UNIT 4

Loop Transformations

The compiler performs several loop restructuring transformations to help improve the parallelization of a loop in programs. Some of these transformations can also improve the single processor execution of loops as well. The transformations performed by the compiler are described below.

Loop Distribution

Often,loops contain a few statements that cannot be executed in parallel and many statements that can be executed in parallel. Loop Distribution attempts to remove the sequential statements in a separate loop and gather the parallelizable statements in a different loop. This is illustrated in the following example:

Example Candidate for Loop Distribution

```
for (i=0; i < n; i++) {
   x[i] = y[i] + z[i]*w[i];              /* S1 */
   a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;  /* S2 */
   y[i] = z[i] - x[i];                   /* S3 */
}
```

Assuming that arrays x, y, w, a, and z do not overlap, statements S1 and S3 can be parallelized but statement S2 cannot be. Here is how the loop looks after it is split or distributed into two different loops:

Example The Distributed Loop

```
/* L1: parallel loop */
for (i=0; i < n; i++) {
   x[i] = y[i] + z[i]*w[i];             /* S1 */
   y[i] = z[i] - x[i];                 /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++) {
   a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

After this transformation, loop L1 does not contain any statements that prevent the parallelization of the loop and may be executed in parallel. Loop L2, however, still has a non-parallelizable statement of the original loop.

Loop distribution is not always profitable or safe to perform. The compiler performs analysis to determine the safety and profitability of distribution.

Loop Fusion

If the granularity of a loop, or the work performed by a loop, is small, the performance gain from distribution may be insignificant. This is because the overhead of parallel loop start-up is too high compared to the loop workload. In such situations, the compiler uses loop fusion to combine several loops into a single parallel loop, and thus increase the granularity of the loop. Loop fusion is easy and safe when loops with identical trip counts are adjacent to each other. Consider the following example:

Example Loops With Small Work Loads

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
   a[i] = a[i] + b[i];        /* S1 */
}
/* L2: another short parallel loop */
for (i=0; i < 100; i++) {
   b[i] = a[i] * d[i];        /* S2 */
}
```

The two short parallel loops are next to each other, and can be safely combined as follows:

Example The Two Loops Fused

```
/* L3: a larger parallel loop */
for (i=0; i < 100; i++) {
   a[i] = a[i] + b[i];       /* S1 */
   b[i] = a[i] * d[i];       /* S2 */
}
```

The new loop generates half the parallel loop execution overhead. Loop fusion can also help in other ways. For example, if the same data is referenced in two loops, then combining them can improve the locality of reference.

However, loop fusion is not always safe to perform. If loop fusion creates a data dependence that did not exist before then the fusion may result in incorrect execution. Consider the following example:

Example Unsafe Fusion Candidates

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
   a[i] = a[i] + b[i];      /* S1 */
}
/* L2: a short loop with data dependence */
for (i=0; i < 100; i++) {
   a[i+1] = a[i] * d[i];    /* S2 */
}
```

If the loops in 1.2 Loop fusion are fused, a data dependence is created from statement S2 to S1. In effect, the value of a[i] in the right hand side of statement S1 is computed in statement S2. If the loops are not fused, this would not happen. The compiler performs safety and profitability analysis to determine if loop fusion should be done. Often, the compiler can fuse an arbitrary number of loops. Increasing the granularity in this manner can sometimes push a loop far enough up for it to be profitable for parallelization.

Loop Interchange

It is generally more profitable to parallelize the outermost loop in a nest of loops, since the overheads incurred are small. However, it is not always safe to parallelize the outermost loops due to dependences that might be carried by such loops. This is illustrated in the following:

Example Nested Loop That Cannot Be Parallelized

```
for (i=0; i <n; i++) {
   for (j=0; j <n; j++) {
         a[j][i+1] = 2.0*a[j][i-1];
   }
}
```

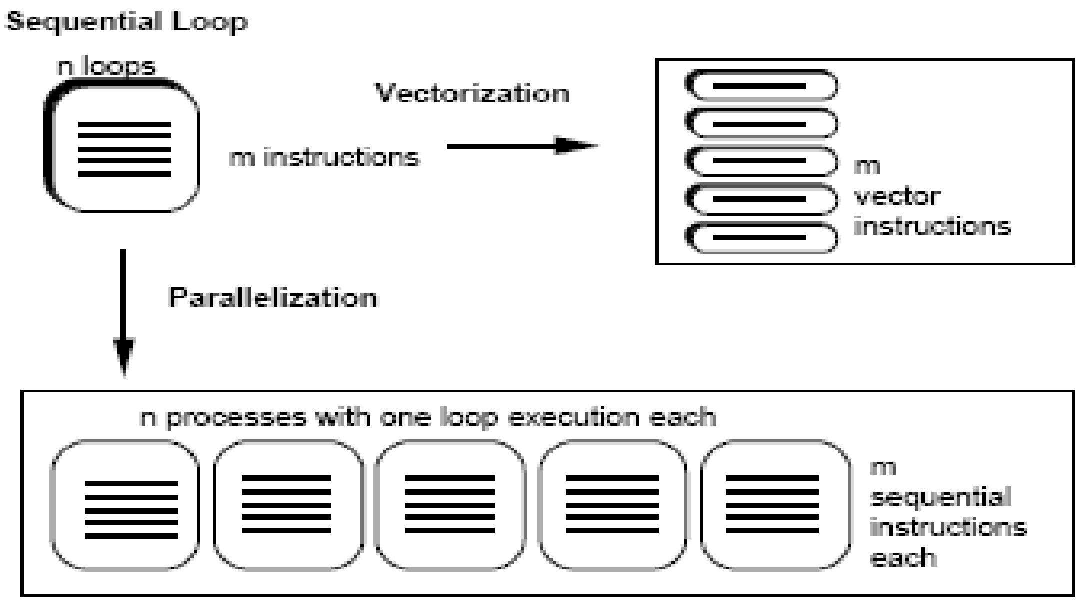
In this example, the loop with the index variable i cannot be parallelized, because of a dependency between two successive iterations of the loop. The two loops can be interchanged and the parallel loop (the j-loop) becomes the outer loop:

Example The Loops Interchanged

