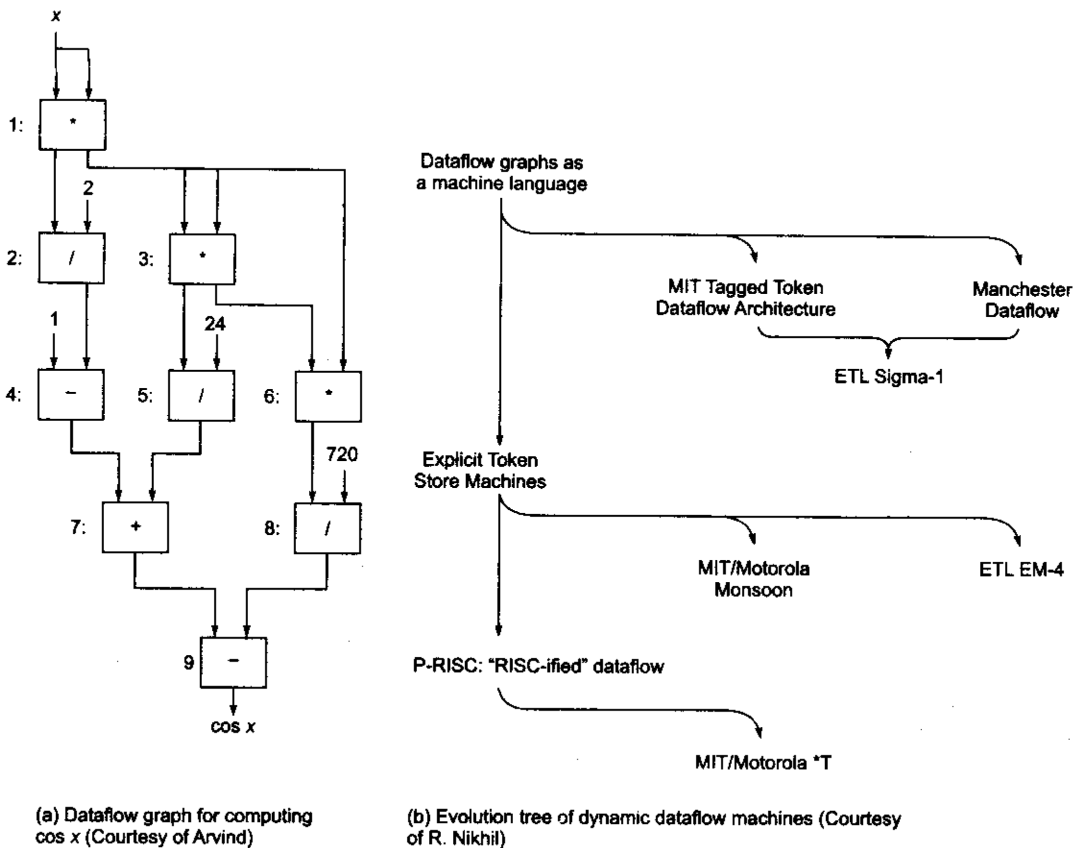


(a) Dataflow graph for computing cos x (Courtesy of Arvind)

(b) Evolution tree of dynamic dataflow machines (Courtesy of R. Nikhil)

Fig. 9.31 An example dataflow graph and dataflow machine projects

Example 9.7 The dataflow graph for the calculation of cos$x$ (Arvind, 1991).

This dataflow graph shows how to obtain an approximation of cos$x$ by the following power series computation:

$$\cos x \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} \tag{9.6}$$

The corresponding dataflow graph consists of nine operators (actors or nodes). The edges in the graph interconnect the operator nodes. The successive powers of x are obtained by repeated multiplications. The constants (divisors) are fed into the nodes directly. All intermediate results are forwarded among the nodes.

Static versus Dynamic Dataflow *Static dataflow computers* simply disallow more than one token to reside on any one arc, which is enforced by the firing rule: A node is enabled as soon as tokens are present on all input arcs and there is no token on any of its output arcs. Jack Dennis proposed the very first static dataflow computer in 1974.

The static firing rule is difficult to implement in hardware. Special feedback *acknowledge signals* are needed to secure the correct token passing between producing nodes and consuming nodes. Also, the static rule makes it very inefficient to process arrays of data. The number of acknowledge signals can grow too fast to be supported by hardware.

