EIGHT SEMESTER EXAMINATION, 2007-2008

ADVANCE COMPUTER ARCHITECTURE

Time – 3 hours Total Marks – 100

Note: (1) Attempt all questions.

(2) All questions carry equal marks.

(3) Be precise in your answer.

(4) No second answer book will be provided.

Q. 1. Attempt any two parts of the following:-

Q. (a). Explain how instruction set, compiler technology, CPU implementation and control and memory hierarchy affect the CPU performance and justify the effects in terms of program length, clock rate and effective CPI.

Ans. CPU Performance: Most CPUs, and indeed most sequential logic devices, are synchronous in nature. That is, they are designed and operate on assumptions about a synchronization signal. This signal, known as a clock signal, usually takes the form of a periodic square wave. By calculating the maximum time that electrical signals can move in various branches of a CPU's many circuits, the designers can select an appropriate period for the clock signal.

This period must be longer than the amount of time it takes for a signal to move, or propagate, in the worst-case scenario. In setting the clock period to a value well above the worst-case propagation delay, it is possible to design the entire CPU and the way it moves data around the "edges" of the rising and falling clock signal. This has the advantage of simplifying the CPU significantly, both from a design perspective and a component-count perspective. However, it also carries the disadvantage that the entire CPU must wait on its slowest elements, even though some portions of it are much faster. This limitation has largely been compensated for by various methods of increasing CPU parallelism.

Q. 1(b). Distinguish between the following:

(i). Medium-grain and fine-grain multicomputers in their architectures and programming requirements.

(ii). Single threaded and multi-threaded processor architectures.

Ans.(i). Medium grain multicomputers: Vs fine grain multicomputers:

- Medium-grain multicomputers; Mbytes of memory per node
- Fine-grain multicomputers; tens of Kbytes of memory per node

An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly

parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

(ii). Single threaded and Multithreaded processor architecture:

Advantages to employing barrel processors ver single-tasking processors include:

- The ability to do useful work on the other threads while the stalléd thread is waiting.
- Designing an n-way barrel processor with n-deep pipelines is much simpler than designing a single-tasking processor because a barrel processor never has a pipeline stall and doesn't need feed-forward circuits.
- For real-time applications, a Barrel processor can guarantee that a "real-time" thread can execute with precise timing, no matter what happens to the other threads – even if some other thread locks up in an infinite loop or is continuously interrupted by hardware interrupts.

By maintaining the multiple threads in the hardware, and switching among them, the multithreaded processor can overlap the waiting time for the synchronization delay so that it decreases the processor idle time. This correspondence describes the multithreaded processor architecture, in which there are a number of hardware contexts per processor. It uses coarsegrain method to schedule threads and two-phase waiting algorithm to synchronize the waiting threads.

Q. 1(c). Consider a shared bus parallel computer built using 32 bit RISC processors running at 150 MHz with CPI-1. Assume that 15% of the instructions are loads and 10% are stores. Assume 0.95 hit ratio to cache for read and write through caches. The bandwidth of the bus is 2GB/sec.

(i). How many processors can the bus support without getting saturated?

(ii). If caches are not there, how many processors can the bus support assuming the main memory is as far as the cache?

Ans.

Q. 2. Attempt any two parts of the following:

Q.(a). Discuss data and resource dependences to exploit implicit parallelism in the program. Analyze the following instructions sequence:

S1 : Load R_1, 1024 / $R_1 \leftarrow$ 10241 /

S2 : Load R_2, M(10) / $R_2 \leftarrow$ Mem (10) /

S3 : add R_1, R_2 / $R_1 \leftarrow R_1 + R_2$

S4 : store M (1024), R1/Mem ((1024)) ← (R1)

S5 : store M ((R_2)), 1024 / Memo (R_2)←1024)/

Memory (10) constains 64 initialiy

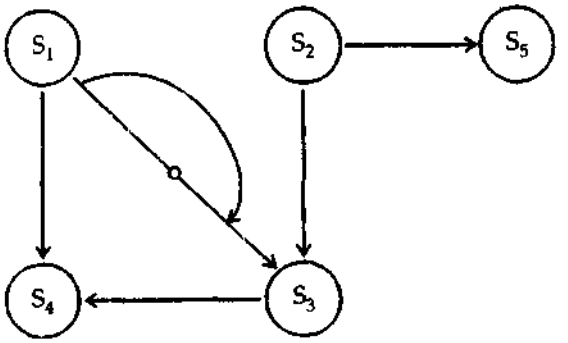
(i). Draw a dependence graph to show all the dependences.