```
for (j=0; j<n; j++) {
   for (i=0; i<n; i++) {
         a[j][i+1] = 2.0*a[j][i-1];
   }
}
```

The resulting loop incurs an overhead of parallel work distribution only once, while previously, the overhead was incurred n times. The compiler performs safety and profitability analysis to determine whether to perform the loop interchange.

<u>Loop parallelization</u>



- Exploiting multi-processors
- Allocate individual loop iterations to different processors
- Additional synchronization is required depending on data dependences

  Example:

```
for i:=1 to n do
S1: A[i]:= C[i];
S2: B[i]:= A[i];
end;
```

- Data dependency: S1 δ(=) S2 (due to A[i])
- Synchronization required: NO

```
doacross i:=1 to n do
S1: A[i]:= C[i];
```

```
S2: B[i]:= A[i];
enddoacross;
```

- ☐ The inner loop is to be parallelized:

```
for i:=1 to n do
    for j:=1 to m do
    S1: A[i,j]:= C[i,j];
    S2: B[i,j]:= A[i-1,j-1];
    end;
end;
```

- ☐ Data dependency: S1 δ(<,<) S2 (due to A[i,j])
- ☐ Synchronization required: NO

```
for i:=1 to n do
doacross j:=1 to m do
S1: A[i,j]:= C[i,j];
S2: B[i,j]:= A[i-1,j-1];
enddoacross;
end;
```

```
for i:= 1 to n do
S1: A[i] := B[i] + C[i];
S2: D[i] := A[i] + E[i-1];
S3: E[i] := E[i] + 2 * B[i];
S4: F[i] := E[i] + 1;
end;
```

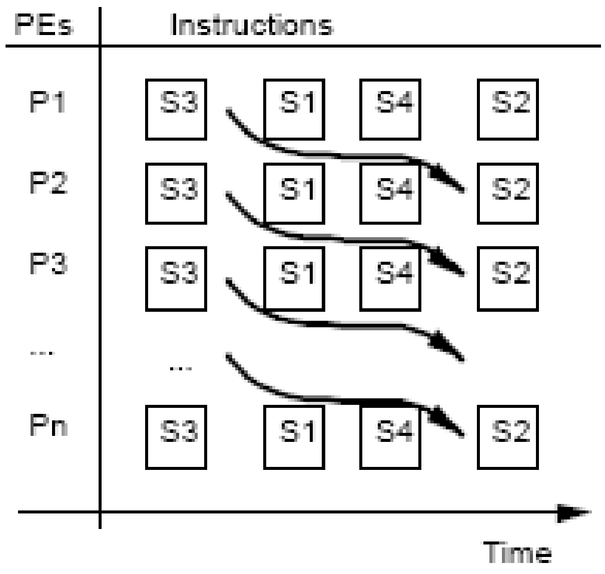
- ☐ Data Dependences:
    - ■ S1 δ(=) S2 (due to A[i]) ← no synch. required

- S3 δ(=) S4 (due to E[i]) ← no synch. required
- S3 δ(<) S2 (due to E[i]) ← synch. Required

- After re-ordering and adding sync code

```
var sync: array [1..n] of semaphore[0];

