine grain
- Explicit versus implicit parallelism
- Global parallelism in the entire program
- Loop parallelism in iterations
- Task-split parallelism
- Shared task queue
- Divide-and-conquer paradigm
- Shared abstract data types
- Task dependency specification

Data parallelism Features

Data parallelism is used to specify how data are accessed and distributed in either SIMD or MIMD computers.

- Run-time automatic decomposition
- Mapping specification
- Virtual processor support
- Direct access to shared data
- SPMD (single program multiple data) support

Process management Features

These features are needed to support the efficient creation of parallel processes, implementation of multithreading or multitasking, program partitioning and replication, and dynamic load balancing at run lime.

- Dynamic process creation at run time
- Lightweight processes (threads)
- Replicated workers
- Partitioned networks
- Automatic load balancing

Parallel Language Constructs

Special language constructs and data array expressions are presented below for exploiting parallelism in programs.

- Fortran 90 Array Notation
- A multidimensional data array is represented by an array name indexed by a sequence of subscript triplets, one for each dimension. Triplets for different dimensions are separated by commas

$$\begin{array}{rcl} e_1 : e_2 : e_3 \\ e_1 : e_2 \\ e_1 : * : e_3 \\ e_1 : * \\ e_1 \\ * \end{array}$$

where each e_i is an arithmetic expression that must produce a scalar integer value. The first expression e_1 is a lower bound, the second e_2 an upper bound, and the third e_3 an increment stride.

For example, B(1: 4 : 3, 6 : 8 : 2,3) represents four elements B(l, 6, 3), B(4 ,6, 3), B(1, 8,3), and B(4, 8, 3) of a three-dimensional array

- When the third expression in a triplet is missing, a unit stride is assumed
- The * notation in the second expression indicates all elements in that dimension starting from e1, or the entire dimension if e1, is also omitted

When both e2 and e3 are omitted, the e1 alone represents a single element in that dimension.

- For example, A(5) represents the fifth element in the array A(3 : 7 : 2). This notation allows us to select array sections or particular array elements.
- Array assignments are permitted under the following constraints:The array expression on the right must have the same shape and the same number of elements as the array on the left
- For example, the assignment A(2 : 4, 5 : 8) =A(3 : 5, 1 : 4) is valid, but the assignment 4(1 : 4, 1 : 3) =A(1:2, 1 : 6) is not valid, even tempt each side has 12 elements.
- When a scalar is assigned to an array, the value of the scalar is assigned to every element of the array.
- For instance, the statement B(3 : 4, 5) = 0 sets B(3, 5) and B(4, 5) to 0.

Parallel Flow Control

The conventional Fortran Do loop declares that all scalar instructions within the (Do, Enddo) pair are executed sequentially, and so are the successive iterations

- To declare parallel activities, we use the (Doall, Endall) pair. All iterations in the Doall loop are totally independent of each other. This implies that they can be executed in parallel if there are sufficient processors to handle different iterations.However, the computations within each iteration are still executed serially in program order.
- When the successive iterations of a loop depend on each other, we use the (Doacross, Endacross) pair to declare parallelism with loop-carried dependences.Synchronizations must be performed between the iterations that depend on each other.For example, dependence along the J-dimension exists in the following program. We use Doacross to declare parallelism along the I-dimension, but synchronization between iterations is required

```
Doacross I = 2, N
      Do J = 2, N
S1:         A(I, J) = (A(I,(J − 1)) + A(I, J + 1))/2
      Enddo
Endacross
```

- Another program construct is the (Cobegin, Coend) pair. All computations specified within the block could be executed in parallel.

```
Cobegin
     P1
     P2
     :
     Pn
  Coend
```

The command Parbegin and Parend have the same meaning .

- Fork and join:
- During the execution of a process P, we can use a Fork Q command to spawn a new process Q:

```
Process P                    Process Q
    :                            :
Fork Q                           :
    :                           End
Join Q
```

- The Join Q command recombines the two processes into one process. Execution of Q is initialized when the Fork Q statement in P is executed. Programs P and Q are executed concurrently until either P executes the Join Q statement or Q terminates. Whichever one finishes first must wait for the other to complete execution, before they can he rejoined

Optimizing Compilers for Parallelism

Because high-level languages are used almost exclusively to write programs today, compilers have become a necessity in modem computers.The role of a compiler is to remove the burden of program optimization and code generation from the programmer.

A parallelizing compiler consists of the following three major phases:

- Flow analysis, optimizations, and code generation

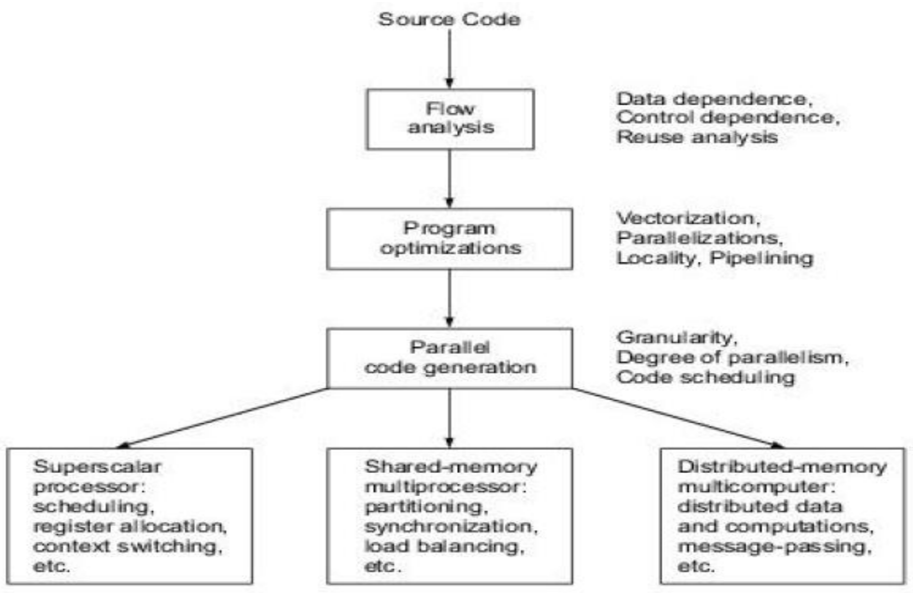

Source Code
Flow analysis
Data dependence, Control dependence, Reuse analysis
Program optimizations
Vectorization, Parallelizations, Locality, Pipelining
Parallel code generation
Granularity, Degree of parallelism, Code scheduling
Superscalar processor: scheduling, register allocation, context switching, etc.
Shared-memory multiprocessor: partitioning, synchronization, load balancing, etc.
Distributed-memory multicomputer: distributed data and computations, message-passing, etc.

Fig. 10.4 Compilation phases in parallel code generation

- **Flow Analysis**: This phase reveals the program flow patterns in order to determine data and control dependences in the source codeGenerally speaking, instruction-level parallelism is exploited in superscalar or VLSI processors; loop level in SIMD, vector, or systolic computers; and task level in multiprocessors. multicomputers, or a network of workstations

- **Program Optimizations**:This refers to the transformation of user programs in order to explore the hardware capabilities as much as possible. Transformation can be conducted at the loop level, locality level, or prefetching level with the ultimate goal of reaching global optimization. The optimization often transforms a code into an Equivalent but "better" form in the same representation language. These transformations should be machine-independent.

The ultimate goal of program optimization is to maximize the speed of code execution. This involves the minimization of code length and of memory accesses and the exploitation of parallelism in programs. The optimization techniques include vectorization using pipelined hardware and parallelization using multiple processors simultaneously.

Parallel code generation

Code generation usually involves transformation from one representation to another, called an intermediate form

Two optimizing compilers are:

- Parafrase and parafrase2

Is a source-to-source program restructurer (or compiler preprocessor) which transforms sequential Fortran 77 programs into forms suitable for vectorization or parallelization

- The PFC and Parascope

Automatic source-to-source vectorizer. It translated Fortran 97 code into Fortran 90code.

DEPENDENCE ANALYSIS OF DATA ARRAYS

Dependence testing of successive iterations in multidimensional data arrays

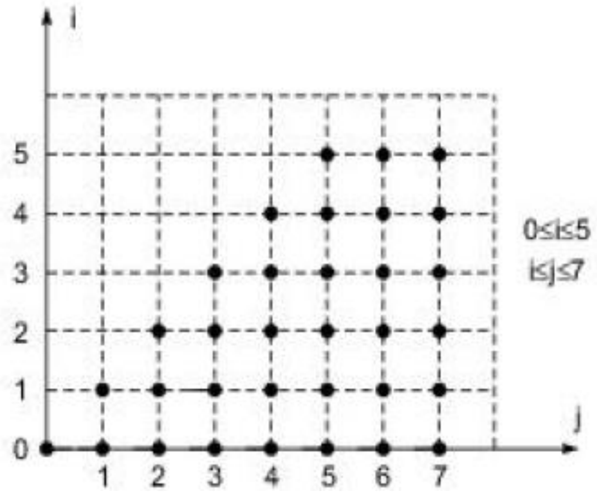
- Iteration Space and Dependence analysis
- Flow dependence, antidependence, and output dependence were defined for scalar data
- They can be summarized by the existence of dynamic references of R1 and R2, if and only if either R1, or R2 is a write operation, R1 executes before R2, or R1 and R2 both write the same variable
- When the referenced object is a data array indexed by a multidimensional subscript, the dependence becomes very difficult to determine at compile time

- Precise and efficient dependence tests are essential to the effectiveness of a parallelizing compiler. The process of computing all the data dependences in a program is called dependence analysis
- Dependence Testing
- Calculating data dependence for analysis complicated by the fact that two array references may not access the same memory location.
- Dependence testing is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest

Iteration space

- The n-dimensional discrete Cartesian space for n-deep loops is called an iteration space.
- The iteration is represented as coordinates in the iteration space.
- The following example clarifies the concept of lexicographic order for the successive iterations in a loop nest

```
Do $i_1 = L_1, U_1$
    Do $i_2 = L_2, U_2$
        ...
        Do $i_n = L_n, U_n$
$S_1$ :         $A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \cdots$
$S_2$ :         $\cdots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))$
        Enddo
        ...
    Enddo
Enddo
```



The following sequential order of iteration is a lexicographic order:

(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7)
(1,1), (1, 2), (1, 3), (1, 4), (1,5), (1,6), (1, 7)
(2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7)
(3, 3), (3, 4), (3, 5), (3, 6), (3, 7)
(4, 4), (4, 5), (4, 6), (4, 7)
(5, 5), (5, 6), (5, 7)

- The lexicographic order is important to performing matrix transformation, which can be applied for loop optimization

CODE OPTIMIZATION AND SCHEDULING

Describe the roles of compilers in code optimization and code generation for parallel computers

Scalar Optimization with Basic Blocks

- Instruction scheduling is often supported by both compiler techniques and dynamic scheduling hardware. In order to exploit instruction-level parallelism (ILP), we need to optimize the code generation and scheduling process under both machine and program constraints
- Machine constraints are caused by mutually exclusive use of functional units, registers, data paths, and memory. Program constraints are caused by data and control dependences

Static instruction scheduling

- Provide an additional set of nontrapping instructions
- This approach requires an extension of the instruction set of existing processors

Dynamic instruction scheduling

- To support out-of-order execution
- This approach usually does not require the instruction set to be modified but requires complex hardware support

In general, instruction scheduling methods ensure that control dependences, data dependences, and resource limitations are properly handled during concurrent execution.

- The goal is to produce a schedule that minimizes the execution time or the memory demand, in addition to enforcing correctness of execution.
- Static scheduling at compile time requires intelligent compilation support, whereas dynamic scheduling at run time requires sophisticated hardware support

Precedence Constraints

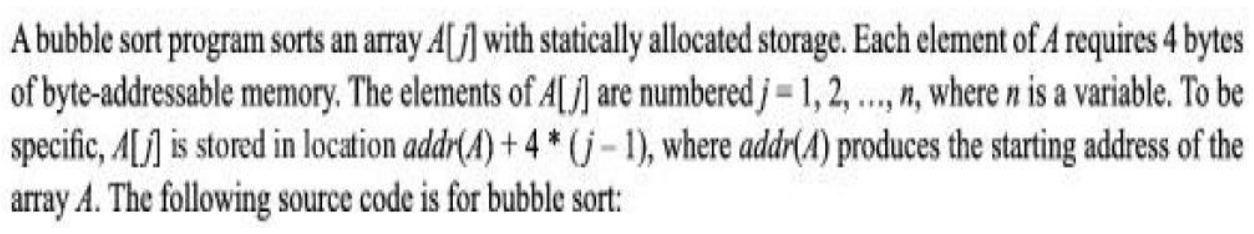
- If a flow dependence is detected, the write must proceed ahead of the read operation involved. Similarly, output dependence produces different results if two writes to the same location are executed in a different order. Antidependence enforces a read to be ahead of the write operation involved.
- We need to analyze the memory variables.
- Scalar data dependence is much easier to detect. Dependence among arrays of data elements is much more involved

Basic Block Scheduling

- A basic block (block) is a sequence ofstatements satisfying two properties:No statement but the first can be reached from outside the block; i.e. there are no branches into the middle of the block.
- All statements are executed consecutively if the first one is. Therefore, no branches out or halts are allowed until the end of the block. All blocks are required to be maximal in the sense that they cannot be extended up or down without violating these properties
- An extended basic block is defined as a sequence of statements in which the first statement is the only entry point. Thus an extended block may have branches out in the middle of the code but no branches into it

The basic steps for constructing basic blocks:

- Find the leaders, which are the first statements in a block. Leaders are identified as being one or more of the following:
    - The first statement of the code.
    - The target of a conditional or unconditional branch.
    - A statement following a conditional branch.
- For a leader, a basic block consists of the leader and all statements following up to but excluding the next leadcr.



A bubble sort program sorts an array $A[j]$ with statically allocated storage. Each element of A requires 4 bytes of byte-addressable memory. The elements of $A[j]$ are numbered $j = 1, 2, \ldots, n$, where n is a variable. To be specific, $A[j]$ is stored in location $addr(A) + 4 * (j - 1)$, where $addr(A)$ produces the starting address of the array A. The following source code is for bubble sort:

```
For i := n − 1 downto 1 do
    For j := 1 to i do
        If A[j] > A[j + 1] then
            Begin
                temp := A[j]
                A[j] := A[j + 1]
                A[j + 1] := temp
            End
    End of j-loop
End of i-loop
```

If a three-address machine is assumed, the above code is translated into the following assembly language code.

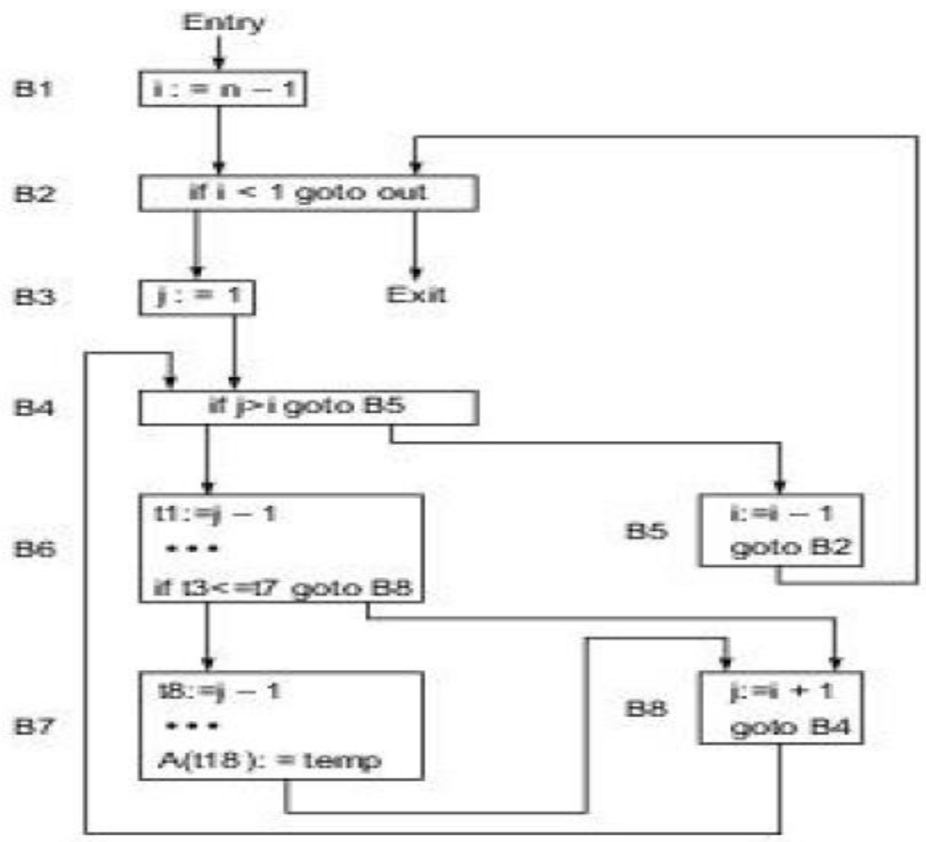
- Variable names an the right of := stand for values, and on the left for addresses.

```
        i := n − 1
s5:     if i < 1 goto sl
        j := 1
s4:     if j > i goto s2
        tl := j − 1
        t2 := 4 * tl
        t3 := A[t2]                     /A[j]/
        t4 := j + 1
        t5 := t4 − 1
        t6 := 4 * t5
        t7 := A[t6]                     /A[j+1]/
```

```
if t3 < = t7 goto s3        /if A[j] > A[j+1] then begin .../
t8 := j – 1
t9 := 4 * t8
temp := A[t9]               / temp := A[j]/
t10 := j + 1
t11 := tl0 – 1
t12 := 4 * t11
t13 := A[t12]               /A[j+1] /
t14 := j − 1
t15 := 4 * t14
A[t15] := t13               /A[j] := A[j+1]/
t16 := j + 1
t17 := t16 − 1
t18 := 4 * t17
A[t18] := temp              /A[j+1] := temp/
```

```
s3:   j := j + 1
      goto s4
s2:   i := i – 1
      goto s5
s1:   halt
```

The above 31 statements are divided into 8 basic blocks



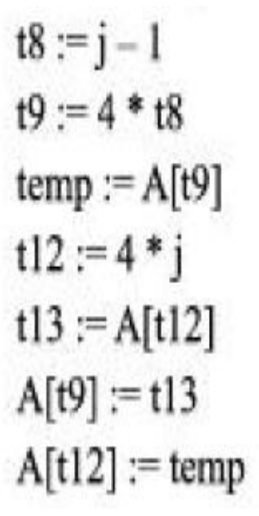
Local and Global Optimizations

- These are code optimizations performed only within basic blocks. The information needed for optimization is gathered entirely from a single basic block, not from an extended basic block. No control—flow information between blocks is considered.

