[UW84] E. Upfal and A. Wigderson. How to share memory in a distributed system. In *25th Annual Symposium on Foundations of Computer Science*, 1984. 171-180.

[VB81] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *13th Annual Symposium on Theory of Computing*, 1981. 263-277.

[KR88] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report 408, (EECS) University of California, Berkeley, 1988.

[KRS88] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *15th International Colloquium on Automata, Languages and Programming*, 1988. 333-346.

[KS89] P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. In *8th Annual ACM Symposium on Principles of Distributed Computing*, 1989.

[KU86] A.K. Karlin and E. Upfal. Parallel hashing — an efficient implementation of shared memory. In *18th ACM Symposium on Theory of Computing*, 1986. 160-168.

[Lam78] L. Lamport. Times, clocks and ordering of events in a distributed system. *CACM*, 21(7), 1978. 558-565.

[LG89] N. Lynch and K.J. Goldman. Distributed algorithms. Technical report, MIT, 1989.

[LL89] L. Lamport and N. Lynch. Distributed computing. To appear in TCS, 1989.

[LM88] C.E. Leiserson and B. Maggs. Communication-efficient parallel algorithms for distributed random access machines. *Algorithmica*, 3, 1988. 53-77.

[OCP88] Opportunities and constraints in parallel computing, December 1988. Position papers for Workshop at IBM Almaden.

[PN85] G.F. Pfister and V.A. Norton. "Hot-spot" contention and combining in multistage interconnection networks. In *International Conference on Parallel Processing*, August 1985. 790-797.

[PU87] C.H. Papadimitriou and J.D. Ullman. A communication-time trade-off. *SIAM Journal on Computing*, 16(14), 1987.

[PY88] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *20th Annual Symposium on Theory of Computation*, May 1988. 510-513.

[Qui87] M. Quinn. *Designing Efficient Algorithms for Parallel Computers*. Prentice-Hall Inc., 1987.

[Ran87] A.G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, 1987. 185-194.

[RK89] V.N. Rao and V. Kumar. Analysis of scalability of parallel algorithms. To appear as UTCS-TR, 1989.

[RT86] R. Rettberg and T. Thomas. Contention is no obstacle to shared-memory multiprocessing. *CACM*, 29(12), 1986. 1202-1212.

[Sei84] C.L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, 33(12), 1984. 1247-1265.

[Sny86] L. Snyder. Type architectures, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1, 1986. 289-317.

[SS86] Y. Saad and M.H. Schultz. Data communication in parallel architectures. Technical Report YALEU/DCS/RR-461, Yale University, Computer Sciences, 1986.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM TOPLAS*, 10(2), 1988. 282-312.

[Ull84a] J.D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[Ull84b] J.D. Ullman. Some thoughts about supercomputer organization. In *IEEE COMPCON*, 1984.

References

[AA82] T. Agerwala and Arvind. Data-flow systems: Guest editor's introduction. *IEEE Computer*, 15(2), February 1982. 10-13.

[AC88] A. Aggarwal and A. Chandra. Communication complexity of PRAMs. In *15th International Colloquium on Automata, Languages and Programming*, 1988. 1-18.

[ACS87] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, 1987. 204-216.

[ACS89] A. Aggarwal, A. Chandra, and M. Snir. On the communication latency in PRAM computations. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1989.

[AG89] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Series in Computer Science and Engineering. Benjamin/Cummings, 1989.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic of idealized parallel computers on more realistic ones. *SIAM Journal of Computing*, 1987. 808-835.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[And89] R. Anderson. On the implementation and performance of algorithms for small shared-memory machines. Technical report, University of Washington at Seattle, 1989.

[AS88] W.C. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, August 1988. 9-24.

[Ath87] W.C. Athas. Fine-grain concurrent computation. Technical Report 5242:TR:87, California Institute of Technology, 1987.

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tre, leader election and related problems. In *19th Annual Symposium on Theory of Computation*, 1987. 230-240.

[CM88] K.M. Chandy and J. Misra. *A Foundation for Parallel Programming.* McGraw Hill Publishers, 1988.

[Coo85] S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64, 1985. 2-22.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1989. 169-178.

[Den80] J.B. Dennis. Data-flow supercomputers. *IEEE Computer*, 13, November 1980. 48-56.

[Fei88] D. Feitelson. *Optical Computing: A Survey for Computer Scientists.* MIT Press, 1988.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[Gib89] P.B. Gibbons. A more practical PRAM model. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1989. 158-168.

[GR88] A.M. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[Hil85] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[KLM+89] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving simulations of fixed-connection networks. Submitted for publication, 1989.

networks, it is probably essential as well. This is because it is very unlikely that non-uniformities that can cause hot-spots will entirely be eliminated. The problem is, of course, that using sophisticated switches renders them quite complex and their cost begins to approach the cost of the actual processing elements.