Ans. Data & resource dependency to exploit implicit parallelism:

Implicit parallelism is a characteristic of a programming language that allows a compiler to automatically exploit the parallelism inherent to the computations expressed by some of the language's constructs. A pure implicitly parallel language does not need special directives, operators or functions to enable parallel execution.

A programmer that writes implicitly parallel

code does not need to worry about task division or process communication, focusing instead in the problem that his or her program is intended to solve. Implicit parallelism generally facilitates the design of parallel programs and therefore results in a substantial improvement of programmer productivity.



(ii). Are there any resource dependences? If only one copy of functional unit is available in the CPU.

Ans S4 and S5 need to use the same store unit in accessing the memory. Therefore they are potentially storage-dependent.

Q.2(b). Explain the cache address mapping techniques. Consider the following characteristics:

M_1 : 16 k words, 50 ns access time

M2 : 1 m words, 400 m access time

Assume A words cache blocks and a set size of 256 words with set anociative mapping

(i). Show the mapping between M_2 and M_1

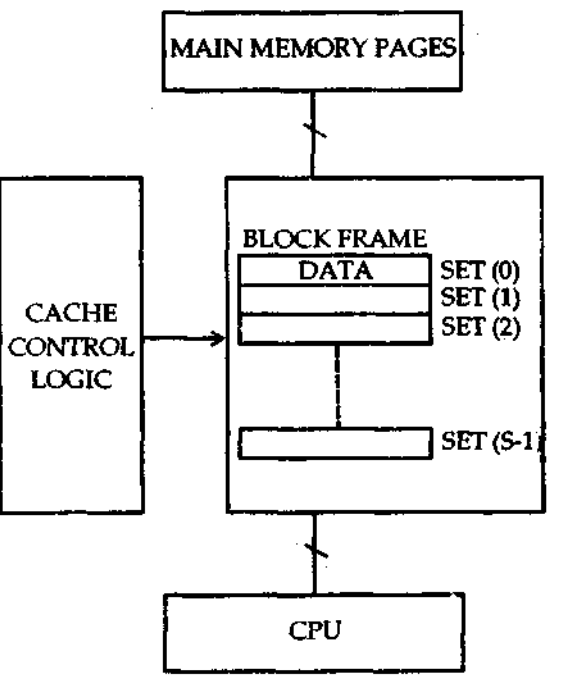
(ii). Calculate the effective memory access time with a cache hit ratio G = 0.95.

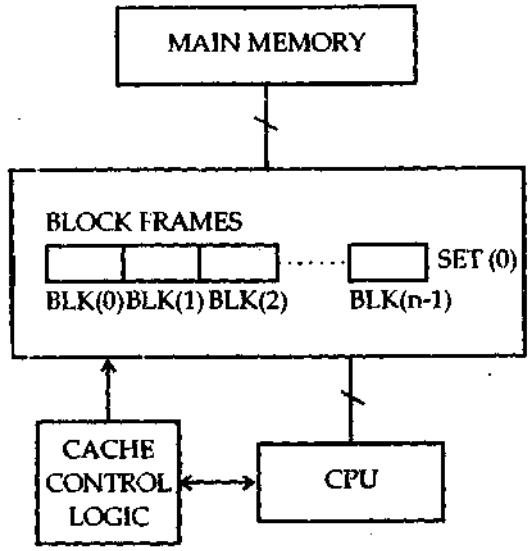
Ans. Cache address mapping techniques:

Cache mapping is the method by which the contents of main memory are brought into the cache and referenced by the CPU. The mapping method used directly affects the performance of the entire computer system.