Listed below are some local optimizations often performed:

- **Local common subexpression elimination**

If a subexpression is to be evaluated more than once within a single block, it can be replaced by a single evaluationin block B7, t9 and tl5 each compute 4 *(j-1), and t12 and t18 each compute 4 *j. Replacing t15 by t9, and t18 by t12, we obtain the following revised code for B7, which is shorter to execute.



```
t8 := j - 1
t9 := 4 * t8
temp := A[t9]
t12 := 4 * j
t13 := A[t12]
A[t9] := t13
A[t12] := temp
```

- **Local Constant Folding or Propogation**

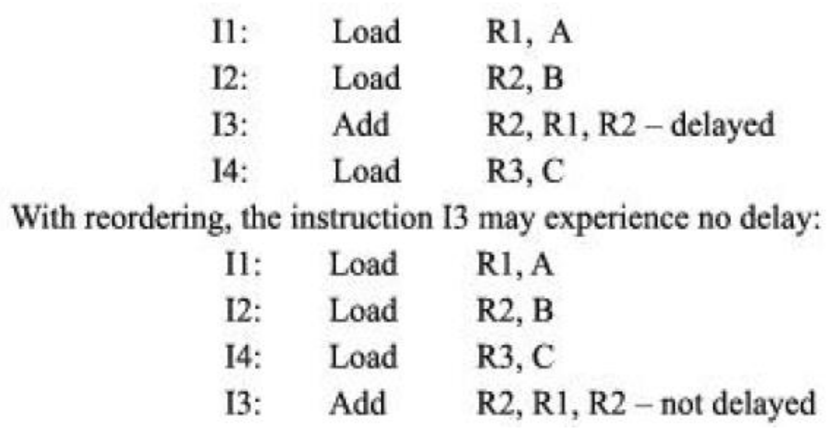
  Sometimes some constants used in instructions can be computed at compile time. This often takes place in the initialization blocks. The compile-time generated constants are then folded to eliminate unnecessary calculations at run time, in other cases, a local copy may be propagated to eliminate unnecessary calculations

- **Algebraic Optimization to Simplify Expressions**

  For example, one can replace the identity statement A := B + 0 or A := B * 1 by A := B and later even replace references to this A by references to B. Or one can use the commutative law to combine expressions C := A + B and D := B + A. The associative and distributive law can also be applied on equal-priority operators, such as replacing (a-b) - c by a -(b-c) if (b-c) has already been evaluated earlier.

- **Instruction reordering**

Code reordering is often practiced to maximize the pipeline utilization or to enable overlapped memory accesses. Some orders yield better code than others. Reordered instructions lead to better scheduling, preventing pipeline or memory delays **.**



```
I1:    Load    R1, A
I2:    Load    R2, B
I3:    Add     R2, R1, R2 – delayed
I4:    Load    R3, C
```

With reordering, the instruction I3 may experience no delay:

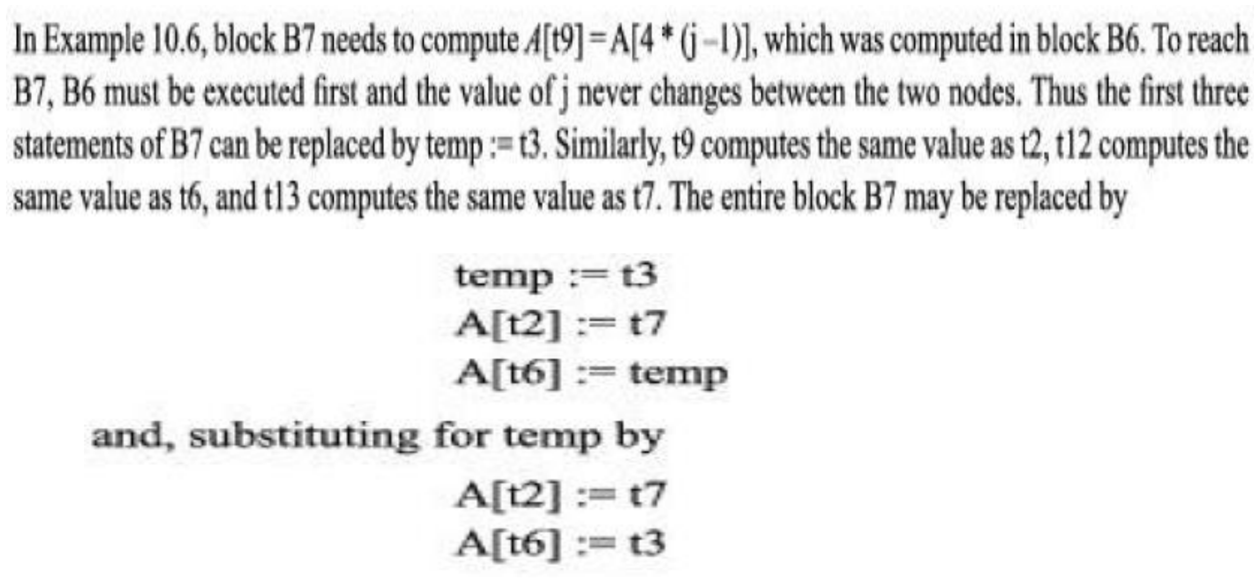
```
I1:    Load    R1, A
I2:    Load    R2, B
I4:    Load    R3, C
I3:    Add     R2, R1, R2 – not delayed
```

Global Optimizations

These are code optimizations performed across basic block boundaries.

- **Global versions of local optimizations**

These include global common subexpression elimination, global constant propagation, dead code elimination, etc. The following example further optimizes the code in Example if some global optimizations are performed



In Example 10.6, block B7 needs to compute *A*[t9] = A[4 * (j – 1)], which was computed in block B6. To reach B7, B6 must be executed first and the value of j never changes between the two nodes. Thus the first three statements of B7 can be replaced by temp := t3. Similarly, t9 computes the same value as t2, t12 computes the same value as t6, and t13 computes the same value as t7. The entire block B7 may be replaced by

```
temp := t3
A[t2] := t7
A[t6] := temp
```

and, substituting for temp by

```
A[t2] := t7
A[t6] := t3
```

- The revised program, after both local and global optimizations, is obtained as follows:

```
B1:  i := n – 1
B2:  If i < 1 goto out
B3:  j := 1
B4:  If j > i goto B5
B6:  t1:= j – 1
     t2 := 4 * t1
     t3 := A[t2]                /A[j]/
     t6 := 4 * j
     t7 := A[t6]                /A[j+1]/
     If t3 <= t7 goto B8
B7:  A[t2] := t7
     A[t6] := t3
B8:  j := j + 1
     goto B4
B5:  i := i – 1
     goto B2
out:
```

Loop Optimizations

These include various loop transformations for the purpose of vectorization, parallelization, or both. Sometimes code motion and induction variable elimination can simplify loop structures.

- For example, one can replace the calculation of an induction variable involving a multiplication by an addition to its former value.
- The addition takes less time to perform and thus results in a shorter execution time.

In other cases, loop-invariant variables or codes can be moved out of the loop to simplify the loop nest.

- One can also lower the loop control overhead using loop unrolling to reduce iteration or loop fusion to merge loops

Control-flow Optimization

These are other global optimizations dealing with control structure but not directly with loops.

- A good example is code hoisting, which eliminates copies of identical code on parallel paths in a flow graph. This can save space significantly, but would have no impact on execution time

Machine-Dependent Optimizations

With a finite number of registers, memory cells, and functional units in a machine, the efficient allocation of machine resources affects both space and time optimization of programs. Strength reduction replaces complex operations by cheaper operations, such as replacing 2a by a + a, a2 by a * a and length(S1+S2) by length (S1) + length(S2)

Vectorization and Parallelization Methods

Besides scalar optimizations, we need to perform vector and for parallel optimizations. The purpose is to improve the performance of programs that manipulate large data arrays or can be partitioned for parallel execution.

- Vectorization is the process of converting scalar looping operations into equivalent vector instruction execution.
- Parallelition aims at converting sequential code into parallel form, which can enable parallel execution by multiple processors

An optimizing compiler that does vectorization automatically or semi-automalically with directives from programmers is-called a vectorizing compiler or simply a vectorizer. Similarly, a parallelizing compiler should be designed to generate parallel code from sequential code automatically or semi-automatically.

Vectorization methods

- We use Fortran 90 notation; for example, successive iterations in the following loop are totally independent:

```
        Do  20 I = 8, 120, 2
20             A(I) = B(I+3) + C(I+1)
```

- This scalar loop can be convened into one vector-add instruction defined by the following array assignment:

```
A(8:120:2) = B(11:123:2) + C(9:121:2)
```

Use of temporary storage

```
        Do 20 I = 1, N
           A(I) = B(I) + C(I)
20         B(I) = 2 * A(I+1)
```

This loop represents the following sequence of scalar operations:

```
A(1) = B(1) + C(1)
B(1) = 2 * A(2)
A(2) = B(2) + C(2)
B(2) = 2 * A(3)
      ⋮
```

In order to enable pipelined execution by vector hardware, we need to introduce a temporary array TEMP(1:N) to produce the following vector code:

```
TEMP(1:N) = A(2:N+1)
A(1:N) = B(1:N) + C(1:N)
B(1:N) = 2 * TEMP(1:N)
```

Loop interchanging

Vectorization is often performed in the inner loop rather than in the outer loop.

- Sometimes we interchange the loops to enable the vectorization
- The general rules for loop interchanges are to make the most profitable vectorizable loop the innermost loop, to make the most profitable parallelizable loop the outermost loop, to enable memory accesses to consecutive elements in arrays

```
      Do 20 I = 2, N
         Do 10 J = 2, N
S1:         A(I, J) = (A(I, J – 1) + A(I, J + 1)/2
10       Continue
20    Continue
```

- The statement s1 is both flow dependent and anti-dependent on itself. The inner loop cannot be vectorized in j-dimension

```
     Do  20 J = 2, N
          Do  20 I = 2, N
               A(I, J) = (A(I, J – 1) + A(I, J + 1))/2
20   Continue
```

- Now the inner loop can be vectorized in i-dimension

```
     Do 20 J = 2, N
          A(2:N, J) = (A(2:N, J – 1) + A(2:N, J + 1))/2
20   Continue
```

Loop Distribution

- Nested loops can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests.

```
          Do  10 I = 1, N
               B(I, 1) = 0
               Do  20 J = 1, M
                    A(I) = A(I) + B(I, J) * C(I, J)
20             Continue
               D(I) = E(I) + A(I)
10        Continue
```

- The I-loop is distributed to three copies, separated by the nested J-loop from the assignment to array B and D, and vectorized as follows:

```
     B(1:N, 1) = 0 (a zero vector)
     Do  30 I = 1, N
          A(I) = A(I) + B(I, 1:M) * C(I, 1:M)
30   Continue
     D(1:N) = E(1:N) + A(1:N)
```

Vector Reduction

- In general, a vector reduction produces a scalar value from one or two data arrays.