However, static dataflow inspired the development of *dynamic dataflow computers*, which were researched vigorously at MIT and in Japan. In a dynamic architecture, each data token is tagged with a context descriptor, called a *tagged token*. The firing rule of tagged-token dataflow is changed to: A node is enabled as soon as tokens with identical tags are present at each of its input arcs.

With tagged tokens, tag matching becomes necessary. Special hardware mechanisms are needed to achieve this. In the rest of this section, we discuss only dynamic dataflow computers. Arvind of MIT pioneered the development of tagged-token architecture for dynamic dataflow computers.

Although data dependence does exist in dataflow graphs, it does not force unnecessary sequentialization, and dataflow computers schedule instructions according to the availability of the operands. Conceptually, "token"-carrying values flow along the edges of the graph. Values or tokens may be memory locations.

Each instruction waits for tokens on all inputs, consumes input tokens, computes output values based on input values, and produces tokens on outputs. No further restriction on instruction ordering is imposed. No side effects are produced with the execution of instructions in a dataflow computer. Both dataflow graphs and machines implement only functional languages.

Pure Dataflow Machines Figure 9.31b shows the evolution of dataflow computers. The MIT *tagged-token dataflow architecture* (TTDA) (Arvind et al, 1983), the Manchester Dataflow Computer (Gurd and Watson, 1982), and the ETL Sigma-1 (Hiraki and Shimada, 1987) were all pure dataflow computers. The TTDA was simulated but never built. The Manchester machine was actually built and became operational in mid-1982. It operated asynchronously using a separate clock for each processing element with a performance comparable to that of the VAX/780.

The ETL Sigma-1 was developed at the Electrotechnical Laboratory, Tsukuba, Japan. It consisted of 128 PEs fully synchronous with a 10-MHz clock. It implemented the I-structure memory proposed by Arvind. The full configuration became operational in 1987 and achieved a 170-Mflops performance. The major problem in using the Sigma-1 was lack of high-level language for users.

Explicit Token Store Machines These were successors to the pure dataflow machines. The basic idea is to eliminate associative token matching. The waiting token memory is directly addressed, with the use of full/empty bits. This idea was used in the MIT/Motorola Monsoon (Papadopoulos and Culler, 1988) and in the ETL EM-4 system (Sakai et al, 1989).

Multithreading was supported in Monsoon using multiple register sets. Thread-based programming was conceptually introduced in Monsoon. The maximum configuration built consisted of eight processors and eight I-structure memory modules using an 8×8 crossbar network. It became operational in 1991.

EM-4 was an extension of the Sigma-1. It was designed for 1024 nodes, but only an 80-node prototype became operational in 1990. The prototype achieved 815 MIPS in an 80×80 matrix multiplication benchmark. We will study the details of EM-4 in Section 9.5.2.

Hybrid and Unified Architectures These are architectures combining positive features from the von Neumann and dataflow architectures. The best research examples include the MIT P-RISC (Nikhil and Arvind, 1988), the IBM Empire (Iannucci et al., 1991), and the MIT/Motorola *T (Nikhil, Papadopoulos, Arvind, and Greiner, 1991).

P-RISC was a "RISC-ified" dataflow architecture. It allowed tighter encodings of the dataflow graphs and produced longer threads for better performance. This was achieved by splitting "complex" dataflow instructions into separate "simple" component instructions that could be composed by the compiler. It used traditional instruction sequencing. It performed all intraprocessor communication via memory and implemented "joins" explicitly using memory locations.

P-RISC replaced some of the dataflow synchronization with conventional program counter-based synchronization. IBM Empire was a von Neumann/dataflow hybrid architecture under development at IBM based on the thesis of Iannucci (1988). The *T was a latter effort at MIT joining both the dataflow and von Neumann ideas, to be discussed in Section 9.5.3.