doacross i:=1 to n do
S3:   e[i] := e[i] + 2 * b[i];
      V(sync[i]);
S1:   a[i] := b[i] + c[i];
S4:   f[i] := e[i] + 1;
      if i>1 then P(sync[i-1]) end;
S2:   d[i] := a[i] + e[i-1]
enddoacross;
```



Data Parallel Model

- Programming Model
    - Operations are performed on each element of a large (regular) data structure (array, vector, matrix)
    - Program is logically a single thread of control, carrying out a sequence of either sequential or parallel steps
    - The Simple Problem Strikes Back

A = (A + B) * C

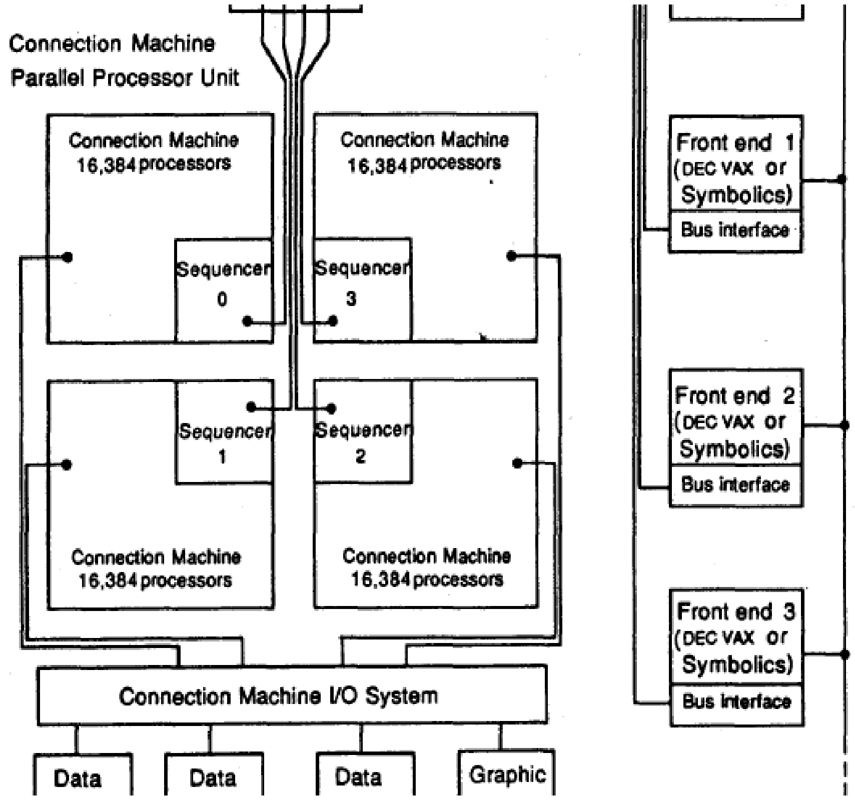
sum = global_sum (A)

- Language supports array assignment
- Early architectures directly mirrored programming model
- Single control processor (broadcast each instruction to an array/grid of processing elements)
    - Consolidates control
- Many processing elements controlled by the master
- Examples: Connection Machine, MPP
    - Batcher, "Architecture of a massively parallel processor," ISCA 1980.
        - 16K bit-serial processing elements
    - Tucker and Robertson, "Architecture and Applications of the Connection Machine," IEEE Computer 1988.

64K bit-serial processing elements

- Later data parallel architectures
    - Higher integration → SIMD units on chip along with caches
    - More generic → multiple cooperating multiprocessors with vector units
    - Specialized hardware support for global synchronization
        - E.g. barrier synchronization
        - Example: Connection Machine 5

- ❑ Hillis and Tucker, "The CM-5 Connection Machine: a scalable supercomputer," CACM 1993.
- ❑ Consists of 32-bit SPARC processors
- ❑ Supports Message Passing and Data Parallel models
- ❑ Special control network for global synchronization



Connection Machine Parallel Processor Unit
Connection Machine 16,384 processors
Sequencer 0
Connection Machine 16,384 processors
Sequencer 3
Sequencer 1
Sequencer 2
Connection Machine 16,384 processors
Connection Machine 16,384 processors
Connection Machine I/O System
Data
Data
Data
Graphic
Front end 1 (DEC VAX or Symbolics)
Bus interface
Front end 2 (DEC VAX or Symbolics)
Bus interface
Front end 3 (DEC VAX or Symbolics)
Bus interface

Connection Machine CM5

- ■ Shared Memory
  - ❑ Single shared address space
  - ❑ Communicate, synchronize using load / store
  - ❑ Can support message passing

- Message Passing
    - Send / Receive
    - Communication + synchronization
    - Can support shared memory
- Data Parallel
    - Lock-step execution on regular data structures
    - Often requires global operations (sum, max, min...)
    - Can be supported on either SM or MP

Data Flow Models

- A program consists of data flow nodes
- A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens
- No artificial constraints, like sequencing instructions
- How do we know when operands are ready?
    - Matching store for operands (remember OoO execution?)
    - large associative search
    - Later machines moved to coarser grained dataflow (threads + dataflow across threads)
    - allowed registers and cache for local computation
    - introduced messages (with operations and operands)

Dependence:

A **data dependency** in computer science is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

There are three types of dependencies: data, name, and control.

Three cases exist:

- Flow (data) dependence: O(S1) ∩ I (S2), S1 → S2 and S1 writes after something read by S2
- Anti-dependence: I(S1) ∩ O(S2), S1 → S2 and S1 reads something before S2 overwrites it
- Output dependence: O(S1) ∩ O(S2), S1 → S2 and both write the same memory location.

A Flow dependency, also known as a data dependency or true dependency or read-after-write (RAW), occurs when an instruction depends on the result of a previous instruction:

An anti-dependency, also known as write-after-read (WAR), occurs when an instruction requires a value that is later updated.

An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable.

Goal: Identify loops whose iterations can be executed in parallel on different processors of a shared-memory multiprocessor system.

Matrix Multiplication

```
for I = 1 to n do -- parallel
 for J = 1 to n do -- parallel
for K = 1 to n do –- not parallel
  C[I,J] = C[I,J] + A[I,K]*B[K,J]
```

Flow Dependence:

S1: X = …. S1 δ^f S2

S1 δ^a S2

S1 δ^o S2

S2: … = X

Anti Dependence:

```
    S1: … = X
    S2: X = …
```

Output Dependence:

```
    S1: X = …
    S2: X = …
```

Example data dependences:

```
do I = 1, 40
    S1: A(I+1) = ….
    S2: … = A(I-1)
enddo
do I = 1, 40
    S1: A(I-1) = …
    S2: … = A(I+1)
enddo
do I = 1, 40
    S1: A(I+1) = …
    S2: A(I-1) = …
enddo
```

Data Profiling

The use of analytical techniques about data for the purpose of developing a thorough knowledge of its content, structure and quality.

- Data profiling is the process of examining the data available in an existing data source (e.g. a database or a file) and collecting statistics and information about that data. The purpose of these statistics may be to:

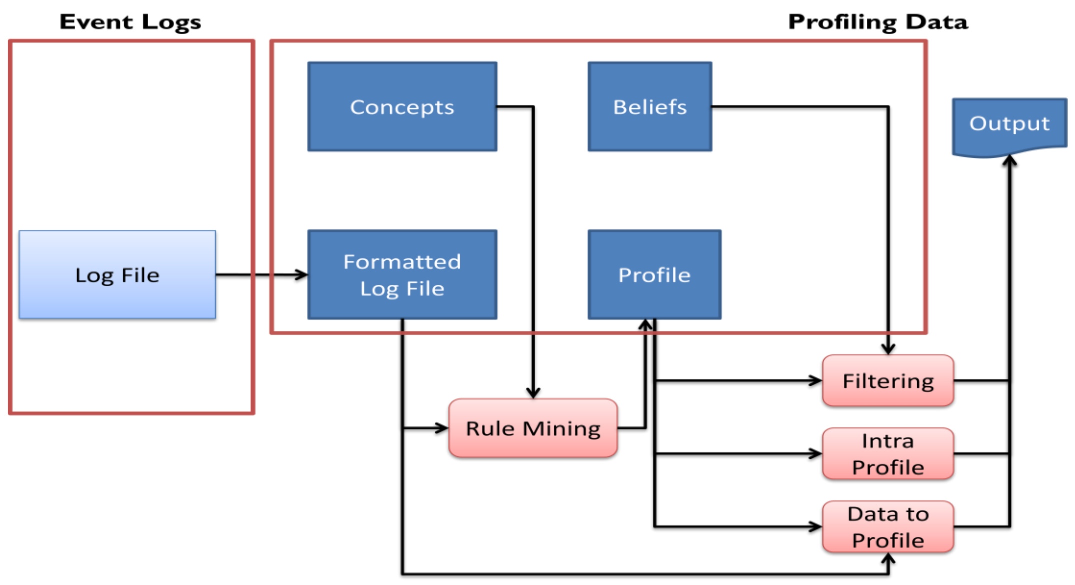
- find out whether existing data can easily be used for other purposes
- give metrics on data quality including whether the data conforms to company standards
- assess the risk involved in integrating data for new applications, including the challenges of joins
- track data quality
- assess whether metadata accurately describes the actual values in the source database
- understanding data challenges early in any data intensive project, so that late project surprises are avoided. Finding data problems late in the project can incur time delays and project cost overruns.
- have an enterprise view of all data, for uses such as Master Data Management where key data is needed, or Data governance for improving data quality

Process Profiling

- The practice of tracking information about processes by monitoring their execution. This can be done by analyzing the case perspective, process perspective and resource perspective, to assess their behavior, predict certain characteristics and to configure optimum runtime parameters.

  Applications

- Analyzing rendering behavior. A user could be provided with a set of options that allow one to analyze very specific rendering behavior in parts of a process.
- Profiling process outcomes- the use of some techniques to analyze the outcome of processes in order to determine what may be causing the observed behavior.
- Event Tracing and Prediction. Based on an event log, real-time event logs can be traced to troubleshoot, determine where the performance issues are occurring and predict the likely execution pattern.



DFD for Profiling Process

Where is Data Profiling used?

- **Enterprise Data Quality Improvement Program**
  - Traditional Six Sigma Like Program
  - Recursive application of data quality assessment
  - Based on the historical success of companies who have used it
- **Support Consolidation of Databases after mergers and acquisitions**
  - Dramatically reduce cost and time to complete projects
  - Improve quality of data in resulting system
  - **Support application renovation projects**
  - Dramatically reduce time and cost to complete
  - Improve quality of data in resulting system
- **Support data integration functions for data warehousing/ business intelligence data stores**
  - Develop processes to cleanse data in transit