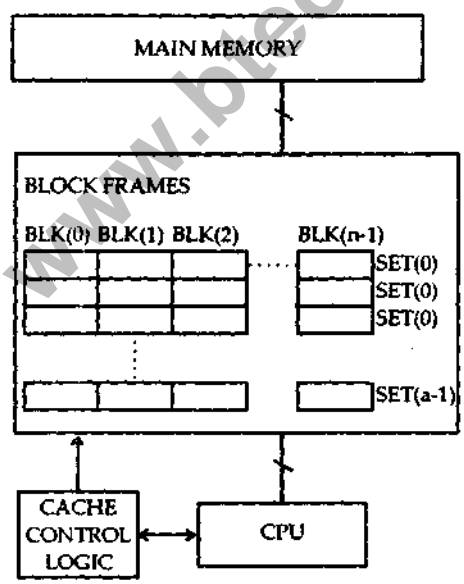
Direct mapping: Main memory locations can only be copied into one location in the cache. This is accomplished by dividing main memory into pages that correspond in size with the cache.

Fully associative mapping: Fully associative cache mapping is the most complex, but it is most flexible with regards to where data-can reside. A newly read block of main memory can be placed anywhere in a fully associative cache. If the cache is full, a Example of direct mapping used in cache Memory fully associated mapping used in cache memory.





Set associative mapping: Set associative cache mapping combines the best of direct and associative cache mapping techniques. As with a direct mapped cache, blocks of main memory data will still map into as specific set, but they can now be in any N-cache block frames within each set.



We are given

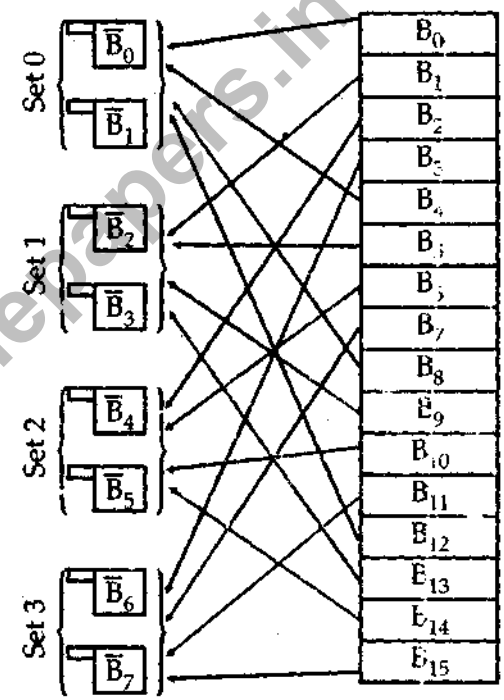
Set size k = 256 words

Cache size m = 8 words

∴ Number of sets, V = m/k= 8/256

= 1/32

(i). The mapping between M2 and M1 can be shown as follows:

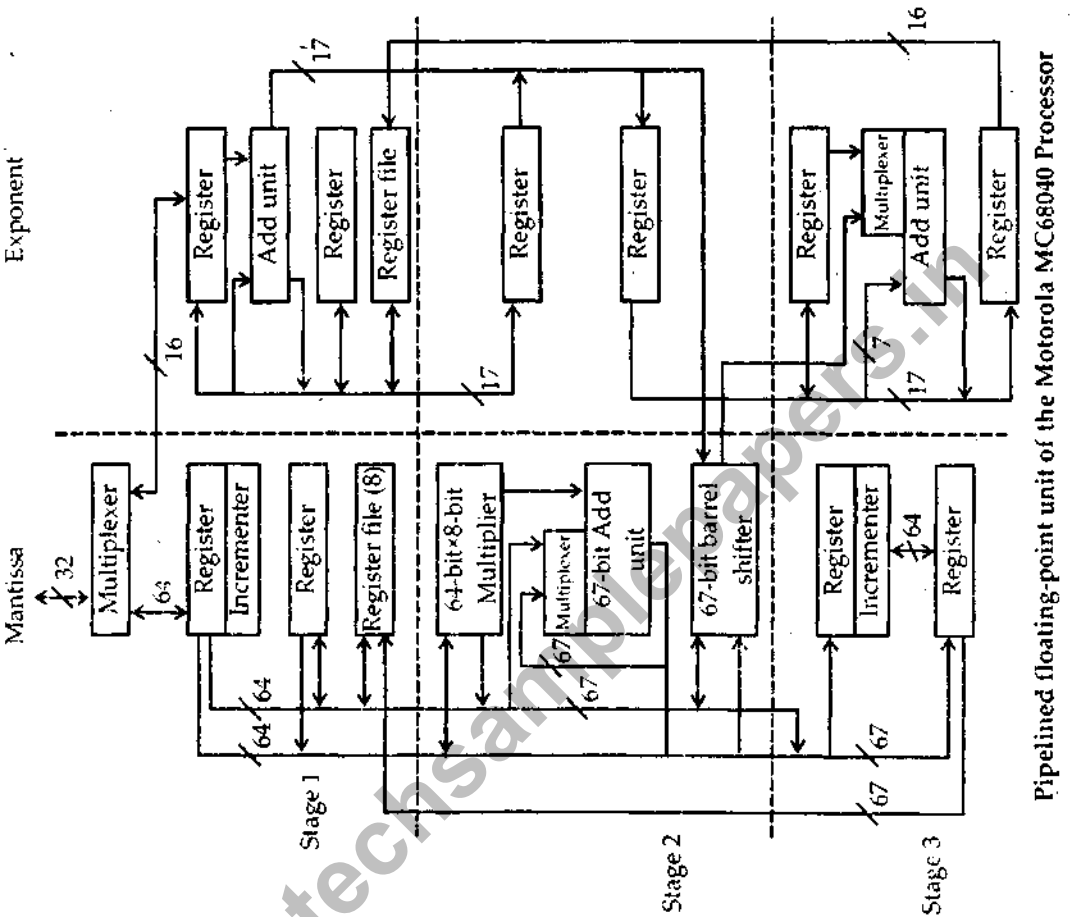


(ii). Effective memory access time

$T_{eff} = L_1 t_1 = 0.95 \times 50 = 47.50$ ns

Q.2(c). Give the block diagram for a pipelined floating point adder. Assume that exponent matching takes 0.1 sec., mantissa alignment 0.2 nsec., addiding mantissa. 1.0 n sec and normalizing result 0.2 nsec. What will be the highest clock speed which can be used to drive the adder. If two vectors of 100 components are to be added using the adder what will be the time of addition?

Ans.



Pipelined floating-point unit of the Motorola MC68040 Processor

Q. 3. Attempt any two parts of the following:

Q.(a). Consider the five-stage pipelined processor specified by the following reservation table:

	1	2	3	4	5	6
S_1	X					X
S_2		X			X	
S_3				X		
S_4					X	
S_5		X				X

(i). List the set of forbidden patencies and the collision vector.

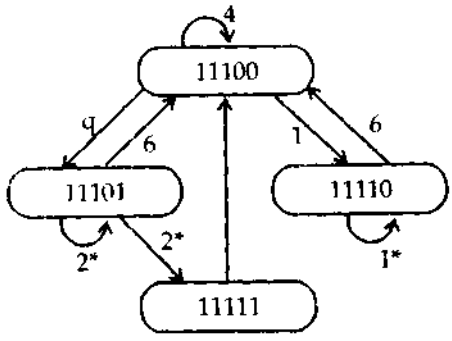
(ii). Draw the transition diagram showing all possible initial sequences without causing collision in the pipeline.

(iii). Identify simple cycles, greedy cycles and MAL (minimum average latency)

(iv). What will be the maximum through put of this pipeline.

Ans.(i). The balencies are, 3,4,5 and the collision vector is (11100).

(ii). State transition diagram is shown below:



(iii). Simple cycle : (1), (2), (1,6), (2,6), (2,2,6)

Greedy cycle : (2,6)

MAL : 3

Q.3(b). Discuss the superscalar and superpipelined processing. Also estimate the performance of superpipelined superscalar processor of degree (m, n).

Ans. Super scalar and super pipelined processing: A superscalar CPU architecture implements a form of parallelism called instruction-level parallelism within a single processor. It thereby allows faster CPU throughput than would otherwise be possible at the same clock rate. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

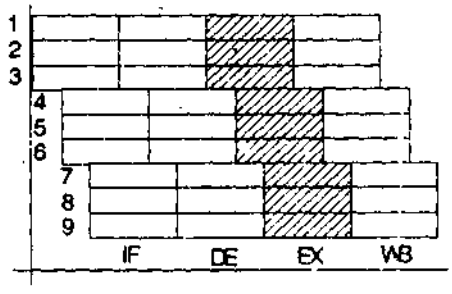
While a superscalar CPU is typically also pipelined, they are two different performance enhancement techniques. It is theoretically possible to have a non-pipelined superscalar CPU or a pipelined non-superscalar CPU..