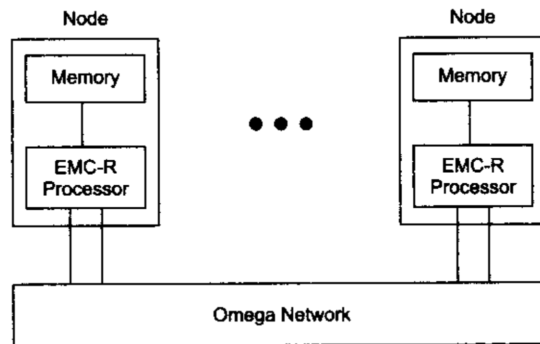
9.5.2 ETL/EM-4 in Japan

EM-4 had the overall system organization as shown in Fig. 9.32a. Each EMC-R node was a single-chip processor without floating-point hardware but including a switch of the network. Each node played the role of I-structure memory and had 1.31 Mbytes of static RAM. An Omega network was used to provide interconnections among the nodes.

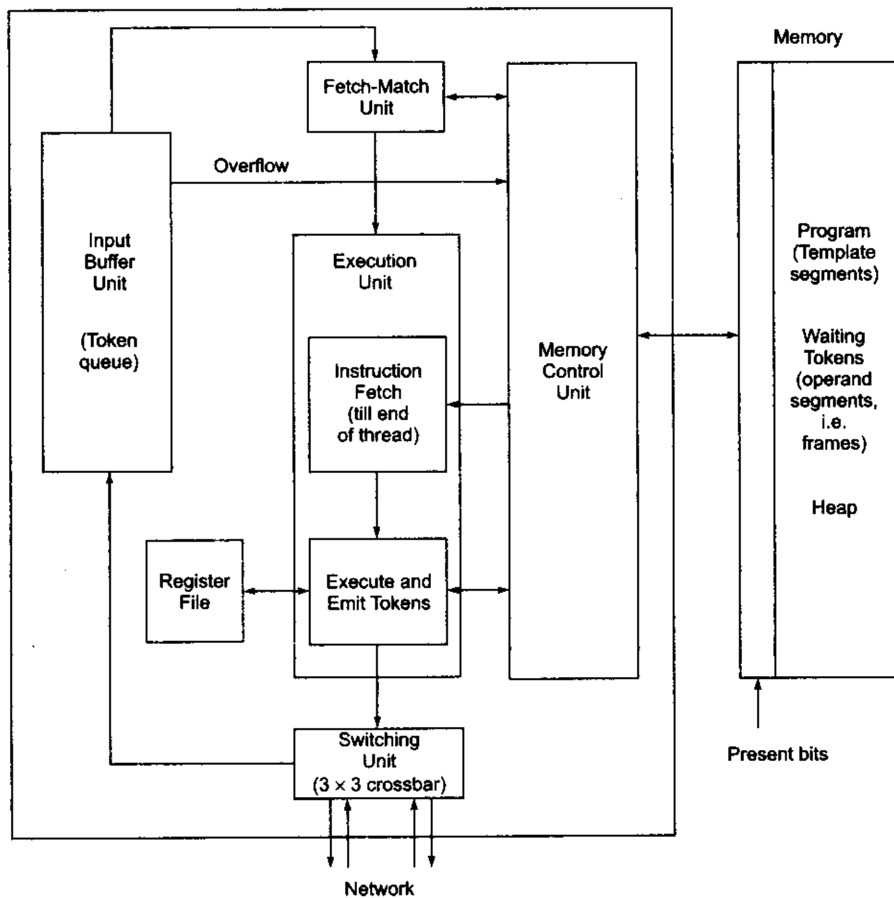
The Node Architecture The internal design of the processor chip and of the node memory are shown in Fig. 9.32b. The processor chip communicated with the network through a 3×3 crossbar *switch unit.* The processor and its memory were interfaced with *a memory control unit.* The memory was used to hold programs (template segments) as well as tokens (operand segments, heaps, or frames) waiting to be fetched.

The processor consisted of six component units. The *input buffer* was used as a token store with a capacity of 32 words. The *fetch-match unit* fetched tokens from the memory and performed tag-matching operations among the tokens fetched in. Instructions were directly fetched from the memory through the memory controller.

The heart of the processor was the *execution unit,* which fetched instructions until the end of a thread. Instructions with matching tokens were executed. Instructions could emit tokens or write to registers. Instructions were fetched continually using traditional sequencing (PC + 1 or branch) until a "stop" flag was raised to indicate the end of a thread. Then another pair of tokens was accepted. Each instruction in a thread specified the two sources for the next instruction in the thread.