  - Improve quality of data in information intelligence stores

Scheduling of parallel programs

Parallel Scheduling Categories

- ☐ Job Scheduling
  - ■ A set of jobs arriving at a parallel system
  - ■ Choosing an order of jobs for execution to minimize total turnaround time
- ☐ Application Scheduling
  - ■ Mapping a single application's tasks, to resources to reduce the total response time
  - ■ In general, difficult to achieve for communication-intensive applications
  - ■ For applications with independent tasks (pleasingly parallel applications), some methods have been proposed

Job Scheduling

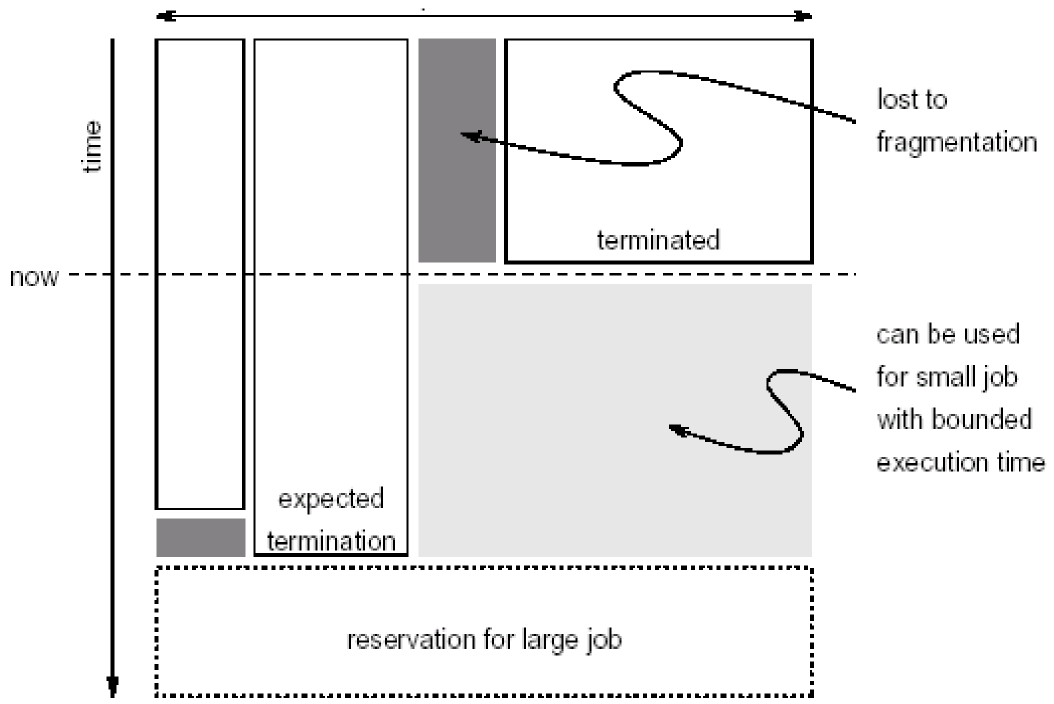
- ☐ A parallel job is mapped to a subset of processors
- ☐ The set of processors dedicated to a certain job is called a partition of the machine
- ☐ To increase utilization, parallel machines are typically partitioned into several non-overlapping partitions, allocated to different jobs running concurrently – space slicing or space partitioning
- ☐ Users submit their jobs to a machine's scheduler
- ☐ Jobs are queued
- ☐ Jobs in queue considered for the allocation whenever the state of a machine changes (submission of a new job, exit of a running job)
- ☐ Allocation – which job in the queue?, which machine?
- ☐ Packing jobs to the processors
- ☐ Goal – to increase processor utilization
- ☐ Lack of knowledge of future jobs and job execution times. Hence simple heuristics to perform packing at each scheduled event

Scheduling policies

- ☐ FCFS
- ☐ If the machine's free capacity cannot accommodate the first job, it will not attempt to start any subsequent job
- ☐ No starvation; But poor utilization
- ☐ Processing power is wasted if the first job cannot run

Backfilling

- ☐ Allows small jobs from the back of the queue to execute before larger jobs that arrived earlier
- ☐ Requires job runtimes to be known in advance – often specified as runtime upper-bound


time
now
lost to fragmentation
terminated
can be used for small job with bounded execution time
expected termination
reservation for large job

- ☐ Identifies holes in the 2D chart and moves smaller jobs to fill those holes
- ☐ 2 types – conservative and aggressive (EASY)

EASY backfilling

- ☐ Aggressive version of backfilling

- ☐ Any job can be backfilled provided it does not delay the first job in the queue
- ☐ Starvation cannot occur for the first job since queuing delay for the first job depends only on the running jobs
- ☐ But jobs other than the first may be repeatedly delayed by newly arriving jobs

Conservative backfilling

- ☐ Makes reservations for all queued jobs
- ☐ Backfilling is done subject to checking that it does not delay any previous job in the queue

Starvation cannot occur at all

LOS (Look ahead Optimized Scheduler)

- ☐ Examines all jobs in the queue to maximize utilization
- ☐ Instead of scanning the queue in any order and starting any job that is small enough not to violate prior reservations