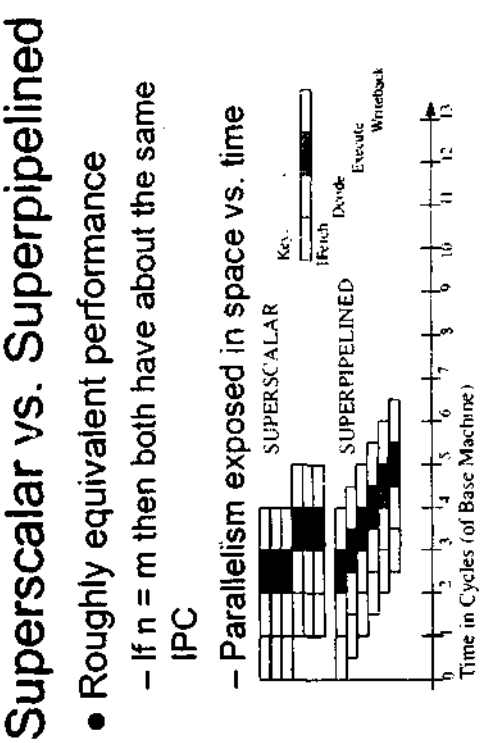
The superscalar technique is traditionally associated with several identifying characteristics. Note these are applied within a given CPU core.

- Instructions are issued from a sequential instruction stream
- CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
- Accepts multiple instructions per clock cycle

A superpipelined architecture will experience a performance hit whenever the pipeline isn't full, in the same way extra time is needed before the first car makes it through all the stages and comes off the assembly line. Whenever the processor's branch predictor fails to predict a branch instruction in a program, the pipeline gets flushed out while a (comparatively) slow read from random access memory into the processor cache takes place. From that point, the number of clock cycles for the first instruction to finish is equal to the number of stages in the pipeline. This has led some to criticize this design practice where they feel it has been taken to the extreme, such as on Intel's Prescott core, rumored to have somewhere around 30 stages.

- **Part-ii Superpipelined-Superscalar**



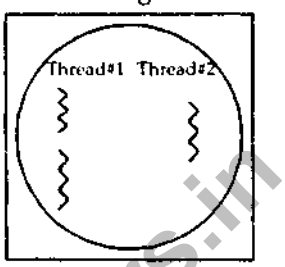


– Issue parallelism = IP = n inst / minor cycle
– Operation latency = OP = m minor cycles
– Peak IPC = n x m instr / major cycle

Q.3(c). Give the difference between a thread, a trace and a process. Also explain how simultaneous and multithreading is superior to multithreading (blocked and interleaved). What extra processor resources are required to support simultaneous multithreading?

Ans. Thread, trace and process: Thread of execution is a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in general, a thread is contained inside a process and different threads in the same process share some resources (most commonly memory), while different processes do not.

Tracing is a specialized use of logging to record information about a program's execution. This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical support personnel to diagnose common problems with software. Tracing is a cross-cutting concern.



A process is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently.

A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several windows of the same program often means more than one process is being executed. In the computing world, processes are *formally* defined by the operating system(s)(OS) running them and so may differ in detail from one OS to another.

Part-ii Simultaneous Multithreading Vs Blocked and interleaved Multithreading:

The most advanced type of multi-threading applies to superscalar processors. A normal superscalar processor issues multiple instructions from a single thread every CPU cycle. In Simultaneous Multi-threading (SMT), the superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction level parallelism, this type of multithreading is trying to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

For example:

1. Cycle i : instructions j and j+1 from thread A;

instruction k from thread B all simultaneously issued

2. Cycle i+1: instruction j+2 from thread A; instruction k+1 from thread B; instruction m from thread C all simultaneously issued
3. Cycle i+2: instruction j+3 from thread A; instructions m+1 and m+2 from thread C all simultaneously issued

To distinguish the other flavors of multithreading from SMT, the term Temporal multithreading is used to denote when instructions from only one thread can be issued at a time.