Examples include the sum, product, maximum, and minimum of all the elements in a single array. A dot product produces a scalar S from two array

```
      Do 40 I = 1, N
S1:       A(I) = B(I) + C(I)
S2:       S = S + A(I)
S3:       AMAX = MAX(AMAX, A(I))
40    Continue
```

- Each statement can be recognized as reduction operation and can be vectorized as:

```
S1:  A(1:N) = B(1:N) + C(1:N)
S2:  S = S + SUM(A(1:N))
S3:  AMAX = MAX(AMAX, MAXVAL(A(1:N)))
```

Node Splitting

- The data dependence cycle can sometimes be broken by node splitting. Consider the following loop:

```
      Do  50 I = 2, N
S1:       T(I) = A(I – 1) + A(I + 1)
S2:       A(I) = B(I) + C(I)
50    Continue
```

```
      Do  50 I=2, N
S1a:      X(I) = A(I + 1)
S2:       A(I) = B(I) + C(I)
S1b:      T(I) = A(I – 1) + X(I)
50    Continue
```

Vectorization Inhibitors

Listed below are some conditions inhibiting or preventing vectorization:

Computed conditional statements such as IF statements which depend on runtime conditions.

- Multiple loop entries or exits (not basic blocks).

- Function or subroutine calls.
- Input/output statements
- Recurrences and their variations
- A recurrence exists when a value calculated in one iteration of a loop might be referenced in another iteration

Code Parallelization

Parallel code optimization spreads a single program into many threads for parallel execution by multiple processors.

- The purpose is to reduce the total execution time.
- Each thread is a sequence of instructions that must execute on a single processor

```
        Doall I = 2, N
            Do J = 2, N
S1 :            A(I, J) = (A(I, J − 1) + A(I, J + 1)) / 2
            Enddo
        Endall
```

Each of the N -1 iterations in the outer loop can be scheduled for a single processor to execute

- Each newly created thread consists of one entire J-loop with a constant index value for I.
- If dependence does exist between the iterations, the Doacross construct can be used with proper synchronization among the iterations

Five execution modes of a FX/Fortran loop on the Alliant FX/80 multiprocessor

- Scalar
- Vector
- Scalar-concurrent
- Vector concurrent
- Concurrent outer/vector iner(COVI) modes

By using array A, the computations involved are expressed by a pure scalar loop:

```
Do K = 1, 2048
    A(K) = A(K) + S
Enddo
```

```
A(1) = A(1) + S, A(2) = A(2) + S, ..., A(2048) = A(2048) + S
```

(a) Scalar execution on one processor in 30,616 cycles

The same code can be vectorized into eight vector instructions and executed serially on a single processor equipped with vector hardware. Each vector instruction works on 256 iterations of the following loop:

A(1:2048:256) = A(1:2048:256) + S

```
A(1: 256) = A(1: 256) + S, A(257: 512) = A(257 : 512) + S, ...
A(1793 : 2048) = A(1793 : 2048) + S
```

(b) Vector execution on one processor sequentially in 6048 cycles

The scalar-concurrent mode is shown in Fig. 10.8c. Eight processors are used in parallel, performing the following scalar computations:

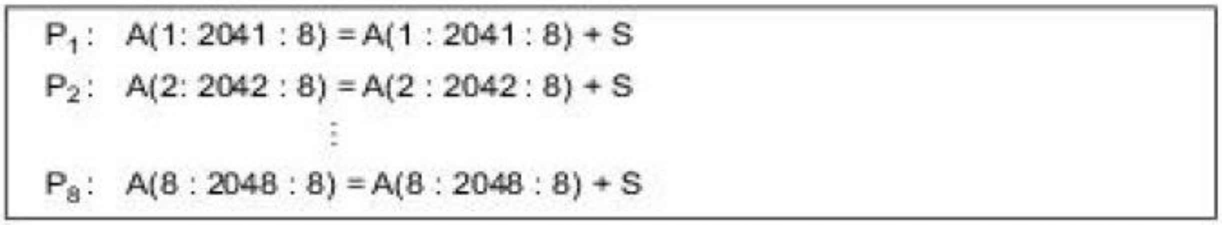
```
Doall J = 1, 8
    Do I = 1, 256
        B(I, J) = B(I, J) + S
    Enddo
Endall
```

```
P1:  B(1, 1) = B(1, 1) + S, B(1, 2) = B(1, 2) + S, ..., B(1,256) = B(1, 256) + S
P2:  B(2, 1) = B(2, 1) + S, B(2, 2) = B(2, 2) + S, ..., B(2, 256) = B(2, 256) + S
                                     ⋮
P8:  B(8, 1) = B(8, 1) + S, B(8, 2) = B(8, 2) + S, ..., B(8,256) = B(8, 256) + S
```

(c) Scalar-concurrent execution on eight processors in 3992 cycles

Figure 10.8d shows the vector-concurrent mode on eight processors, all equipped with vector hardware. The following vector codes are executed in parallel:

```
Doall J = 1, 8
    A(K:2040+K:8) = A(K:2040+K:8) + S
Endall
```

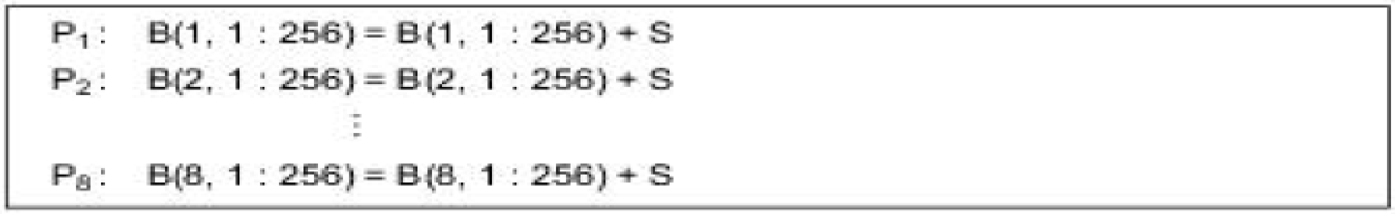


```
P1:  A(1: 2041 : 8) = A(1 : 2041 : 8) + S
P2:  A(2: 2042 : 8) = A(2 : 2042 : 8) + S
                  ⋮
P8:  A(8 : 2048 : 8) = A(8 : 2048 : 8) + S
```

(d) Vector-concurrent execution on eight processors in 960 cycles

Finally, the same program can be executed in COVI mode. The inner loop executes in vector mode, and the outer loop is executed in parallel mode. In Fortran 90 notation, we have:

```
B(1:8, 1:256) = B(1:8, 1:256) + S
```



```
P1:  B(1, 1 : 256) = B(1, 1 : 256) + S
P2:  B(2, 1 : 256) = B(2, 1 : 256) + S
                ⋮
P8:  B(8, 1 : 256) = B(8, 1 : 256) + S
```

(e) COVI execution on eight processors in 756 cycles

Inhibitors of Parallelization

- Multiple entries or exits.
- Function or subroutine calls.
- Input/output statements.
- Nondeterminism of parallel execution.
- Loop-carried dependences.

Code Generation and Scheduling

- Issues involved in code generation include order of execution, instruction selection, register allocation, branch handling, post-optimizations, etc.

Directed Acyclic Graph

Because instructions within each basic block are sequenced without any backtracks, computations performed can thus be represented by a directed acyclic graph (DAG).A DAG can he built in one pass through a basic block. The nodes in a DAG represent values. Each interior node is labelled by the operator that produces its value. Edges on the DAG show the data dependence constraints.

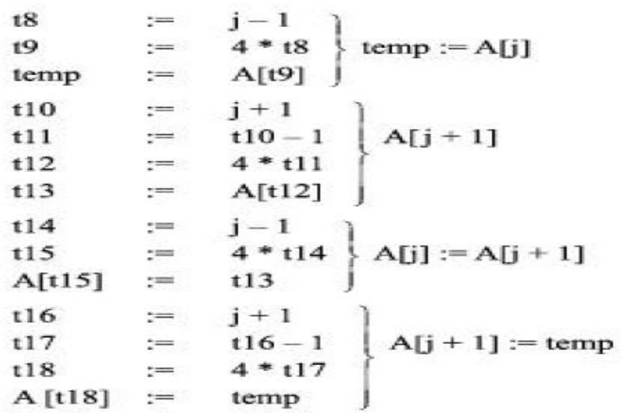
- The children of a node are the nodes producing the operand values.
- The leaf nodes carry the initial values or constants existing on entry to a basic block

DAG construction repeats the following steps from node to node.