- ☐ LOS tries to find combination of jobs
- ☐ Using dynamic programming
- ☐ Results in local optimum; not global optimum
- ☐ Global optimum may leave processors idle in anticipation of future arrivals

What is important in a scheduling algorithm?

- Minimize Response Time
  - Elapsed time to do an operation (job)
  - Response time is what the user sees
    - Time to echo keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Jobs per second

  - Throughput related to response time, but not identical
    - Minimizing response time will lead to more context switching than if you maximized only throughput
  - Minimize overhead (context switch time) as well as efficient use of resources (CPU, disk, memory, etc.)
- Fairness
  - Share CPU among users in some equitable way
  - Not just minimizing average response time

<u>Optimal Scheduling Algorithms</u>

- FCFS scheduling, FIFO Run Until Done:
  - Simple, but short jobs get stuck behind long ones
- RR scheduling:
  - Give each thread a small amount of CPU time when it executes, and cycle between all ready threads
  - Better for short jobs, but poor when jobs are the same length
- SJF/SRTF:
  - Run whatever job has the least amount of computation to do / least amount of remaining computation to do
  - Optimal (average response time), but unfair; hard to predict the future
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a number of tickets (short tasks get more)
  - Every thread gets tickets to ensure forward progress / fairness
- Priority Scheduing:

- Preemptive or Nonpreemptive
- Priority Inversion

Data path design: Parallel and Pipeline design

PIPELINING OF FIR DIGITAL FILTERS 65

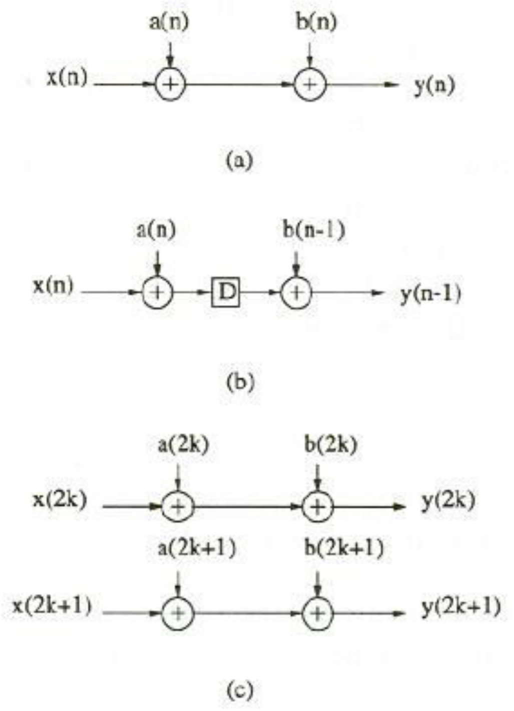


Fig. 3.2 (a) A datapath. (b) The 2-level pipelined structure of (a). (c) The 2-level parallel processing structure of (a).

Parallelizing Serial Programs

Parallelizing Compiler

- Most common type of tool used to automatically parallelize a serial program into parallel programs
- Parallelizing Compiler works in 2 different ways:
  - Fully Automatic
  - Programmer Directed
- The compiler analyzes the source code and identifies opportunities for parallelism

- The analysis includes:
  - Identifying inhibitors to parallelism
  - Possibly a cost weighting on whether or not the parallelism would actually improve performance
  - Loops (do, for) loops are the most frequent target for automatic parallelization
- Using "compiler directives" or possibly compiler flags, the programmer explicitly tell the compiler how to parallelize the code
- May be able to be used in conjunction with some degree of automatic parallelization also

Automatic Parallelization

- Wrong results may be produced
- Performance may actually degrade
- Much less flexible than manual parallelization
- Limited to a subset (mostly loops) of code
- May actually not parallelize code if the analysis suggest there are inhibitors or the code is too complex

