CS405
Computer System Architecture

MODULE 6

Syllabus

- Multithreaded and data flow architectures
  - Latency hiding techniques,
- Principles of multithreading
  - Multithreading Issues and Solutions,
- Multiple context Processors,
- Finegrain Multicomputer
- Fine-grain Parallelism.
- Dataflow and hybrid architecture

LATENCY HIDING TECHNIQUES

- Parallel computers uses shared memory.
- The access of remote memory increases memory latency; because processor speed rate is faster than memory speed rate.
- So, a scalable multiprocessor or large scale multi-computer needs a latency reducing or hiding mechanism
- The latency hiding techniques are:
  - ➢Pre-fetching techniques
  - ➢Distributed coherent caches
  - ➢Relaxed memory consistency
  - ➢Multiple context Switches

Prefetching Techniques

- Prefetching means, bring instructions or data close to the processor before they are actually needed.
- Prefetching classifications are
  1. Binding prefetch
  2. Non- binding prefetch
  3. Software controlled prefetching
  4. Hardware controlled prefetching

1. Binding prefetch

- the fetch is assumed to have happened when the prefetch instruction is issued
  - it is the responsibility of the "fetching process" to ensure that no other processor will update the prefetched value before it is actually used
  - this may require the use of locks, if necessary
- It put prefetched value into register.
- The problem is the value become stale if another processor modifies the same location during the interval between prefetch and reference.

2. Non- binding prefetch

- the system cache coherence protocol will make sure to invalidate a prefetched value if it is updated prior to its use.
- It put prefetched values in to cache.
- It aims at reducing the basic cache miss rate by fetching the data from remote memory in to the cache before it is required by the processor
- The data is visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.

3.Software controlled prefetching

- Here the prefetching operation is initiated by processor executing a prefetch instruction
- It can extends the possible prefetch-reference interval
- Prefetching to be done selectively based on program knowledge.
- It simplifies the hardware
- The disadvantage is , needs extra instruction overhead to generate the prefetch.

4.Hardware controlled prefetching

- Here the hardware prefetches at run time.
- It provides better dynamic information
- No instruction overhead to issue prefetches
- Difficult to detect memory access patterns
- Instruction lookahead is limited by branches and buffer size.

Benefits of pre fetching

- When a pre fetch is issued early enough in the code so that the line is already in the cache by the time it is referenced.
- If multiple pre fetches are issued back to back to fetch the data structure, the latency of all but the first pre fetched reference can be hidden due to the pipelining of the memory accesses.
- In multiprocessors that use an ownership based cache coherence protocol,
  - If a cache block line is to be modified, prefetching it directly **with** ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership.

Distributed Coherent Caches

- It is supported by hardware to reduce cache misses.
- In bus-base multi-processors, coherence problem is easily solved using snoopy bus protocols
- In large scale multi-processors, some did not provide caches, others provided caches that must be kept coherent by software.
- Benefits:
  - Reduction in the number of cycles wasted due to read-misses
  - The cycles wasted due to write misses were also reduced

Relaxed Memory Consistency

- Memory consistency model is a set of rules that governs how memory systems will process memory operations (load/store) from multiple processors
- Consistency models are used in distributed systems like distributed shared memory systems or distributed data stores
- The following are the relaxed memory consistency models:
  1. Processor Consistency (PC)
  2. Release consistency

Processor Consistency (PC)

- Here Writes issued by each individual processor are always in program order.
- The order of writes from two different processors can be out of program order.
- Two conditions required for ensuring processor consistency:
  - Before a read is allowed to perform with respect to any other processor, all previous read accesses must be performed.
  - Before a write is allowed to perform with respect to any other processor, all previous read or write accesses must be performed.

- In the first example, all the writes by each processor are seen in the order they occurred in by the other processors; and the transactions are coherent.
- Second example is not processor consistent, as the writes by P1 and P3 are seen out of order by P2 and P4 respectively.

Example 1: Processor Consistent

P_1	W(x)1	W(x)3		
P_2			R(x)1	R(x)3
P_3	W(y)1	W(y)2		
P_4			R(y)1	R(y)2

Example 2: Not Processor Consistent

P_1	W(x)1	W(x)3		
P_2			R(x)3	R(x)1
P_3	W(y)1	W(y)2		
P_4			R(y)2	R(y)1

Release consistency

- Memory consistency must be maintained to avoid undesirable outcomes.
- To achieve better performance, some relaxed models are explored and the release consistency is an aggressive relaxing attempt.
- Release consistency acquire synchronization variables or locks.
- Locks can be acquire and release. An acquire is a read operation, that gains permission to access a set of data. A release is a write operation that gives away such permission.