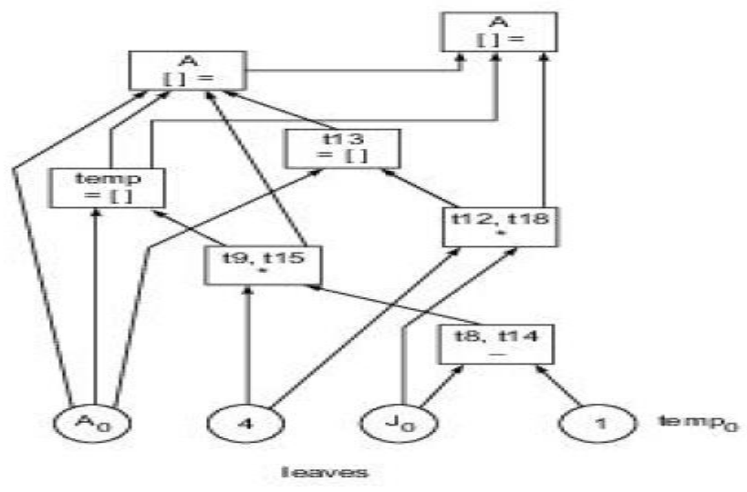
- Consider the statement A := B + C in a basic block
- We first find nodes representing the values of B and C. If B and C are not computed in the block, they must be retrieved from leaf nodes. Otherwise, B and C should come from interior nodes of the DAG. Then we create a node labelled "+". Children of this node are the nodes for values of B and C

If there is already an identical node [same label and same child nodes], node creation can be skipped. The node for "+" becomes the current node for A

Construction of a DAG for the inner loop kernel of the bubble sort program



```
t8        :=   j - 1     }
t9        :=   4 * t8    } temp := A[j]
temp      :=   A[t9]     }
t10       :=   j + 1     }
t11       :=   t10 - 1   } A[j + 1]
t12       :=   4 * t11   }
t13       :=   A[t12]    }
t14       :=   j - 1     }
t15       :=   4 * t14   } A[j] := A[j + 1]
A[t15]    :=   t13       }
t16       :=   j + 1     }
t17       :=   t16 - 1   } A[j + 1] := temp
t18       :=   4 * t17   }
A [t18]   :=   temp      }
```



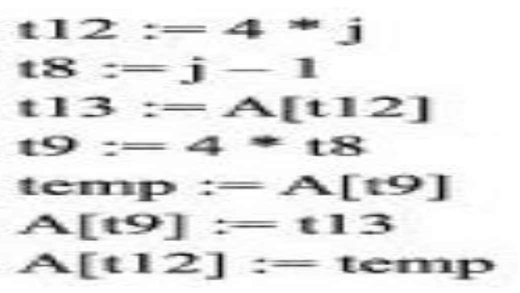
List Scheduling

A DAG represents the flow of instructions in a basic block. A topological sort can be used to schedule the operations. Let READY be a buffer holding all nodes which are ready to execute.

- Initially, the READY buffer holds all leaf nodes with Zero predecessors.
- Schedule each node in READY as early as possible, until it becomes empty.
- After all the predecessor (children) nodes are scheduled, the successor (parent) node should be immediately inserted into the READY buffer.

With list scheduling, each interior node is scheduled after its children.

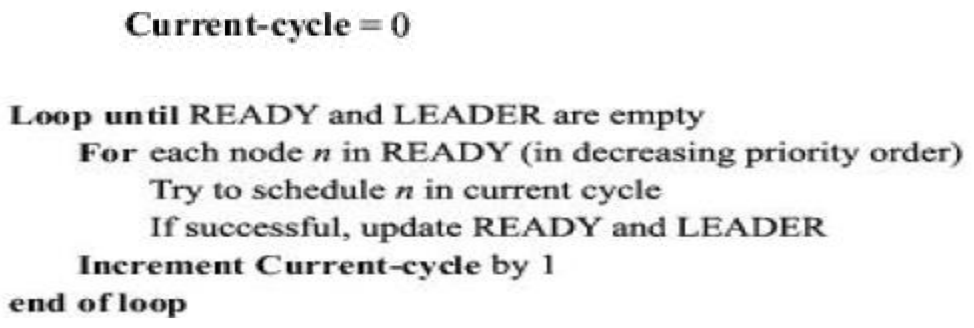
- Additional ordering constraints are needed for a procedure call or assignment through a pointer.
- When the root nodes are reached, the schedule is produced



```
t12 := 4 * j
t8 := j – 1
t13 := A[t12]
t9 := 4 * t8
temp := A[t9]
A[t9] := t13
A[t12] := temp
```

Cycle Scheduling

- List scheduling is operation-based, which has the advantage that the highest-priority operation is scheduled first. Another scheduling method for instructions in basic blocks is based on a cycle scheduling concept in which "cycles" rather “operations” are scheduled in order.
- Let READY be a buffer holding nodes with zero unscheduled predecessors ready to execute in a current cycle. Let LEADER be a buffer holding nodes with zero unscheduled predecessors but not ready in a current cycle



```
    Current-cycle = 0

Loop until READY and LEADER are empty
    For each node n in READY (in decreasing priority order)
        Try to schedule n in current cycle
        If successful, update READY and LEADER
    Increment Current-cycle by 1
end of loop
```

LOOP PARALLELIZATION AND PIPELINING

This section describes the theory and application of loop transformations for vectorization or parallelization purposes

Loop Transformation Theory

- Parallelizing loop nests is one of the most fundamental program optimization techniques demanded in a vectorizing and parallelizing compiler.The goal is to maximize the degree of parallelism or data locality in the loop nest

Elementary Transformations

- A loop transformation rearranges the execution order of the iterations in a loop nest. Three elementary loop transformations are introduced below.

SYNCHRONIZATION AND MULTIPROCESSING MODES

Principles of Synchronization

- The performance and correctness of a parallel program execution rely heavily on efficient synchronization among concurrent computations in multiple processors.The source of the synchronization problem is the sharing of writable objects (data or structures) among processes.
- Once a writable object permanently becomes read-only, the synchronization problem vanishes at that point.Synchronization consists of implementing the order of operations in an algorithm by observing the dependences for writable data

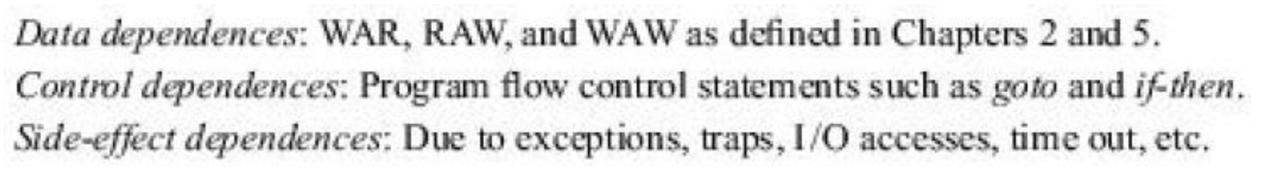
- Atomic operations
- Wait protocols
- Fairness policies
- Access order
- Sole-access protocols

Atomic Operations

- Two classes of shared memory access operations arean individual write or read such as Registerl := x and an indivisible read-modify-write such as x:=f(x) or y =f(x)
- From the synchronization point of view, the order of program operations is described by read-modify-write operations over shared writable objects called atoms. An operation on an atom is called an atomic operation
- Hard atom is one whose access races are resolved by hardware such as Test&Set, whereas a soft atom is one whose access races are resolved by software such as a shared data structure protected by a Test&Set bit

The execution of operations may be out of program order as long as the execution order preserves the meaning of the code.

- Three kinds of program dependencies are identified below:



Data dependences: WAR, RAW, and WAW as defined in Chapters 2 and 5.
Control dependences: Program flow control statements such as *goto* and *if-then*.
Side-effect dependences: Due to exceptions, traps, I/O accesses, time out, etc.

Wait protocols

Busy wait

- The process remains loaded in the processor's context registers and is allowed to continually retry
- While it does consume processor cycles, the reason for using busy wait is that it offers a faster response when the shared object becomes available

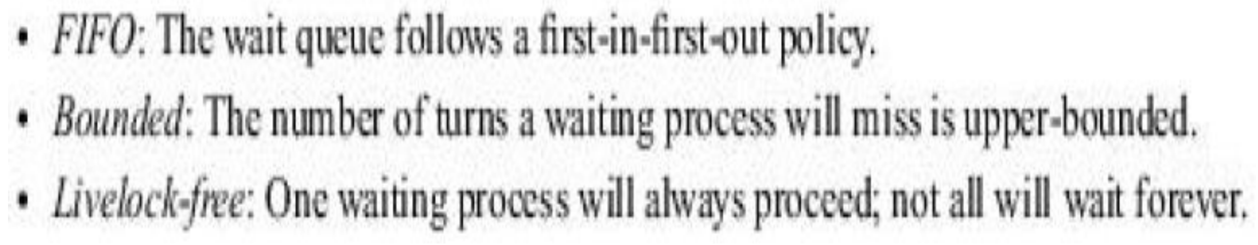
Sleep wait

- The process is removed from the processor and put in a wait queue.
- The process being suspended must be notified of the event it is waiting for.

Fairness policies

- Busy wait may reduce synchronization delay when the shared object becomes available.
- However, it wastes processor cycles by continually checking the object state and also may cause hot spots in memory access.
- In sleep wait, the resources are better utilized, but a longer synchronization delay may result.

For all suspended processes waiting in a queue, a fairness policies must be used to revive one of the waiting processes



- *FIFO*: The wait queue follows a first-in-first-out policy.
- *Bounded*: The number of turns a waiting process will miss is upper-bounded.
- *Livelock-free*: One waiting process will always proceed; not all will wait forever.

Sole- access protocols

Three synchronization methods are described below based on who updates the atom and whether sole access is granted before or after the atomic operations:

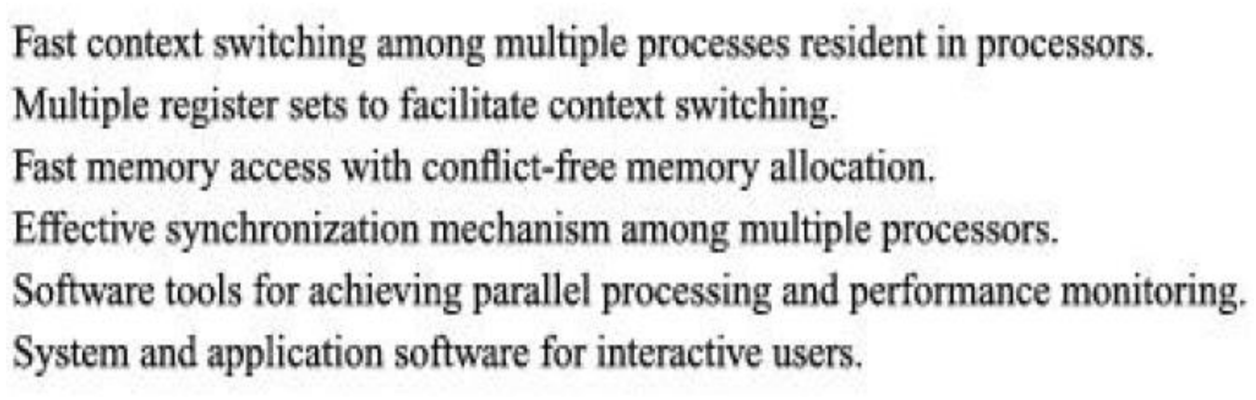
- Lock Synchronization :In this method, the atom is updated by the requester process and sole access is granted before the atomic operation. For this reason, it is also called pre-synchronization
- Optimistic Synchronization :This method also updates the atom by the requester process. But sole access is granted after the atomic operation, as described below. It is also called post-synchronization. A process may secure sole access after first completing an atomic operation on a local version of the atom and then executing another atomic operation on the global version of the atom.
- Server synchronization:This method updates the atom by the server process of the requesting process, as suggested by the name. Compared with lock synchronization and optimistic synchronization, server synchronization offers full service.An atom has a unique update server. A process requesting an atomic operation on the atom sends the request to the atom‘s update server. The update server may be a specialized server processor (SP) associated with the atom‘s memory module.