In addition to the hardware costs discussed for *interleaved* multithreading, SMT has the additional cost of each pipeline stage tracking the Thread ID of each instruction being processed. Again, shared resources such as caches and TLBs have to be sized for the large number of active threads.

Block multi-threading:

The simplest type of multi-threading is where one thread runs until it is blocked by an event that normally would create a long latency stall. Such a stall might be a cache-miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

For example:

1. Cycle i: instruction j from thread A is issued
2. Cycle i+1: instruction j+1 from thread A is issued
3. Cycle i+2: instruction j+2 from thread A is issued, load instruction which misses in all caches
4. Cycle i+3: thread scheduler invoked, switches to thread B
5. Cycle i+4: instruction k from thread B is issued
6. Cycle i+5: instruction k+1 from thread B is issued

Conceptually, it is similar to cooperative multi-tasking used in real-time operating systems in which tasks voluntarily give up execution time when they need to wait upon some type of event.

This type of multithreading is known as *Block* or *Cooperative* or *Coarse-grained* multithreading. Such additional hardware has these benefits:

- The thread switch can be done in one CPU cycle.
- It appears to each thread that they are executing alone and not sharing any hardware resources with any other threads. This minimizes the amount of software changes needed within the application as well as the operating system to support multithreading.

In order to switch efficiently between active threads, each active thread needs to have its own register set. For example, to quickly switch between two threads, the register hardware needs to be instantiated twice.

Interleaved multi-threading

A higher performance type of multithreading is where the processor switches threads every CPU cycle. For example:

1. Cycle i : an instruction from thread A is issued
2. Cycle i+1: an instruction from thread B is issued
3. Cycle i+2: an instruction from thread C is issued

The purpose of this type of multithreading is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there's less chance of one instruction in one pipe stage needing an output from an older instruction in the pipeline.

Conceptually, it is similar to pre-emptive multi-tasking used in operating systems. One can make the analogy that the time-slice given to each active thread is one CPU cycle.

This type of multithreading was first called *Barrel processing*, in which the staves of a barrel

represent the pipeline stages and their executing threads. *Interleaved* or *Pre-emptive* or *Fine-grained* or *time-sliced* multithreading are more modern terminology.

In addition to the hardware costs discussed in the *Block* type of multithreading, *interleaved* multithreading has an additional cost of each pipeline stage tracking the thread ID of the instruction it is processing. Also, since there are more threads being executed concurrently in the pipeline, shared resources such as caches and TLBs need to be larger to avoid thrashing between the different threads.

Q. 4. Attempt any two parts of the following:

Q.(a). Discuss the matrix multiplication on a Mesh. Give the algorithm that uses n × n processors arranged in a mesh contiguration. Also find the time complexity of the algerithm.

Ans. Matrix multiplication on a mesh: To multiply two vectors, a vector and a matrix, and two matrices on an Mesh optoelectronic computer Two mappings, group row and group submesh , of a matrix onto an Mesh are considered and the relative merits of each compared. to multiply a column and row vector use an optimal number of data moves for both the group row and group submesh mappings, algorithm to multiply a row vector and a column vector is optimal for the group row mapping, and our algorithm to multiply a matrix by a column vector is optimal for the group row mapping.

Algorithm that uses n x n processor arranged in a mesh

Stagger 2 matrices

a[0..n-1,0..n-1] and b[0..n-1,0..n-1]

Detailed Algorithm

```
Global   r, {Dimension of matrices}
         k,
Local    a, b, c;
Begin
  for k:=1 to n-1 do
      forall P(i,j) where 1 = i,j < n do
    if i = k then a:= fromleft(a);
      if j = k then b:=fromdown(b);
    end forall;
  endfor k;
```

Part II:

The Fox-otto-key has a total overhead $h(S,n) = O(n \log n + S^2\sqrt{n})$. The workload $w = O(S^3) = O(n \log n + S^2\sqrt{n})$. Thus we must have $O(S^3) = O(n \log n)$ and $O(S) = O(\sqrt{n})$. Combining the two, we obtain the efficiency function $O(S^3) = O(n^{3/2})$, where $1 \le n \le S^2$.