10.7 Small-scale Parallelism and PRAM Computations

Research in [And89] exhibits several systems aspects that need to be considered for implementing PRAM algorithms on real machines with a small number of parallel processors. The paper describes the results of implementing parallel merge sorting, and Prim's spanning-tree algorithm on a 20-processor Sequent Symmetry. The motivation was to spur theoretical research that has impact on programming aspects of real machines.

Several general observations are made in the paper. Firstly, it is observed that there is a necessity to change the granularity of the tasks so as to offset the effects of the numerous overheads. Next, it is observed that achieving high speed-ups asymptotically is inadequate for real-world problem sizes. A requirement suggested for the efficient utilization of algorithms is that the algorithms converge quickly (in terms of problem size) to the high speed-up values. It is also suggested that algorithms be re-designed to take into consideration the effects of the overheads such as locking, synchronization etc.. Finally, it is observed that not all algorithm domains are susceptible to encouraging speed-ups — graph algorithms appear to be especially intransigent.

11 Conclusions

To be able to realize useful and usable parallel systems, several different areas have to be studied in conjunction. Since a typical parallel system is quite complex, it is imperative that models be available that abstract away from the real machines to provide a tractable framework in which ideas may be developed. Unfortunately, it is unclear which model or models may be suitable. Although every model is reasonable from the technical point-of-view, they may not be from the practical viewpoint. Eventually, the approaches settled upon will actually depend more on non-technical factors such as social acceptance, the willingness to accept poorer performance, and the economics of building the systems.

systems usually fall into the categories of master-slave, separate supervisor or symmetric organizations. In the context of process creation and pre-emption, there has been considerable interest recently in lightweight processes called threads (e.g., in the Mach OS). These are processes that carry a very small amount of the state of the system so that context switching becomes much faster.

10.4 Some Specific Papers

The [SS86] paper studies algorithms to effect different communication modes (one-to-all, all-to-one, all-to-all, one-to-one, scatter-gather, multiscatter-multigather) using some actual performance metrics for different types of computers. Although it is customary to use only a few types of communication modes in algorithm development for theoretical models (e.g., one-to-all and one-to-one in shared memory and distributed memory models), in actual programming practice, it pays in terms of efficiency to use all the these techniques — often the time taken to implement a seemingly less complicated mode of communication is the same as that for a more complex one (see the scan model in Section 5). The paper provides parametrized performance measures for the ring, grid, hypercube, and switch architectures. The efficiency criteria depend on the specific characteristics of the architectures. The results are available in a tabulated form in the paper.

10.5 Contention

Contention for shared resources has been extensively studied for a long time. However, parallel systems introduce several new avenues of increased contention since several processors are involved — unlike conceptually concurrent processes. Examples are contention for memory bandwidth, communication bandwidth etc.. The [RT86] paper indicates that balancing of the load in several aspects, is of paramount importance for the efficient utilization of the resources. For example, by the even distribution of data, the contention to access a single memory module is reduced (this has been observed in a more theoretic framework in Section 6). The availability of multiple redundant paths between processors and memory modules with an equitable distribution of paths to processors also alleviates the problem.

In fact, the question of balancing pervades the whole area of parallel systems — distribute the load to speed-up overall system performance. The results of [RT86] are not surprising; they only confirm the expected behavior. However, the details of achieving optimal load balancing, or even approaching an optimal balance, is difficult.

10.6 Hot Spots

As mentioned above, a specific instance of contention is that involving the access of a single memory module by several processors. In a multistage interconnection network, this phenomenon can cause a rapid and severe decline of system performance as described in [PN85]. The paper makes reasonable assumptions that the routing control for the messages is distributed and that the buffer space at the switches is bounded.

Simulation studies on multistage networks exhibited that in the absence of combining networks, the requests at the contended memory modules fill-up the queue space quickly. Furthermore, the exhaustion of buffer space at a particular switch has a cascading effect on the switches that precede it in the routing: the next stage of switches also exhaust their buffers. In this manner, in a short period of time, all the switches — beginning from the contended memory unit upto the processors — fanning-out in a tree-like fashion, have their buffers filled-up. This obviously degrades the performance, not only with respect to the contended unit, but also all the other memory units. This happens because several of the switches that are blocked play a part in the routing of other messages. This phenomenon is of great concern and is, unfortunately, quite general: it is independent of a specific network topology, packet or circuit switching modes, and shared or distributed[20] memory systems.

On the other hand, the simulation studies also showed that combining networks that are built using sophisticated switches alleviate the problem effectively. Thus, it is not only desirable to have combining

[20] Message passing systems often do not have switches, but the processing elements themselves provide the functions of switching. Since the processors themselves are certainly more sophisticated as compared to switching elements, incorporating hardware combining should not affect the costs overly.

passing models have CSP, Ada, and Occam for their programming notations (among several others). Pattern driven notations include Prolog for shared-memory models and Actors for message-passing models. Data-driven notations have examples in Val, Id, LAU, and Sisal — all for the message-passing situations. Finally, FP is a notation for message-passing models that exhibit the characteristics of a demand driven notation.