Multiprocessor execution modes

Multiprocessor supercomputers are built for vector processing as well as for parallel processing across multiple processors

- Multiprocessing Requirement: Multiprocessing at the process level requires the use of shared memory in a tightly coupled system.

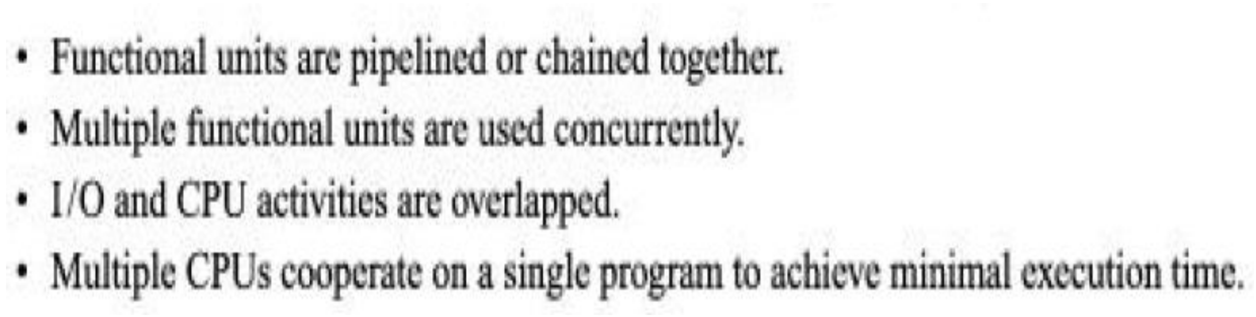
Summarized below are special requirements for facilitating efficient multiprocessing:



Fast context switching among multiple processes resident in processors.
Multiple register sets to facilitate context switching.
Fast memory access with conflict-free memory allocation.
Effective synchronization mechanism among multiple processors.
Software tools for achieving parallel processing and performance monitoring.
System and application software for interactive users.

Multitasking Environment:

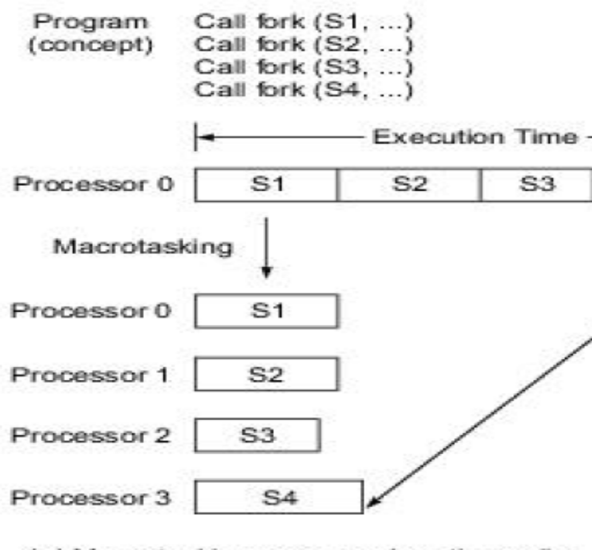
Multitasking exploits parallelism at several levels:



- Functional units are pipelined or chained together.
- Multiple functional units are used concurrently.
- I/O and CPU activities are overlapped.
- Multiple CPUs cooperate on a single program to achieve minimal execution time.

Multitasking on Cray Multiprocessor

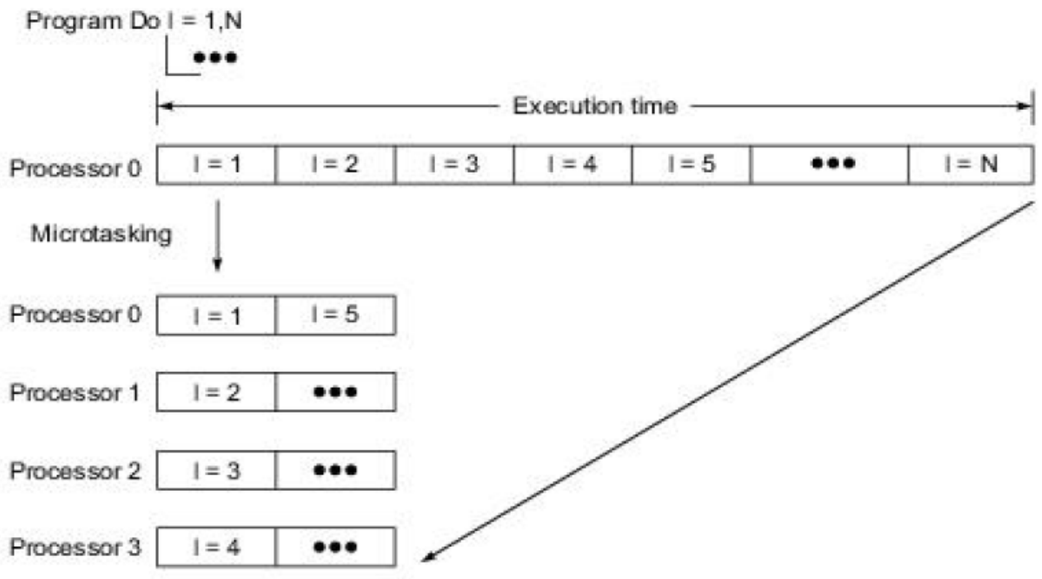
- Macrotasking -When multitasking is conducted at the level of subroutine calls



(a) Macrotasking across subroutine calls

Microtasking

- This corresponds to multitasking at the loop control level with finer granularity



(b) Microtasking across loop iterations

Autotasking

- The autotasking feature automatically divides a program into discrete tasks for parallel execution on a multiprocessor.

- In the past, macrotasking was achieved by direct programmer intervention.
- Microtasking was aided by an interactive compiler.

COMPILER-DETECTED INSTRUCTION LEVEL PARALLELISM

In the process of translating a sequential source program into machine language, the compiler performs extensive syntactic and semantic analysis of the source program. The compiler can uncover the instruction level parallelism which is implicit in the program.

- **Loop unrolling**

One relatively simple technique which the compiler can employ is know as loop unrolling, by which independent instructions from multiple successive iterations of a loop can be made to execute in parallel

Unrolling means that the body of the loop is repeated n times for n successive values of the control variable so that one iteration of the transformed loop performs the work of n iterations of the original loop.

```
for i = 0 to 58 do
     c[i] = a[i]*b[i] - p*d[i];
```

```
for j = 0 to 52 step 4 do
{
  c[j] = a[j]*b[j] - p*d[j];
  c[j+1] = a[j+1]*b[j+1] - p*d[j+1];
  c[j+2] = a[j+2]*b[j+2] - p*d[j+2];
  c[j+3] = a[j+3]*b[j+3] - p*d[j+3];
}
c[56] = a[56]*b[56] - p*d[56];
c[57] = a[57]*b[57] - p*d[57];
c[58] = a[58]*b[58] - p*d[58];
```

In the unrolled program fragment, the loop contains four independent instances of the original loop body—indeed this is the meaning of loop unrolling

- If the processor has sufficient floating point arithmetic resources—instructions from the four loop iterations can he in progress in parallel on the various functional units.It is clear that code length of the machine language program increases as a result of loop unrolling; this increase may have an effect on the cache hit ratio.

- Also, more registers are needed to exploit the instruction level parallelism within the longer unrolled loop. To discover and exploit the parallelism implicit in loops, the compiler must perform the loop unrolling transformation to generate the machine codeClearly, this strategy makes sense only if sufficient hardware resources are provided within the processor for executing instructions in parallel
- In the example above, the loop control variable in the original program goes from 0 to 58—i.e. Its initial and final values are both known at compile time. If, on the other hand, the loop control values are not known at compile time, the compiler must generate code to calculate at run-time the control values for the unrolled loop.
- When the compiler schedules machine instructions for execution on the processor, the form of scheduling is known as static scheduling. Instruction scheduling carried out by the processor hardware on the fly is known as dynamic scheduling.If the compiler is to schedule machine instructions, then it must perform the required dependence analysis amongst instructions
- Dependences amongst references to simple variables, or amongst array elements whose index values are known at compile time, can be analyzed relatively easily at compile time.But when pointers are used to refer to locations in memory, or when array index values are known only at run-time, then clearly dependence analysis is not possible at compile time. Therefore processor hardware must provide support at run-time for alias analysis—i.e. based on the respective effective addresses, to determine whether two memory accesses for read or write operations refer to the same location.
- Another reason why static scheduling by the compiler must be backed up by dynamic scheduling by the processor hardware : Cache misses, I/O interrupts, hardware exceptions cannot be predicted at compile time

Operand forwarding

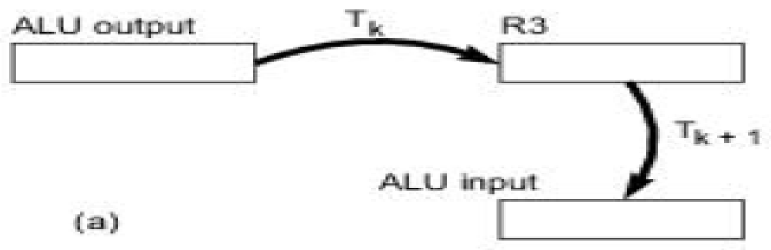
We now know that pipeline flushes caused by conditional branch, indirect jump, and procedure return instructions lead to degradation in performance, and therefore attempts must be made to minimize them;

Similarly pipeline stalls caused by data dependences and cache misses also have adverse impact on processor performance.Therefore the strategy should be to minimize the number of pipeline stalls and flushes encountered while executing an instruction stream. In other words, we must minimize wasted processor clock cycles within the pipeline and also, if possible, within the various functional units of the processor.