<u>Parallelization Techniques</u>

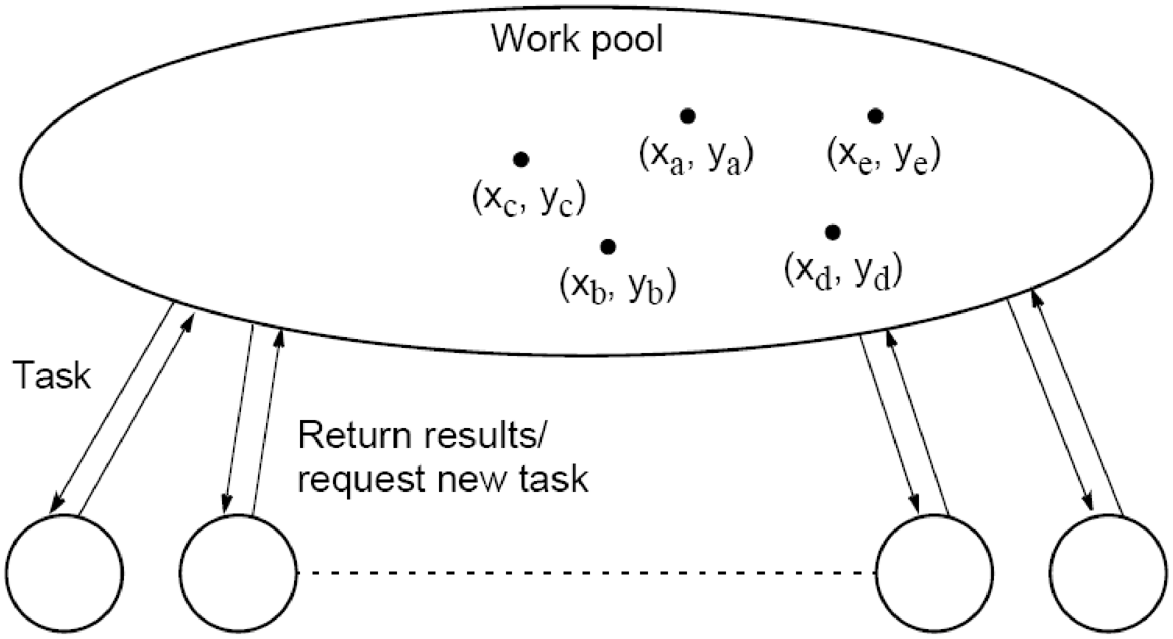
Parallelizing Mandelbrot Set Computation

Static Task Assignment

Simply divide the region into fixed number of parts, each computed by a separate processor.

<u>Disadvantage</u>:

Different regions may require different numbers of iterations and time.



Dynamic Task Assignment

Monte Carlo Methods

- Another embarrassingly parallel computation

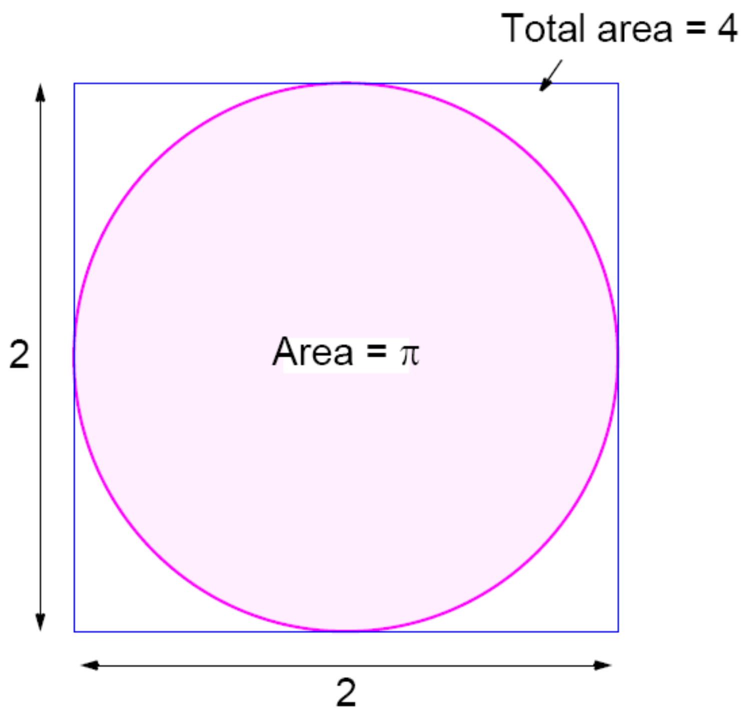
<u>Example</u>:

calculate **π** using the ratio:

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi . r^2}{2 \times 2} = \frac{\pi}{4}$$

- Randomly choose points within the *square*

- Count the points that lie within the *circle*
- Given a sufficient number of randomly selected samples fraction of points within the circle will be: $\pi/4$



A Parallel Formulation of Random Number Generation

$x_{i+1} = (ax_i + c)$ ***mod*** m

$x_{i+k} = (Ax_i + C)$ ***mod*** m

A and C above can be derived by computing

$$x_{i+1}=f(x_i)\ ,\quad x_{i+2}=f(f(x_i)),\quad \ldots\quad x_{i+k}=f(f(f(\ldots f(x_i))))$$

and using the following properties:

- (A+B) ***mod*** M = [(A ***mod*** M) + (B ***mod*** M)] ***mod*** M
- [X(A ***mod*** M)] ***mod*** M = (X.A ***mod*** M)
- X(A + B) ***mod*** M = (X.A + X.B) ***mod*** M

= [(X.A ***mod*** M) + (X.B ***mod*** M)] ***mod*** M

- [X((A+B) ***mod*** M)] ***mod*** M = (X.A + X.B) ***mod*** M

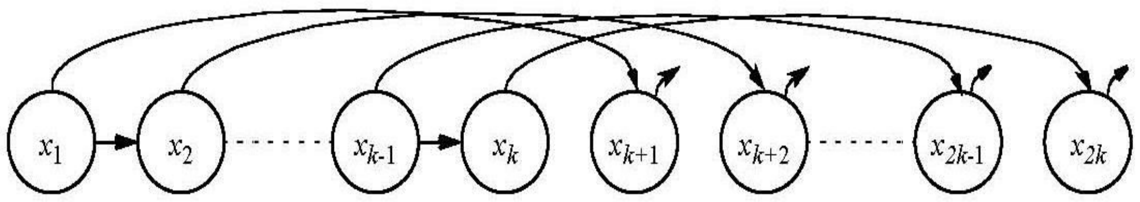
Parallel Random Number Generation

It turns out that

$$x_{i+1} = (ax_i + c) \bmod m$$

$$x_{i+k} = (Ax_i + C) \bmod m$$

where $A = a^k \bmod m$, $C = c(a^{k-1} + a^{n-2} + \ldots + a^1 + a^0) \bmod m$, and k is a selected "jump" constant.



Instruction Level Parallelism

► ILP is a measure of the number of instructions that can be performed during a single clock cycle.

Parallel instructions are a set of instructions that do not depend on each other to be executed.

- ► Hierarchy
  - ▪ Bit level Parallelism
    - ► 16 bit add on 8 bit processor
  - ▪ Instruction level Parallelism
  - ▪ Loop level Parallelism

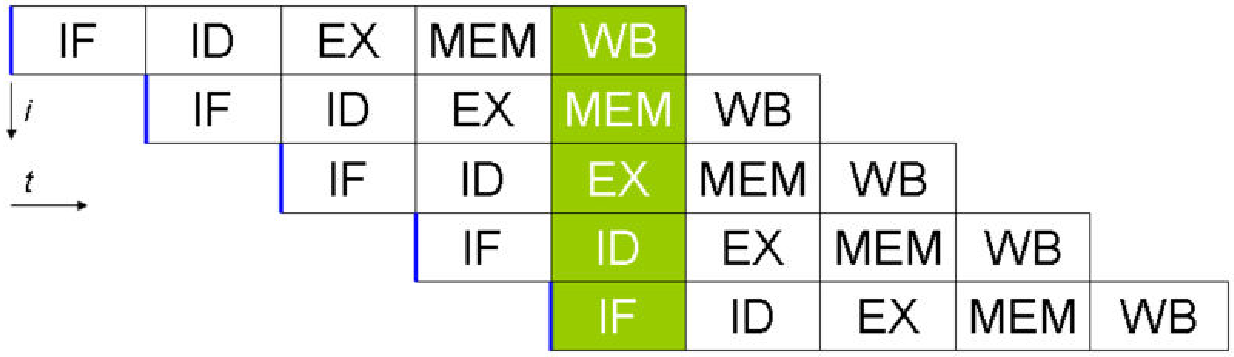
- for (i=1; i<=1000; i= i+1)
  x[i] = x[i] + y[i];

- Thread level Parallelism (SMT, multi-core computers)

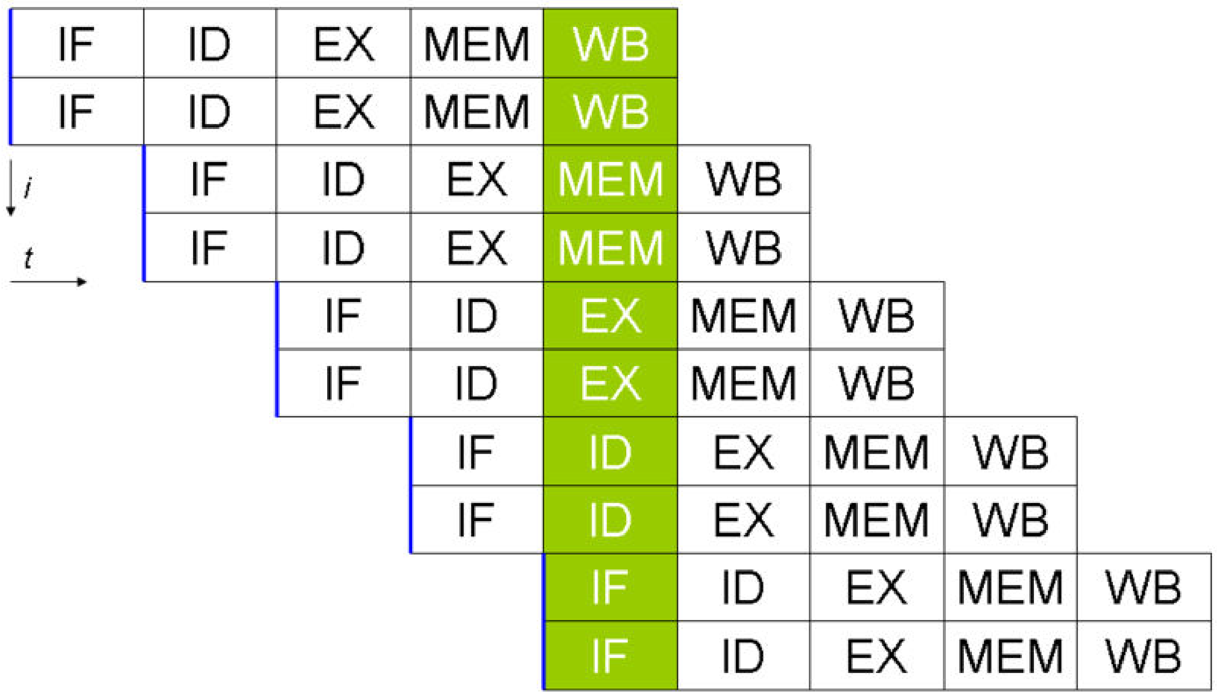
Implementations of ILP

- Pipelining
- Superscalar Architecture
  - Dependency checking on chip.
  - Multiple Processing Elements eg. ALU, Shift
- VLIW (Very Long Instruction Word Architecture)
  - Simple hardware, Complex Compiler
- Multi processor computers

Pipelining



Superscalar



IF - Instruction Fetch

ID - Instruction Decode

EX – Execute

MEM – Store in Memory

WB – Write Back

- Identifying parallel instructions
- Hardware Techniques
  - Out of order execution
    - Window Size
  - Speculative execution
    - Branch Prediction
    - Branch Fanout
- Compiler Techniques
  - Register Renaming