(a) Global organization



(b) The EMC-R processor design

Fig. 9.32 The ETL EM-4 dataflow architecture (Courtesy of Sakai, Yamaguchi et al, Electrotechnical Laboratory, Tsukuba, Japan, 1991)

The same idea was used as in Monsoon for token matching, but with different encoding. All data tokens were 32 bits, and instruction words were 38 bits. EM-4 supported remote loads and synchronizing loads. The *full/empty* bits present in memory words were used to synchronize remote loads associated with different threads.

9.5.3 The MIT/Motorola *T Prototype

The *T project was a direct descendant of a series of MIT dynamic dataflow architectures unifying with the von Neumann architectures. In this final section, we describe *T, a prototype multithreaded MPP system based on the work of Nikhil, Papadopoulos, and Arvind of MIT in collaboration with Greiner and Traub of Motorola. Finally, we compare the dataflow and von Neumann perspectives in building fine-grain, massively parallel systems.

The Prototype Architecture The *T prototype was a single-address-space system. A "brick" of 16 nodes was packaged in a 9-in cube (Fig. 9.33a). The local network was built with 8 × 8 crossbar switching chips. A brick had the potential to achieve 3200 MIPS or 3.2 Gflops. The memory was distributed to the nodes. One gigabyte of RAM was used per brick. With 200-Mbytes/s links, the I/O bandwidth was 6.4 Gbytes/s per brick.

A 256-node machine could be built with 16 bricks as illustrated in Fig. 9.33b. The 16 bricks were interconnected by four switching boards. Each board implemented a 16 × 16 crossbar switch. The entire system could be packaged into a 1.5-m cube. No cables were used between the boards. The package was limited by connector-pin density. The 256-node machine had the potential to achieve 50,000 MIPS or 50 Gflops. The bisection bandwidth was 50 Gbytes/s.

The *T Node Design Each node was designed to be implemented with four component units. A Motorola superscalar RISC microprocessor (MC88110) was modified as a *data processor* (dP). This dP was optimized for long threads. Concurrent integer and floating-point operations were performed within each dP.

A *synchronization coprocessor* (sP) was implemented as an 88000 special-function unit (SFU), which was optimized for simple, short threads. Both the dP and the sP could handle fast loads. The dP handled incoming continuation, while the sP handled incoming messages, rload/rstore responses, and joins for messaging or synchronization purposes. In other words, the sP off-loaded simple message-handling tasks from the main processor (the dP). Thus the dP would not be disrupted by short messages.

The *memory controller* handled requests for remote memory load or store, as well as the management of node memory (64 Mbytes). The *network interface unit* received or transmitted messages from or to the network, respectively, as illustrated in Fig. 9.33c. It should be noted that the sP was built as an on-chip SFU of the dP.

The MC 88110 family allowed additional on-chip SFUs, with reserved opcode space, common instruction-issue logic and caches, etc., and direct access to processor registers. Example SFUs included the floating-point unit, graphics unit, coprocessor, etc. The MC 88110 was itself a two-way superscalar processor driven by a 50-MHz clock.

New SFUs were added into the MC 88110 to provide 16 buffers for incoming messages and 4 buffers for outgoing messages. Other SFUs included a *continuation stack* with 64 entries and a *microthreaded scheduler*, which supplied continuations from messages and the continuation stack, etc. Special instructions were available for packing or unpacking continuations.