Q.4(b). Give the PRAM algorithm for solving a first order inear recurrence:

$x_i = a_i x_{i-1} + d_i$ for i = 1,2,....n where the value of x_0, a_1, a_2, ... a_n and d_1, d_2, d_3,... d_n are given. Assume $x_0 = 0$ and $a_1 = 0$

Ans. Step (1)

1. Compute a(i) * x(i–1) + d(i)
2. Store in x(i).

Step (2)

```
1. i ← 0
2. Repeat
     begin
       Read x (i – 1)
       Read d (i)
       Read a (i)
       Compute a(i) × x (i – 1) + d(i)
       Store in x(i)
     end
   until (i=n)
```

Q.4(c). Explain the Bidirectional Gaussian elemination for solving a set of linear algebraic equation.

Ans. Bidirectional Gaussian elimination for saving a set of linear algebraic equation:

The regular nature of this algorithm and the regularity of the domain make it inherantly parallel and suitable for domain decomposition.

The LU factorisation is a triple nested loop. The outer loop controls how much of the matrix remains to be factorized. At each iteration, the remaining part of the matrix is a smaller submatrix in the lower right hand corner. The

elimination process requires that information from one node be broadcast to all the other nodes.

A sequential factor algorithm is

for j = 1 to n – 1 do
Find max [A(i,j) to A(n,j)] in jth column
swap row to make A(j,j) the pivot
divide A(j+1,j) to A(n,j) by A(j,j)
for j = jth to n do
for k = j + 1 to n do
A(k, j) ← A(k,j) - A(k,j) * A(j,j)
end of k-loop
end of j-loop
and of j loop

This sequential code can be executed directly on a single node. The next step is to distribute the matrix elements. The matrix is distributed by columns among the processor nodes. The matrix domain is mapped to the nodes so that all the processors aur approximately the same number of columns of the matrix.

Q. 5. Attempt any two parts of the following:

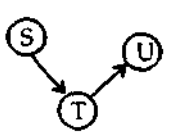
Q.(a). Consider the following double loop (L_1&L_2)

L_1 : for (2 = 0; i < 4, 1++
{
L_2 : for (J = 0; J<4; J++)
S : A [L+1] [J]= B[i] [J] + C [i] [J]
T : B [i] [J+1] = A[2] [J+1] + 1
U : D (i, j) = B [i] [J–2] – 2
}
1

(i). Find the dependence caused by all possible pairs of variables. Identify the type of each dependences and find its distance and direction vector.

(ii). Draw the statement and iteration dependence graph for the loop.

Ans.(i). The dependence can be shown as follows



- There is flow dependence betweer S and T and T and U.
- The distance and direction vectors in S, are (1,0) and (<,=) respectively.
- The distance and direction vectors in T, are (0,1) and (=,<) respectively.
- The distance and direction vectors in U are, (0,0) and (=,=) respectively.

Q.5(b). Explain the following terms associated with fast and efficient synchronization schemes on a shared-memory multiprocessor:

(i). Busy-wait verses sleep-wait protocols for sole access of a critical section.

(ii). Lock mechanisms, for pre-synchronization to achieve sole access to a critical section.

(iii). Post-synchronization method.

Ans.(i). Busy-wait verses sleep-wait protocols for sole access of a critical section: In busy wait, the process remains loaded in the processors context registers and is allowed to continually retry. The reason for using busy wait is that it offers a faster response when the shared object becomes available.

In 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. The hardware complexity increases in a multiprocessor using sleep wait as compared with those implementing busy wait.

(ii). Lock mechanism for pre-synchronization to achieve sole access to a critical section: When locks are used to synchronize processes in a multiprocessor, busy wait is used more often than sleep wait. Busy wait may offer a better performance if it entails lees use of processors, memories or network channels. If sleep wait queues are managed using lock synchronization, it may be necessary for a process to wait for access to a sleep wait queue.

(iii). Post-synchronization method: It is also known as optimistic synchronization. This

method also updates the atom by the register process. But sole access is granted after the atomic operation via abortion. A process may secure sole access after first completing an atomic operation on a local version o f the atom and then executing another atomic operation on the global version of the atom.

The second atomic operation determines if a concurrent update of the atom has been made since the first operation was begun. If a concurrent update has taken place, the global version is not updated instead, the first atomic operation is aborted and restarted from the new global version.