- For a lock “L”, acquire and release operations can be denoted by

```
acquire(L)        // read
release(L)        //write
```

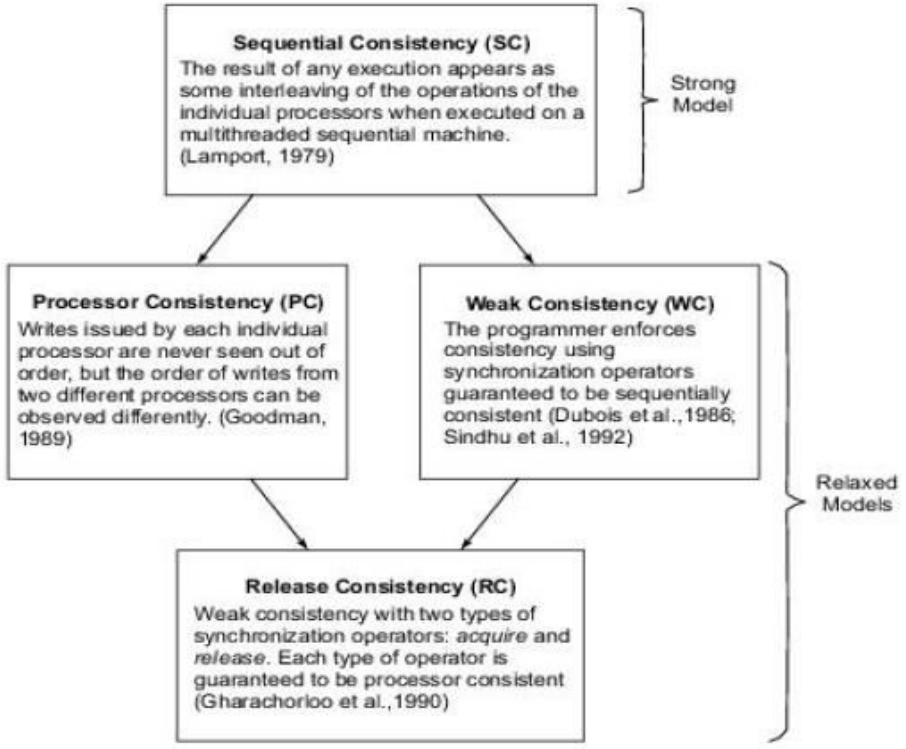
- The conditions for release consistency are:
  - Before a read or write access is allowed to perform with respect to any other process, all previous acquires accesses must be performed.
  - Before a release access is allowed to perform with respect to any other process ,all previous read or write access must be performed.
  - Special access are processor consistent with one another. Release consistency requires processor consistency.

- Release consistency can be satisfied by
  1. stalling the processor on an acquire access until it completes and
  2. delaying the completion of release access until all previous memory accesses complete

- Advantages
  - Have the potential for increased performance by hiding as much write latency as possible.
- Disadvantages
  - is increased hardware complexity and a more complex programming model.

Using multiple-contexts to hide latency

- Continuous context switching can hide latency

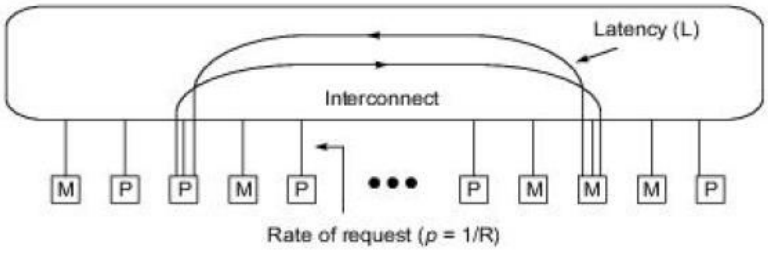


Four memory consistency models

19

PRINCIPLES OF MULTITHREADING

- Multi-threading means the processor should handle multiple contexts simultaneously on a context-switching basis.
- One of the multi-threaded architecture is shown below. It is a network of processor(P) and memory (M) nodes.