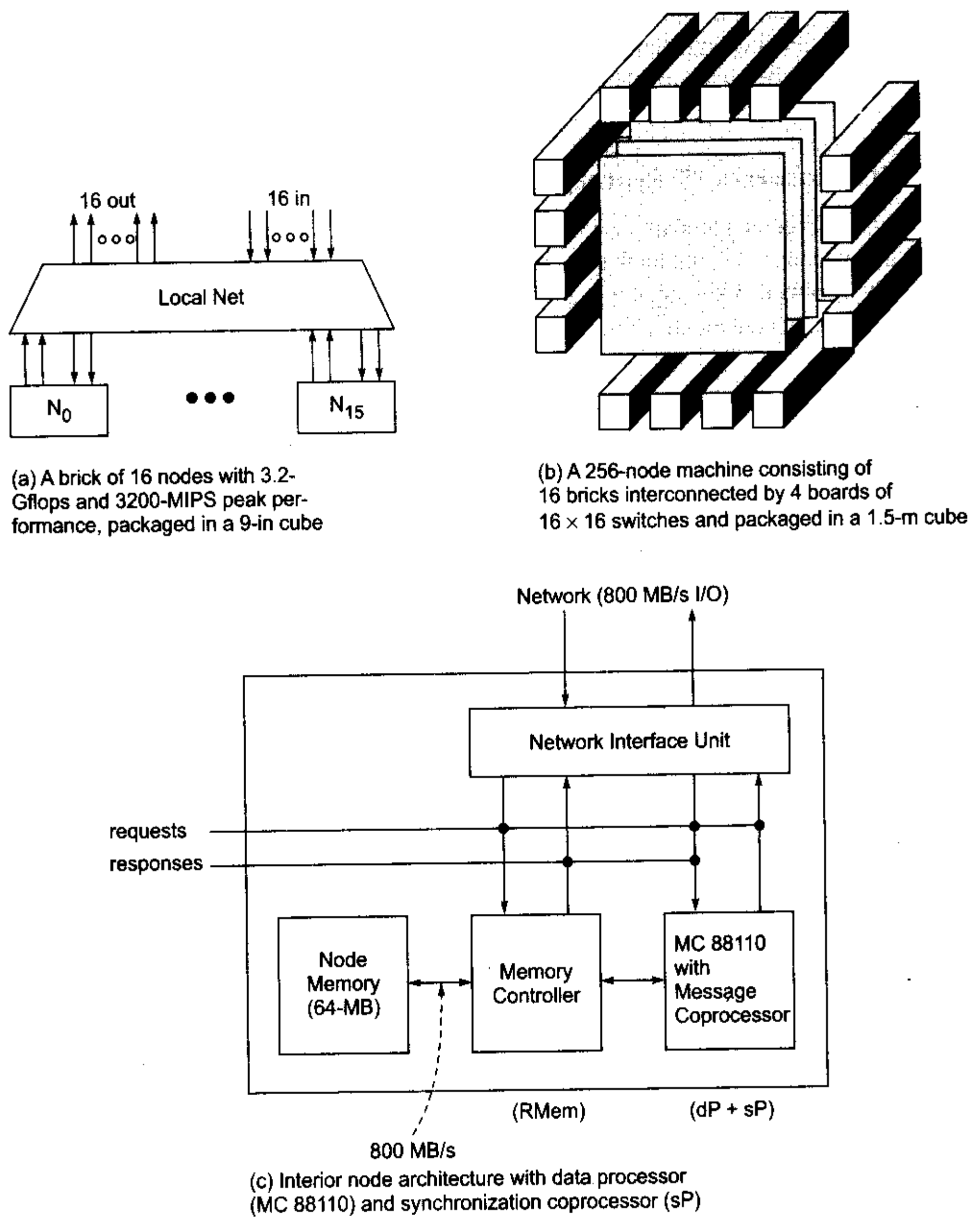


Fig. 9.33 The MIT/Motorola *T prototype multithreaded architecture (Courtesy of Nikhil, Papadopoulos, and Arvind, *Proc. 19th Int. Symp. Computer Arch.*, Australia, May 1992)

Research Experiments The *T prototype was used to test the effectiveness of the unified architecture in supporting multithreading operations. The development of *T was influenced by other multithreaded architectures, including Tera, Alewife, and J-Machine.

The I-structure semantics was also implemented in *T. Full/empty bits were used on producer-consumer variables. *T treated messages as virtual continuations. Thus busy-waiting was eliminated. Other optimizations in *T included speculative avoidance of the extra loads and stores through multithreading and coherent cacheing.

The *T designers wanted to provide a superset of the capabilities of Tera, J-Machine, and EM-4. Compiler techniques developed for these machines were expected to be applicable to *T. To achieve these goals, a promising approach was to start with declarative languages while the compiler could aim to extract a large amount of fine-grain parallelism.

Multithreading: A Perspective The Dash, KSR-1, and Alewife leveraged existing processor technology. The advantages of these directory-based cacheing systems include compatibility with existing hardware and software. But they offer a less aggressive pursuit of parallelism and depend heavily on compilers to obtain locality. The synchronizing loads are still problematic in these distributed cacheing solutions.