Q.5(c). Write short notes on any two of the following:

(i). Combined parallel work-sharing construct.

(ii). Run-time library routines

(iii). Parallel execution environment routines.

Ans. (i) Combined parallel work sharing construct:

A combined parallel work-sharing construct allows you to specify a parallel region that already contains a single work-sharing construct. These combined constructs are semantically identical to specifying a parallel construct enclosing a single work-sharing construct.

The semantics of these directives are identical to that of explicitly specifying a parallel directive followed by a single work-sharing construct. The following sections describe the combined parallel work-sharing constructs:

- the parallel for directive.
- the parallel sections directive.

API compiler directives begins execution as a single process, called the master thread of execution. The master thread executes sequentially until the first parallel construct is encountered.

In OpenMP Fortran API, the PARALLEL and END PARALLEL directives define the parallel construct. When the master thread encounters a parallel construct, it creates a team of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes the static extent as well as the routines called from within the construct. When the END PARALLEL directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.

You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

(ii)Run time library routines:

Run-time library routines to assist you in managing your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

The routines are structured in following three parts:

Control threads, processors and the parallel environment.

- omp_get_active_level: Number of active parallel regions
- omp_get_ancestor_thread_num: Ancestor thread ID
- omp_get_dynamic: Dynamic teams setting
- omp_get_level: Number of parallel regions
- omp_get_max_active_levels: Maximal number of active regions

- omp_get_max_threads: Maximal number of threads of parallel region
- omp_get_nested: Nested parallel regions
- omp_get_num_procs: Number of processors online
- omp_get_num_threads: Size of the active team
- omp_get_schedule: Obtain the runtime scheduling method
- omp_get_team_size: Number of threads in a team
- omp_get_thread_limit: Maximal number of threads
- omp_get_thread_num: Current thread ID
- omp_in_parallel: Whether a parallel region is active
- omp_set_dynamic: Enable/disable dynamic teams
- omp_set_max_active_levels: Limits the number of active parallel regions
- omp_set_nested: Enable/disable nested parallel regions
- omp_set_num_threads: Set upper team size limit
- omp_set_schedule: Set the runtime scheduling method

Initialize, set, test, unset and destroy simple and nested locks.

- omp_init_lock: Initialize simple lock
- omp_set_lock: Wait for and set simple lock
- omp_test_lock: Test and set simple lock if available
- omp_unset_lock: Unset simple lock
- omp_destroy_lock: Destroy simple lock
- omp_init_nest_lock: Initialize nested lock
- omp_set_nest_lock: Wait for and set simple lock
- omp_test_nest_lock: Test and set nested lock if available
- omp_unset_nest_lock: Unset nested lock
- omp_destroy_nest_lock: Destroy nested lock

Portable, thread-based, wall clock timer.

- omp_get_wtick: Get timer precision.
- omp_get_wtime: Elapsed wall clock time.

(iii) Parallel execution environment routines:

Parallel threads created by the run-time environment through the OpenMP interface are considered independent of the threads you create and control using calls to the Fortran Pthreads library module. References within the following descriptions to "serial portions of the program" refer to portions of the program that are executed by only one of the threads that have been created by the run-time environment. Execution environment routines are:

- omp_get_dynamic: see omp_get_dynamic
- omp_get_max_threads: see omp_get_max_threads
- omp_get_nested: see omp_get_nested
- omp_get_num_procs: see omp_get_num_procs
- omp_get_num_threads: see omp_get_num_threads
- omp_get_thread_num: see omp_get_thread_num
- omp_in_parallel: see omp_in_parallel
- omp_set_dynamic: see omp_set_dynamic
- omp_set_nested: see omp_set_nested
- omp_set_num_threads: see omp_set_num_threads

Included in the OpenMP run-time library are two routines that support a portable wall-clock timer. The OpenMP timing routines are:

- omp_get_wtick: see omp_get_wtick
- omp_get_wtime: see omp_get_wtime

The run-time library also supports a set of simple and nestable lock routines. You must only lock variables through these routines. Simple locks may not be locked if they are already in a locked state. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable locks may be locked multiple times by the same thread. Nestable lock variables are associated with nestable locks and may only be passed to nestable lock routines.