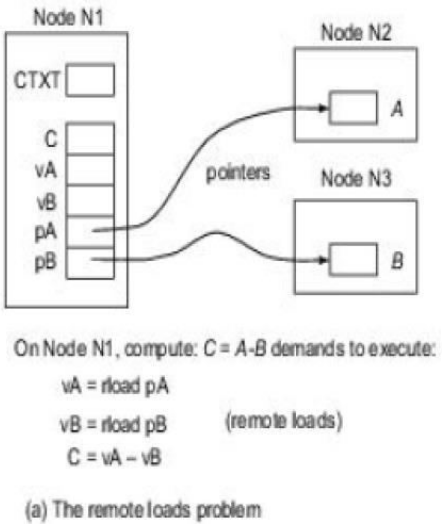
(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)

20

- To analyze the performance of this network , 4 parameters are used.
  1. Latency (L) – communication latency on a remote memory access. The value of L includes network delay , cache miss penalty etc.
  2. Number of threads (N) – It is the number of threads that can be interleaved in each processor. Thread contains program counter, register set and context status words
  3. Context switching overhead (C) – Cycle lost in performing context switching in a processor
  4. Interval between switches (R) – The cycles between switches triggered by remote reference. The inverse p=1/R is called the rate of requests for remote access.

- To increase efficiency, reduce the rate of requests by using distributed coherent caches, and eliminate processor waiting through multithreading.
- Parallel processors operate asynchronously in a network environment. These asynchrony creates 2 types of latency problems.
- “Remote loads” and “synchronizing loads”

Multithreading Issues & Solutions-Remote load problem

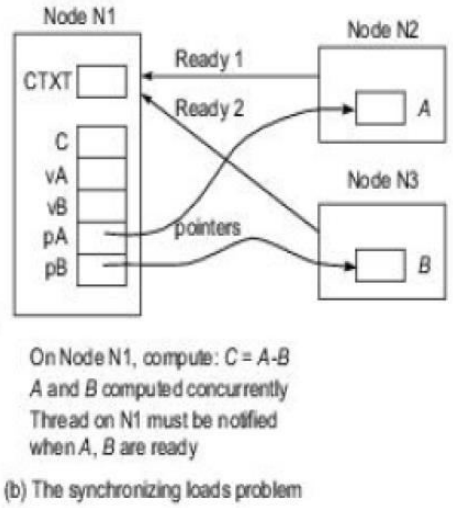


(a) The remote loads problem

- Remote load problem
  - In the following figure , variable A and B are located on nodes N2 and N3.
  - They need to be brought to node N1 to compute the difference A-B in variable C. So we need to execute 2 remote load (rload) and then subtraction.
  - The main issue in remote load is how to avoid idling in node N1 during the load operation. The latency caused by remote load is architectural property

- In figure, PA and PB are the pointers to A and B.
- The two rload can be issued from the same thread or from two different threads.
- The context of the computation on N1 is represented by the variable CTXT.