In von Neumann multithreading approaches, the HEP/Tera replicated the conventional instruction stream. Synchronizing-loads problems were solved by a hardware trap and software. Hybrid architectures, such as Empire, replicated conventional instruction streams, but they did not preserve registers across threads. The synchronizing loads were entirely supported in hardware. J-Machine supported three instruction streams (priorities). It grew out of message-passing machines but added support for global addressing. Remote synchronizing loads were supported by software convention.

In the dataflow approaches, the system-level view has stayed constant from the Tagged-Token Dataflow Architecture to the *T. The various designs differ in internal node architecture, with trends toward the removal of intra-node synchronization, using longer threads, high-speed registers, and compatibility with existing machine codes. The *T designers claimed that the unification of dataflow and von Neumann ideas would support a scalable shared-memory programming model using existing SIMD/SPMD codes.

Summary

Computer systems have always operated with processors having much faster cycle times than main memories. With steady advances in VLSI technology over the years, both processors and main memories have become faster, but the relative speed mismatch between them has in fact widened over the years. *Latency hiding techniques* are therefore devised to allow processors to operate at high efficiency in spite of having to access slower memories from time to time; use of cache memories is a common latency hiding technique. In the context of Massively Parallel Processing (MPP) systems, other technical challenges also confront system designers in minimizing the impact of memory access latencies.

In this chapter, we studied some basic latency hiding techniques applicable to such systems, namely: shared virtual memory with some specific examples; prefetching techniques and their effectiveness; and the use of distributed coherent caches. Scalable Coherent Interface (SCI) provides cache coherence with distributed directories and sharing lists. We studied several relaxed memory consistency models which can permit greater exploitation of parallelism in applications; the impact of relaxed consistency models while running three specific applications was presented.

Principles of multi-threading were introduced, with specific attention paid to the technical factors relevant to system design, namely: communication latency on remote access, number of threads, context-switching overhead, and the interval between context switches. Multiple context processors have been designed to provide hardware support for single cycle context switching. Possible context-switching policies were studied, along with their impact on system efficiency. Multidimensional architectures were reviewed as a possible platform for multi-threaded systems.

Fine-grain multicomputers are specially designed to provide efficient support for fine-grain parallelism in applications. The MIT J-machine was studied from the points of view of its overall system design, its Message-Driven Processor (MDP) and instruction set architecture, and the message format and routing employed in its 3-dimensional mesh. The design goal of Caltech Mosaic C system was to exploit the advances which had taken place in VLSI and packaging technologies; we studied the basic node design with its two contexts (for user program and message handler), and basic 8×8 mesh design employed in the system.

In the category of scalable multithreaded architectures, the Stanford Dash multiprocessor system utilized directory-based cache coherence in a single address-space distributed memory system. Kendall Square Research KSR-1 system employed a cache-only memory design with a ring-based interconnect. The Tera multiprocessor system relied for its performance on a large degree of multi-threading and aggressive use of pipelining throughout the system, with a sparse 3-dimensional torus interconnect.

We also studied the basic concepts and evolution of dataflow and hybrid architectures, from the first introduction of the concept in 1974 by Jack Dennis at MIT. Specific dataflow and hybrid systems studied in this context were the ETL/EM-4 system developed in Japan, and the MIT/Motorola *T prototype system.

Exercises

Problem 9.1 Consider a scalable multiprocessor with p processing nodes and distributed shared memory. Let R be the rate of each processing node generating a request to access remote memory through the interconnection network. Let L be the average latency for remote memory access. Derive expressions for the processor efficiency E under each of the following conditions:

(a) The processor is single-threaded, uses only a private cache, and has no other latency-hiding mechanisms. Express E as a function of R and L.

(b) Suppose a coherent cache is supported by hardware with proper data sharing and h is the probability that a remote request can be satisfied by a local cache. Express E as a function of R, L, and h.

(c) Now assume each processor is multithreaded to handle N contexts simultaneously. Assume a context-switching overhead of C. Express E as a function of N, R, L, h, and C.

(d) Now consider the use of a 2-D $r \times r$ torus with $r^2 = p$ and bidirectional links. Let t_d be the time delay between adjacent nodes and t_m be the local memory-access time. Assume that the network is fast enough to respond to each request without buffering. Express the latency L as a function of p, t_d, and t_m. Then express the efficiency E as a function of N, R, h, C, p, t_d, and t_m.

Problem 9.2 The following two questions are related to the effect of prefetching on latency tolerance:

(a) Perform an analytical study of the effects of data prefetching on the performance (efficiency) of processors in a scalable multiprocessor system without multithreading.