```
ADD $t0,$s1,$2
SW $t0, 0($s3)
ADD $t0,s$4,$s5
SW $t0, 0($s6)
```

- Unrolling loops
  - ► Takes advantage of loop level parallelism

<u>Data Dependency Test</u>

DependenceTests: extreme value test; GCD test; Generalized GCD test; Lambda test; Delta test; Power test; Omega test etc…

Example: (Extreme Value Test)

```
DO I = 1, 10
    DO J = 1, 10
        A[10*I+J-5] = ….A[10*I+J-10]….
```

10*I1+J1-5 = 10*I2+J2-10

10*I1-10*I2+J1-J2 = -5

f: $R^4 \rightarrow R$; f(I1,I2,J1,J2) = 10*I1-10*I2+J1-J2

1<=I1,I2,J1,J2<=10;

lower bound, b=-99; upper bound, B=+99

since -99 <= -5 <= +99 there is a dependence

GCD Test

It is difficult to analyze array references in compile time to determine data dependency (whether they point to same address or not). A simple and sufficient test for the absence of a dependence is the greatest common divisor (GCD) test. It is based on the observation that if a loop carried dependency exists between X[a*i + b] and X[c*i + d] (where X is the array; a, b, c and d are integers, and i is the loop variable), then GCD (c, a) must divide (d – b). The assumption is that the loop must be normalized – written so that the loop index/variable starts at 1 and gets incremented by 1 in every iteration. For example, in the following loop, a=2, b=3, c=2, d=0 and GCD(a,c)=2 and (d-b) is -3. Since 2 does not divide -3, no dependence is possible.

```
for(i=1; i<=100; i++)
{
        X[2*i+3] = X[2*i] + 50;
}
```