Synchronizing loads problem



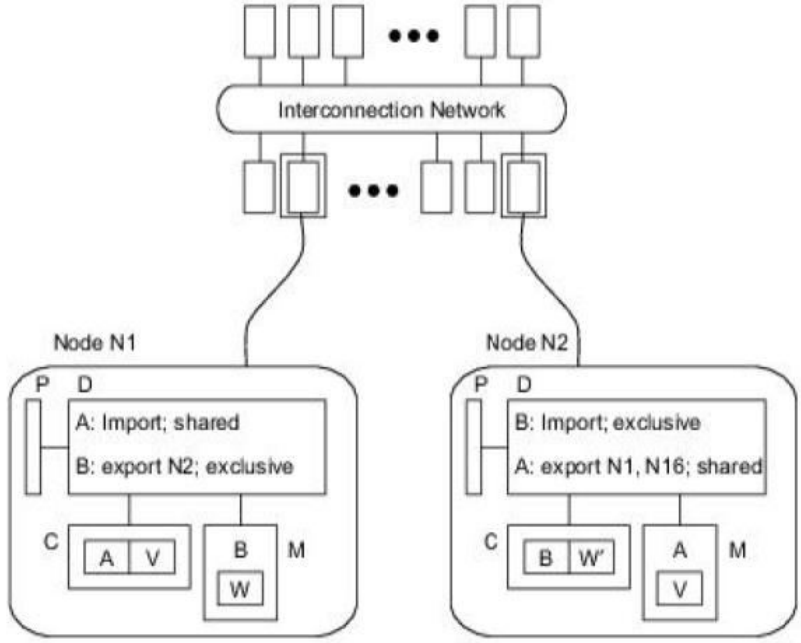
(b) The synchronizing loads problem

- The following figure shows the idling due to synchronizing loads. A and B are computed by concurrent processes and not sure exactly when they will be ready for node N1 to read.
- The ready signal may reach node N1 asynchronously.
- The latency caused by synchronizing loads also depends on scheduling and the time it take to compute A and B.

- Synchronization latency is often unpredictable while the remote load latencies are often predictable.
- The solution to the asynchrony problem is to multiplex among many threads.
- When one thread issues a remote load request, the processor begins work on another thread and so on.

- Make sure that messages carry continuations.
- Suppose after issuing a remote load from thread T1, we switch to thread T2, which also issues a remote load.
- The responses may not return in the same order.
- This may be caused by travelling distance of messages , congestion etc.
  - To avoid this problem, associate each remote load and response with an identifier for the appropriate thread;
  - so that it can be re-enabled on the arrival of a response.
- These identifiers are also called "Continuations"

- Another solution for overcoming the asynchrony problem is, distributed caching.
- Every memory location has an owner node, for example, N3 owns B and N2 owns A.
- The directories are used to contain import, export list and state whether the data is shared or exclusive.
- Distributed caching offers a solution for the remote load problem, but not for the synchronizing load problem.



P = Processor; D = Directory; C = Cache; M = Memory

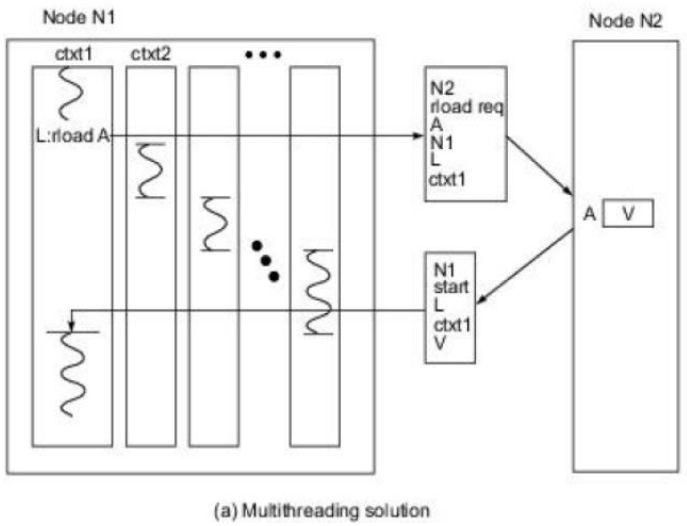
(b) Distributed cacheing

MULTIPLE CONTEXT PROCESSORS

- Multithreaded systems are constructed with multiple-context (or multi-threaded) processors.
- Multithreaded processors are known as multiple context processors.
- A single thread processor will wait during a remote reference, and it is idle for a period of time “L”.
- A multi threaded processor will suspend the current context and switch to another, so that some fixed number of cycles it will again be busy doing useful work , even though the remote reference is outstanding.
- The objective is to maximize the fraction of time that the processor is busy.

Efficiency = busy / (busy+switching+idle)

- The basic idea behind the multithreaded machine is to interleave the execution of several contexts in order to dramatically reduce the value of idle.
- The various context states are – Ready, Running, Leaving and Blocked.
- At most one context running or leaving. If all context are blocked, we say the processor is “idle”.



(a) Multithreading solution

- The following figure is a multi-threaded model , one thread per context.
- Each context is represented by its own program counter (PC), register set and process status word (PSW)

Context-Switching policies

- Different multithreaded architectures are distinguished by the context switching policies adopted.
- The four switching policies are given below

1. Switch on cache miss
2. Switch on every load
3. Switch on every instruction
4. Switch on block of instruction

1. Switch on cache miss
   - A context is preempted when it causes a cache miss. Here ‘R’ is the average interval between misses; and ’L’ is the time required to satisfy the miss
2. Switch on every load
   - Allows switching on every load.
   - Here, ‘R’ is the average interval between loads.
   - There are two sources of latency ; L1 and L2.
   - L1 represents the latency on cache miss.
   - L2 represents zero – cycle memory latency.