(b) Repeat part (a) for a multithreaded multiprocessor system under reasonable assumptions.

Problem 9.3 The following questions are related to the effects of memory consistency models:

(a) Perform an analytical study of the effects of using a relaxed consistency memory model in a scalable multiprocessor without multithreading.

(b) Repeat part (a) for a multithreaded multiprocessor system under reasonable assumptions.

(c) Can you derive an efficiency expression for a multiple-context processor supported by both prefetching and release memory consistency?

Problem 9.4 Consider a two-dimensional multicube architecture with m row buses and m column buses (Fig: 9.18a). Each bus has a bandwidth of B bits/s. The bus is considered active when it is actually in progress. The bus utilization rate a ($0 < a \leq 1$) is defined as the number of active bus cycles over the total cycles elapsed. The per-processor request rate r is defined as the number of requests that a processor sends on either of the two buses (for the purpose of memory access, cache coherence, synchronization, etc.) per second.

(a) Consider a single-column bus with associated processors and memory module and express the bus bandwidth as a function of m, a, and r.

(b) What is the total bus bandwidth available in the entire system?

(c) If r is kept constant as the number of processors increases, how many requests can be sent to the system without exceeding the limit?

(d) Each request goes through a maximum of two buses in the multicube. What bus bandwidth will be needed to satisfy all the requests?

(e) In parts (b) and (d), does the multicube provide enough bus bandwidth? Justify the answer with reasoning.

Problem 9.5 Consider the use of an orthogonal multiprocessor consisting of 4 processors and 16 orthogonally shared memory modules (Fig. 9.18b) to perform an unfolded multiplication of two 8×8 matrices in a partitioned SPMD mode.

(a) Show how to distribute the 2×2 submatrices of the input matrix $A = (a_{ij})$ and $B = (b_{ij})$ to the 16 orthogonally shared memory modules.

(b) Specify the SPMD algorithm by involving all four processors in a synchronized manner to access either the row memories or the column memories. Synchronization is handled at the loop level.

You can assume the use of a pipeline-read to fetch either one column or one row vector of the input matrix A or B at a time, and a pipeline-write to store the product matrix $C = A \times B = (c_{ij})$ elements in a similar fashion. Assume that sufficient large register windows are available within each processor to hold all 2×2 submatrix elements. Each processor can perform inner product operations.

(c) Let $N \times N$ be the matrix size and $k = N/n$ the partitioned block size in mapping a large matrix in the orthogonal memory. Estimate the number of orthogonal memory accesses and the number of synchronizations needed in an SPMD algorithm for multiplying two $N \times N$ matrices on an n-processor OMP.

(d) Repeat the above for a two-dimensional fast Fourier transform over $N \times N$ sample points on-an n-processor OMP, where $N = n \cdot k$ for some integer $k \geq 2$. The idea of performing a two-dimensional FFT on an OMP is to perform a one-dimensional FFT along one dimension in a row-access mode.

All n processors then synchronize, switch to a column-access mode, and perform another one-dimensional FFT along the second dimension. First try the case where $N = 8$, $n = 4$, and $k = 2$ and then work out the general case for large $N \gg n$.

Problem 9.6 The following questions are related to shared virtual memory:

(a) Why has shared virtual memory (SVM) become a necessity in building a scalable system with memories physically distributed over a large number of processing nodes?

(b) What are the major differences in implementing SVM at the cache block level and the page level?

Problem 9.7 The release consistency (RC) model has combined the advantages of both the processor consistency (PC) and the weak consistency (WC) models. Answer the following questions related to these consistency models:

(a) Compare the implementation requirements in the three consistency models.

(b) Comment on the advantages and shortcomings of each consistency model.

Problem 9.8 Answer the following questions involving the MIT J-Machine:

(a) What were the unique features of the message-driven processors (MDP) making it suitable for building fine-grain multicomputers?

(b) Explain the E-cube routing mechanism built into the MDP.

(c) Explain the concept of using a combining tree for synchronization of events on various nodes in the J-Machine.

Problem 9.9 Why are hypercube networks (binary *n-cube* networks), which were very popular in first-generation multicomputers, being replaced by 2D or 3D meshes or tori in the second and third generations of multicomputers?