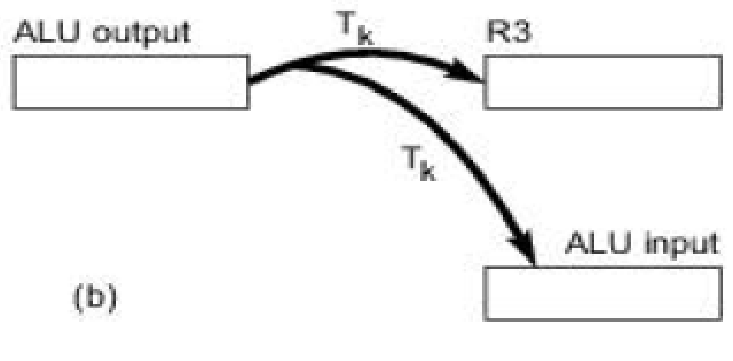
Operand forwarding helps in reducing the impact of true data dependences in the instruction stream.

Consider the following simple sequence of two instructions in a running program:

- There is a RAW dependence between 2 instructions-the output of the first is used as input to the second. In a pipelined processor, ideally the second instruction should be executed one stage and therefore one clock cycle behind the first. However, the difficulty here is that it takes one clock cycle to transfer ALU output to destination register R3, and then another clock cycle to transfer the contents of register R3 to ALU input for the right shift
- Thus a total of two clock cycles are needed to bring the result of the first instruction where it is needed for the second instruction. Therefore, as things stand, the second instruction above cannot be executed just one clock cycle behind the first.



But note that the required two transfers of data can be achieved in only one clock cycle if ALU output is sent to both R3 and ALU input in the same clock cycle. In general, if X is to be copied to Y, and in the next clock cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.



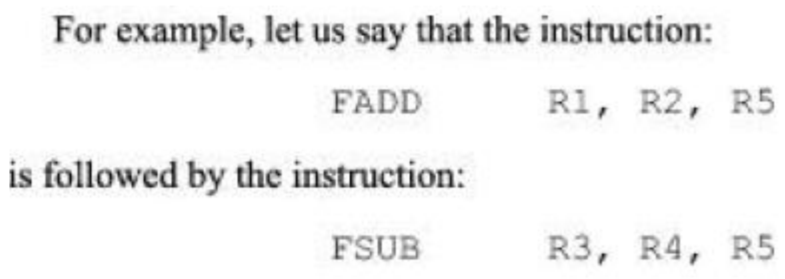
The above reasoning applies even if there is an intervening instruction between ADD and SHIFTR.

- Consider the following sequence of instructions:

- SHIFTER must be executed after ADD, in view of the RAW dependence.
- But there is no such dependence between SUB and any of the other two instructions, which means that SUB can be executed in program order or before ADD, or after SHIFTR.If SUB is executed in program order, then even without operand forwarding between ADD and SHIIFTR, no processor clock cycle is lost, since SHIFTER does not directly follow ADD. But now suppose SUB is executed either before ADD, or after SHIFTR. In both these cases, SHIFTR directly follows ADD, and therefore operand forwarding proves useful in saving a processor cycle, as we have seen above.

Register renaming

- Traditional compilers allocate registers to program variables in such a way as to reduce the main memory accesses required in the running program.Traditional compilers and assembly language programmers work with a fairly small number of programmable registers.Amongst the instructions in various stages of execution within the processor, there would be occurrences of RAW, WAR and WAW dependences on programmable registers
- RAW is true data dependence—since a value written by one instruction is used as an input operand by another. But a WAR or WAW dependence can be avoided if we have more registers to work with. We can simply remove such a dependence by getting the two instructions in question to use two different registers



For example, let us say that the instruction:

```
        FADD      R1, R2, R5
```

is followed by the instruction:

```
        FSUB      R3, R4, R5
```

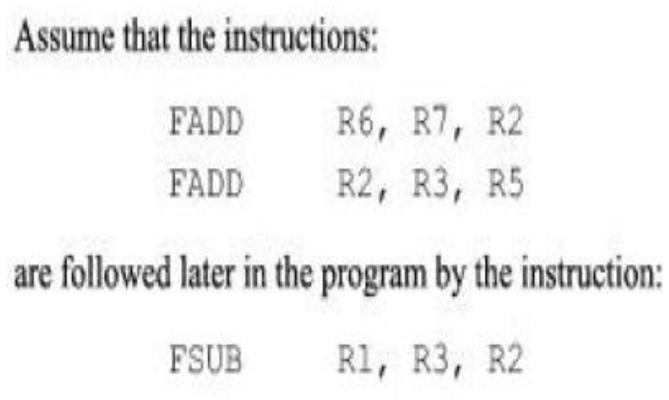
Both these instructions are writing to register R5, creating thereby a WAW dependence, i.e output dependence. Clearly, any subsequent instruction should read the value written into R5 by FSUB, and not the value written by FADD.

With additional registers available for use as these instructions execute, we have a simple technique to remove this output dependence. Let FSUB write its output value to a register other than R5, and let us call that other register X.

Then the instructions which use the value generated by FSUB will refer to X, while the instructions which use the value generated by FADD will continue to refer to R5. Now since FADD and FSUB are writing to two different registers, the output dependence or WAW between them has been removed

When FSUB commits, then the value in R5 should be updated by the value in X i.e. the value computed by FSUB. Then the physical register X, which is not a program visible register, can be freed up for use in another such situation.

- **Register renaming and WAR dependence**

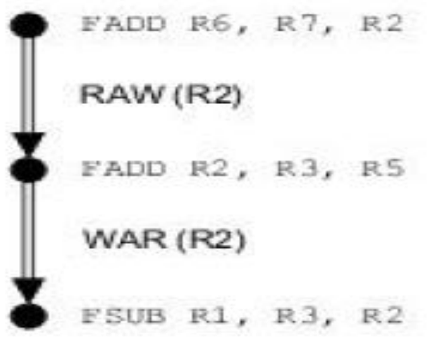


Assume that the instructions:

```
FADD    R6, R7, R2
FADD    R2, R3, R5
```

are followed later in the program by the instruction:

```
FSUB    R1, R3, R2
```

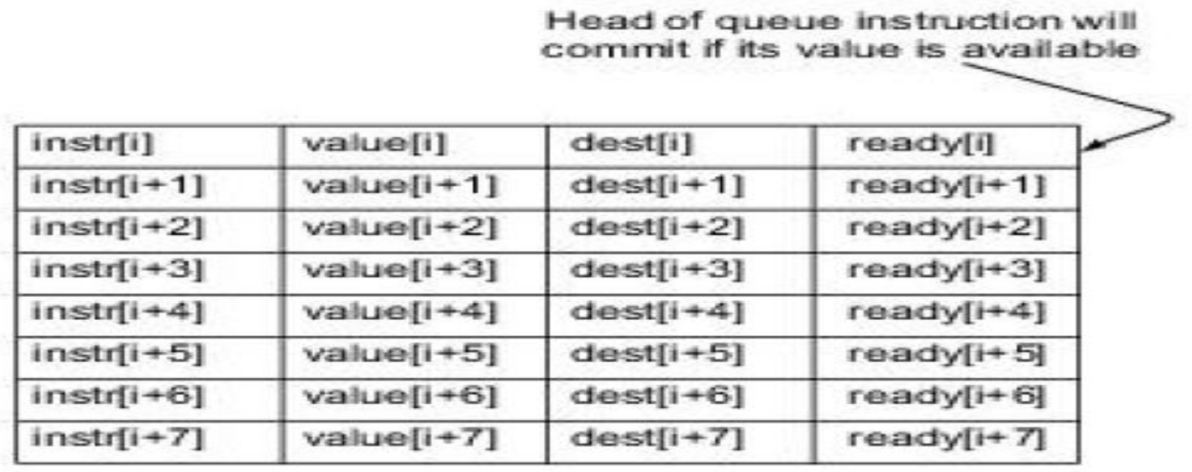


- With register renaming, it is a simple matter to resolve the WAR anti-dependence between the second FADD and FSUB.Let Xm be the program invisible register to which R2 has been mapped when the first FADD executes.
- This is then the remapped register to which the second FADD refers for its first data operand.Let FSUB write its output to a program invisible register other than Xm which we denote by Xn.Instructions which use the value written by FSUB refer to Xn while instructions which use the value written by the first FADD refer to Xm.
- When the first FADD commits, the value in Xm is transferred to R2 and program invisible register Xm is freed up; likewise, later when FSUB commits, the value in Xn is transferred to R2 and program invisible register Xn is freed up.
- Dependences are also caused by reads and writes to memory locations.
- In general, however, whether two instructions refer to the same memory location can only be known after the two effective addresses are calculated during execution.