3. Switch on every instruction
   – Allows switching on every instruction, whether it is a load or not
4. Switch on block of instruction
   – Blocks of instructions from different threads are interleaved.
   – This will improve the cache hit ratio due to locality.

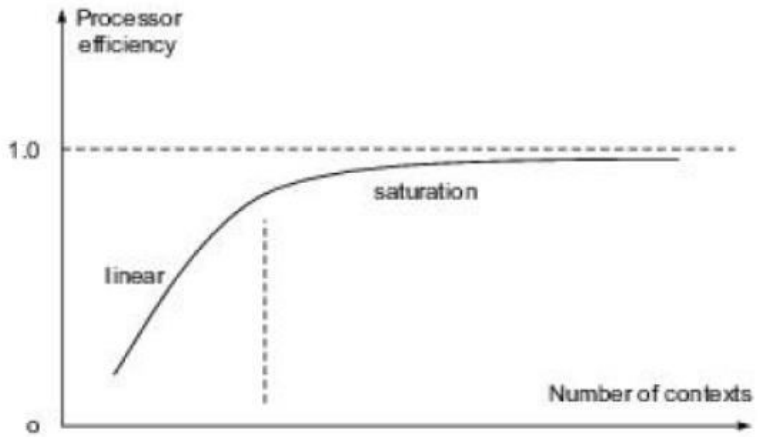
Processor efficiency

- A single thread processor executes a context until a remote reference is issued; and then is idle until the reference is completed.
- There is no context switch and switch overhead.
- Let R be the amount of time the processor is busy. L be the amount of time the processor is idle.
- The efficiency of a single threaded machine is given by

$$E_1 = R/(R+L)$$

- With multiple context, memory latency can be hidden by switching to a new context, and the switch take C cycles of overhead.
- There is always a “context ready” to execute when a switch occurs, so the processor is never idle.

Efficiency curve



(c) Efficiency curve

- The processor efficiency is analyzed in two different conditions.
- In the above figure shows the context switching and processor efficiency as a function of the number of contexts.
- Saturation region
  - Here the processor operates with maximum utilization. Efficiency is

$$E_{sat} = \frac{R}{R+C} = \frac{1}{1+C/R}$$

  - Here , the efficiency is independent of latency

- Liner region
  - When the number of contexts is below the saturation point, there may be no context after a context switch, so the processor will experience idle cycles.
  - The time required to switch to a ready context, execute it until a remote reference is issued, and process the reference is equal to R+C+L.
  - Assuming N is below the saturation point, during this time all the other contexts have a turn in the processor.

- Efficiency is given by :-

$$E_{\text{lin}} = \frac{NR}{R + C + L}$$

- Ie… the efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant.

FINE GRAIN MULTI-COMPUTERS

- In parallel computing, granularity (or grain size) of a task is a measure of the amount of work (or computation) which is performed by that task
- Granularity is usually measured in terms of the number of instructions executed in a particular task.
- Alternately, granularity can also be specified in terms of the execution time of a program

- Depending on the amount of work which is performed by a parallel task, parallelism can be classified into three categories:

1. Fine-grained parallelism
2. Medium-grained parallelism
3. Coarse-grained parallelism

Fine-grained parallelism

- In fine-grained parallelism, a program is broken down to a large number of small tasks. These tasks are assigned individually to many processors.
- The amount of work associated with a parallel task is low and the work is evenly distributed among the processors.
- Hence, fine grained parallelism facilitates load balancing
- As each task processes less data, the number of processors required to perform the complete processing is high.
- This in turn, increases the communication and synchronization overhead.