Problem 9.10 Answer the following questions on the SCI standard:

(a) Explain the sharing-list creation and update methods used in the IEEE Scalable Coherence Interface (SCI) standard.

(b) Comment on the advantages and disadvantages of chained directories for cache coherence control in large-scale multiprocessor systems.

Problem 9.11 Compare the four context-switching policies: switch on cache miss, switch on every load, switch on every instruction (cycle by cycle), and switch on block of instructions.

(a) What are the advantages and shortcomings of each policy?

(b) What additional research would be needed to make an optimal choice among these policies?

Problem 9.12 After studying the Dash memory hierarchy and directory protocol, answer the following questions with an analysis of potential performance:

(a) Define the cache states used in Dash.

(b) How were the cache directories implemented in the memory hierarchy?

(c) Explain the Dash directory-based coherence protocol when reading a remote cache block that is dirty in a remote cluster.

(d) Repeat part (c) for the case of writing to a shared remote cache block.

Problem 9.13 Answer the following questions on multiprocessors:

(a) Describe the ALLCACHE architecture implemented in the Kendall Square Research KSR-1.

(b) Explain how cache coherence can be maintained in the KSR-1.

(c) Study the papers on COMA architectures by Stenström et al (1992) and Hagersten et al (1990). Compare the differences between KSR-1 and the Data Diffusion Machine (DDM) architecture.

Problem 9.14 Answer the following questions on the development of the Tera computer.

(a) What were the design goals of the Tera computer?

(b) Explain the sparse 3D torus used in Tera. What are the advantages of the sparse structure?

(c) Explain how pipelining is applied in supporting the multithreaded operations in each Tera processor.

(d) Explain the thread state and management scheme used in Tera.

(e) Explain the idea of explicit-dependence lookahead and its effects on multithreading in Tera.

(f) What are the contributions of the Tera architecture and software development? Compare the advantages and potential drawbacks of the Tera computer.

Problem 9.15 Answer the following questions related to dataflow computers:

(a) Distinguish between static dataflow computers and dynamic dataflow computers.

(b) Draw a dataflow graph showing the computations of the roots of a sequence of quadratic equations $A_i x_i^2 + B_i x_i + C_i = 0$ for $i = 1, 2, \ldots, N$.

(c) Consider the parallel execution of the successive root computations with a four-PE tagged-token dataflow computer (Fig. 2.12). Show a minimum-time schedule for using the four PEs to compute the N pairs of roots.

Problem 9.16 Consider the mapping of a one-dimensional circular convolution computation on a multiprocessor with 4 processors and 32 memory modules which are 32-way interleaved for pipelined access of vector data. Assume no contention between processors and memories in the interconnection network. The one-dimensional convolution is defined over a $1 \times n$ image and a $1 \times m$ kernel as follows:

$$Y(i) = \sum_{j=0}^{m-1} W(j) \cdot X((i-j) \bmod n) \text{ for } 0 \le i \le n-1$$

(a) How many multiplications and additions are involved in the above computations? Map the image pixels $X(i)$ to memory module M_j if $j = i \pmod{32}$ and assume $n = 256$. The output image $Y(i)$ is also stored in module M_j if $j = i \pmod{32}$ for $0 \le i \le 255$. The kernel is also stored in a similar manner. Assume $m = 4$ and each processor handles the computation of one output image.

(b) Show how to partition the computations among the four processors such that minimum time is spent in both memory-access and CPU executions. Assume no memory conflicts and up to four fetch or store operations (but not mixed) performed at the same time. The interleaved memory can be accessed by one or more processors at the same time.

(c) What is the minimum execution time (including both memory and CPU operations) if each multiply and add and each interleaved memory access is considered one time unit. Assume enough working registers are available in each CPU.

(d) What is the speedup factor of the above multiprocessor solution over a uniprocessor solution? You can make similar assumptions about the use of the 32-way interleaved memory for both uniprocessor and multiprocessor configurations.

Problem 9.17 Answer the following questions on fine-grain multicomputers and massive parallelism:

(a) Why are fine-grain processors chosen for

research-oriented multicomputers and MPP systems over medium-grain processors used in the past?

(b) Why is a single global addressing space desired over distributed address spaces?

(c) From scalability point of view, why is fine-grain parallelism more appealing than medium-grain or coarse-grain parallelism for building MPP systems?