- For example, the two memory references 2000[R1] and 4000[R3] occurring in a running program may or may not refer to the same memory location—this cannot be resolved at compile time.Resolution of whether two memory references point to the same memory location is known as alias analysis, which must be carried out on the basis of the two effective memory addresses. If a load and a store operation to memory refers to two different addresses, their order may be interchanged

Reorder buffer

- Since instructions execute in parallel on multiple functional units, the reorder buffer serves the function of bringing completed instructions back into an order which is consistent with program order. Note that instructions may execute in an order which is not related to program order, but must be committed in program order
- At any time, program state and processor state are defined in terms of instructions which have been committed i.e. their results are reflected in appropriate registers and/or memory locations. The concepts of program state and processor state are important in supporting context switches and in providing precise exceptions
- Entries in the reorder buffer are completed instructions, which are queued in program order. However, since instructions do not necessarily complete in program order, we also need a flag with each reorder buffer entry to indicate whether the instruction in that position has completed
- If the instruction at the head of the queue has not completed, and the reorder butter is full, then further issue of instructions is held up—i.e. the pipeline stalls—because there is no free space in the reorder buffer for one more entry.

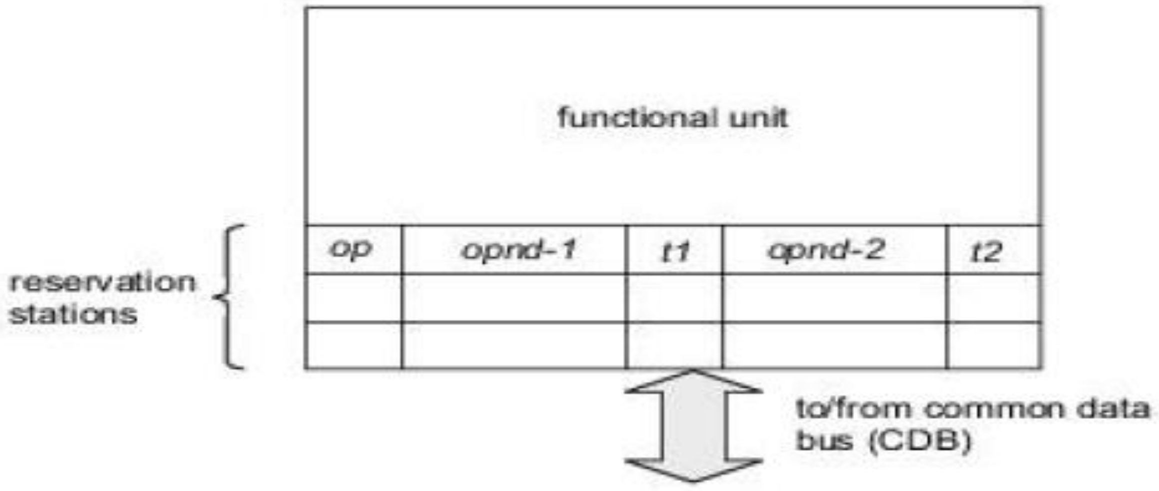

Head of queue instruction will commit if its value is available

instr[i]	value[i]	dest[i]	ready[i]
instr[i+1]	value[i+1]	dest[i+1]	ready[i+1]
instr[i+2]	value[i+2]	dest[i+2]	ready[i+2]
instr[i+3]	value[i+3]	dest[i+3]	ready[i+3]
instr[i+4]	value[i+4]	dest[i+4]	ready[i+4]
instr[i+5]	value[i+5]	dest[i+5]	ready[i+5]
instr[i+6]	value[i+6]	dest[i+6]	ready[i+6]
instr[i+7]	value[i+7]	dest[i+7]	ready[i+7]

Tomasulo's algorithm

The algorithm was based on operand forwarding over a common data bus, with tags to identity sources of data values sent over the bus. Register renaming was also an part of the algorithm. For register renaming, we need a set of program invisible registers to which programmable registers are mapped.Tomasulo's algorithm requires these program invisible registers to be provided with reservation stations of functional units.

- Let us assume that the functional units are internally pipelined and can complete one operation in every clock cycle.
- Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values



The various fields making up a typical reservation station are as follows:

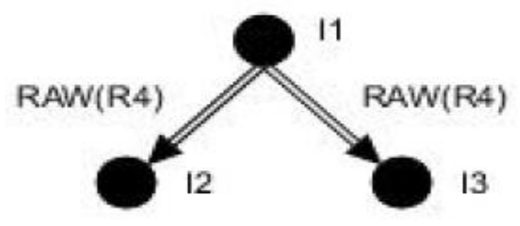
op	operation to be carried out by the functional unit
opnd-1 & *opnd-2*	two operand values needed for the operation
t1 & *t2*	two source tags associated with the operands

- When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle. At the time of instruction issue, the reservation station is filled out with the operation code {op}.

- If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station
- However, if the operand value is not available at the time of issue, the corresponding source tag (t1 and/or t2) is copied into the reservation station. The source tag identifies the source of the required operand. As soon as the required operand value is available at its source—which would be typically the output of a functional unit—the data value is forwarded over the common data bus, along with the source tag.
- This value is copied into all the reservation station operand slots which have the matching tag.

Tomasulo's algorithm and RAW dependence

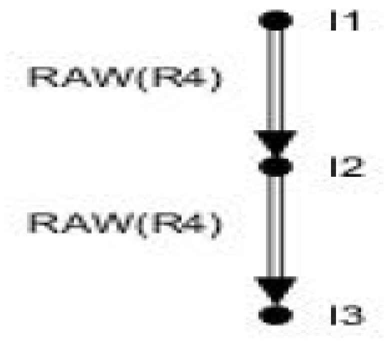
- Assume that instruction I1 is to write its result into R4, and that two subsequent instructions I2 and I3 are to read—i.e. make use of—that result value.
- Thus instructions I2 and I3 are truly data dependent (RAW dependent) on instruction I1



Assume that the value in R4 is not available when I2 and I3 are issued; the reason could be, for example, that one of the operands needed for I1 is itself not available

- When I2 and I3 are issued, they are parked in the reservation stations of the appropriate functional units
- Since the required result value from I1 is not available, these reservation station entries of I2 and I3 get source tag corresponding to the output of I1 —i.e. output of the functional unit which is performing the operation of I1.
- When the result of I1 becomes available at its functional unit, it is sent over the common data bus along with the tag value of its source—i.e. output of functional unit.

Combination of RAW and WAR dependence



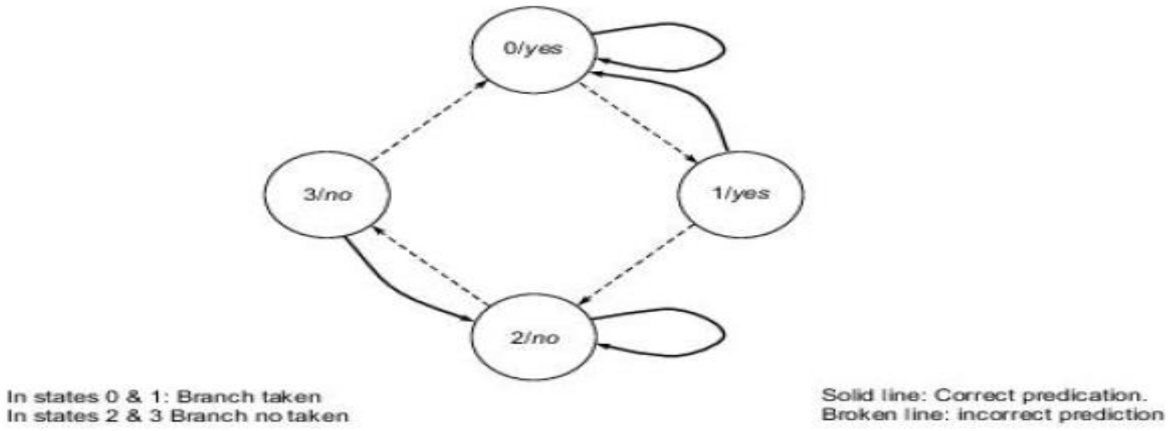
- The question new is: Can I3 be issued even before I1 completes and I2 starts execution?

BRANCH PREDICTION

About 15% to 20% of instructions in a typical program are branch and jump instructions, including procedure returns. Therefore—if hardware resources are to be fully utilized in a superscalar processor—the processor must start working on instructions beyond a branch, even before the branch instruction itself has completed. This is only possible through some form of branch prediction

Two-bit predictor

- A two-bit counter is maintained for every conditional branch instruction in the program.
- The two-bit counter has four possible states; these four states and the possible transitions between these states are shown below


0/yes
1/yes
2/no
3/no
In states 0 & 1: Branch taken
In states 2 & 3 Branch no taken
Solid line: Correct predication.
Broken line: incorrect prediction

- To be effective, branch prediction should be carried out as early as possible in the instruction pipeline.As soon as a conditional branch instruction is decoded, branch prediction logic should predict whether the branch is taken.
- Accordingly, the next instruction address should be taken either as the branch target address (i.e. branch is taken), or the sequentially next address in the program (i.e. branch is not taken).
- Can branch prediction be carried out even before the instruction is decoded i.e. at the instruction fetch stage? Yes, if a so-called branch-target-buffer is provided which has a history of recently executed conditional branches.In some programs, whether a conditional branch is taken or not taken correlates better with other conditional branches in the program—rather than with the earlier history of outcomes of the same conditional branch.
- Accordingly, correlated predictors can be designed, which generate a branch prediction based on whether other conditional branches in the program were taken or not taken.

Branch prediction based on the earlier history of the same branch is known as local prediction, while prediction based on the history of other branches in the program is known as global prediction

- A tournament predictor uses
- A global predictor
- A local predictor
- A selector which selects one of the two predictors for prediction at a given branch instruction

Speculative Execution

Instructions executed on the basis of a predicted branch, before the actual branch result is known, are said to involve speculative execution.

- If a branch prediction turns out to be correct, the corresponding speculatively executed instructions must be committed.
- If the prediction turns out to be wrong, the effects of corresponding speculative operations carried out within the processor must be cleaned up, and instructions from another branch of the program must instead be executed