- Fine-grained parallelism is best exploited in architectures which support fast communication.
- Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.
- It is difficult for programmers to detect parallelism in a program, therefore, it is usually the compilers' responsibility to detect fine-grained parallelism
- Connection Machine (CM-2) and J-Machine (Jellybean-Machine) are examples of fine-grain parallel computers that have grain size in the range of 4-5 μs

Coarse-grained parallelism

- In coarse-grained parallelism, a program is split into large tasks.
- Due to this, a large amount of computation takes place in processors.
- This might result in load imbalance, where in certain tasks process the bulk of the data while others might be idle.
- Coarse-grained parallelism fails to exploit the parallelism in the program as most of the computation is performed sequentially on a processor.
- Cray Y-MP is an example of coarse-grained parallel computer which has a grain size of about 20s

NO	FINE-GRAINED SIMD	COARSE-GRAINED SIMD
1	Fine Grain SIMD have less computation time then the coarse grain architecture	Coarse Grain SIMD have more computation time then the Fine grain architecture.
2	Here, programs are broken into large number of small tasks.	Here, programs are broken into small number of large task.
3	Fine Grain SIMD have much higher level of parallelism then Coarse grain SIMD.	Coarse grain SIMD have lower level of parallelism then Fine Grain SIMD.
4	Here, Grain Size is over 1000 instructions.	Here, Grain Size in range of 2-500 instructions.
5	Here, the size of subcomponents is much smaller than the Coarse grained.	Here, the size of subcomponents is more than the Fine-Grained.
6	Here, two types of parallelism can be obtained – a) Instruction Level Parallelism b) Loop Level Parallelism	Here, these two types of parallelism can be obtained – a) Sub-program b) Program Level Parallelism
7	In Fine Grain SIMD, Load Balancing is proper.	In Coarse Grain SIMD, Load Balancing is improper.

NO	FINE-GRAINED SIMD	COARSE-GRAINED SIMD
8	Here Parallelism can be detected using compiler.	Here Parallelism can't be detected using compiler.
9	Fine Grain SIMD is a much costlier process than the Coarse Grain SIMD.	Coarse Grain SIMD is much cheaper than the Fine Grain SIMD.
10	Fine Grain is the concept of future multi-threaded architectures to be used in the future also.	Coarse Grain is in one of the earlier concepts of single-threaded architectures.
11	The Detailed description is further divided into many small subcomponents and makes the processes less complex from the original one and from the coarse-grained also.	The Detailed description is divided into large subcomponents and makes the processes less complex than the original one but more complex than Fine-Grained.
12	**Examples** Connection Machine (CM-2), J-Machine, etc.	**Examples** CRAY Y, etc.

Latency analysis in Fine-Grain Parallelism

Table 9.3 *Fine-Grain, Medium-Grain, and Coarse-Grain Machine Characteristics of Some Example Systems*

Characteristics	*Machine*			
	Cray Y-MP	*Connection Machine CM-2*	*Intel iPSC/1*	*MIT J-Machine*
Communication latency, T_c	40 ns via shared memory	600 μs per 32-bit *send* operation	5 ms	2 μs
Synchronization overhead, T_s	20 μs	125 ns per bit-slice operation in lock step	500 μs	1 μs
Grain size, T_g	20 s	4 μs per 32-bit result per PE instruction	10 ms	5 μs
Concurrency (DOP)	2–16	8K–64K	8–128	1K–64K
Remark	Coarse-grain supercomputer	Fine-grain data parallelism	Medium-grain multicomputer	Fine-grain multicomputer

- The computing granularity and communication latency of leading early examples of multiproccssors, data-parallel computers, and medium-and fine-grain multicomputers are summarized in Table.
- The communication latency:
  - measures the data or message transfer time on a system interconnect.
  - This corresponds to the shared-memory access time on the Cray Y-MP, the time required to send a 32-bit value across the hypercube network in the CM-2. and the network latency on the iPSC/1 or J-Machine.

- The synchronization overhead :
  - is the processing time required on a processor, or on a PE, or on a processing node of a multicomputer for the purpose of synchronization.
  - The sum **Tc + Ts** gives the total time required For IPC.
  - The shared-memory Cray Y-MP had a short Tc but a long Ts. The SIMD machine CM-2 had a short Ts but a long Tc.
- The grain size
  - is measured by the execution time of a typical program, including both computing time and communication time involved.
  - Supercomputers handle large grain.

- Large grain implies lower concurrency or a lower DOP (degree of parallelism).
- Fine grain leads to a much higher DOP and also to higher communication overhead.

Caltech Mosaic C

- The Caltech Mosaic C was an experimental fine-grain multicomputer
- It employs single-chip nodes and advanced packaging technology to demonstrate the performance/cost advantages of fine-grain multicomputer architecture.