Pattern driven models rely on the use of pattern comparison and search techniques. Logic programming, a branch of study that is closely related to both formal mathematics and computing, hides the algorithmic details from the programmer at the program level. However, the underlying system must make full use of *and-or* search mechanisms and string matching algorithms to efficiently implement the programs.

Data driven models embody the notations used for dataflow approaches [AA82]. Theoretically, they offer the maximum possible concurrency available based on dataflow graphs that essentially are a form of dependency relationships (see Section 7). The execution of a node, or task, of a dataflow graph depends only on the availability of the inputs. The dataflow approach is based on the fact that a large number of tasks are usually available for execution in parallel, and that the basic issue then rests on whether the processors can be made to find and execute the tasks that are ready. As can be expected, having queues for tasks that are ready, and the efficient access of pools of tasks that await input are among the prime considerations for such computers. It is, in fact, these practical issues that limit the full realization of dataflow approaches — which are otherwise very appealing conceptually [Den80].

Functional programming constitutes the demand driven approach. This is a mathematically appealing, side-effect free, easy to verify notation that depends on lazy evaluation strategies (hence the classification) to effect efficiency. As in pattern driven approaches, control flow details are also hidden from the programmer — but, in this case, the execution strategy is not supposed to affect the program semantics (e.g., consider the Prolog strategy). Based strongly on the lambda-calculus and function applications, this notation is appealing but difficult to implement efficiently.

In all the above control strategies, notice that differences in the translation (i.e., compilation) process are implicit. The smaller control provided to the programmer implies that the translation procedure is depended upon to extract the necessary parallelism. In fact, several translators for parallel machines are actually applied to programs written initially for sequential machines so as to extract the parallelism without having to redo the programming. However, the algorithms that are likely to be encoded in such programs may be optimized for sequential machines (i.e., the algorithms involved are sequential) and so may not yield large amounts of parallelism. Some of the notations are actually sequential ones with parallel constructs that augment them to assist the translators in the task of parallelization. The best approach to take is to use notations that are designed for parallel machines specifically.

10.2 Translators

Translators for SIMD machines have enjoyed a fair degree of success. The major source of parallelism in such cases is the parallel execution of loop statements — and these are comparatively easily effected. Several commercial machines with SIMD organizations routinely provide such parallelizing translators (usually as compilers).

For the more general cases of extracting parallelizable tasks, dependency graphs of the programs are analyzed. This is followed by the scheduling of processors effectively on the tasks so that the overheads of communication etc. are minimized. The scheduling of processors is non-trivial and several restricted cases have been examined in theoretical and empirical studies recently. This particular stage of effectively assigning the processors to the tasks is relegated to the operating system since the processing time is essentially a resource.

10.3 Operating Systems

The operating systems allocate the computer resources and provide an easy interface to the machine for the users. As in sequential computing, operating systems for parallel machines must handle the creation of processes, their destruction and scheduling, memory management, I/O and file systems, concurrency control etc.. Not only are the machines that are controlled by these programs very complex, but the controlling program itself executes on a parallel machine. Research in this area is only at its inception. Prototype

Parallel computing systems are far more intricate as compared to sequential ones, and it is clear that systems work occupies a position of particular importance. Fortunately, the research involved is not entirely new. Given that all the different peripherals together with the processing elements in a sequential system run concurrently, systems research has at least conceptually dealt with parallelism for a long time. Though systems research has not been geared towards parallel processor systems specifically, several features from the sequential domain find important applications.

10.1 Programming Notations and Constructs

Parallel programming notations require the ability to define processes, to start and stop them, and to coordinate their activities. Well-known constructs to achieve these goals exist as integral primitives in several notations. These include parbegin-parend, fork-join, doall, process declaration, test-and-set, fetch-and-op, semaphores, barriers, send-receive, rendezvous, and RPCs, and are covered in most standard systems texts. Many are used for sequence control (ensuring that requisite events logically occur in sequence), access control (permitting access to certain entities such as shared variables to only a few processes), several are used to synchronize processes etc.. Many of these constructs are involved in making certain sequences of execution atomic — a notion that is of especial significance in systems where several processes physically execute in parallel. Some may be non-blocking (taking a bounded number of time steps to execute), while others may not exhibit this desirable property. These constructs help in writing systems and application software easily, and they also help make the software portable across systems. However, note that several of these are logical primitives rather than machine supplied ones, and hence, their implementation introduces serious overheads in the algorithms they encode *over and above* the overheads implied in the algorithms themselves. This is true of any mechanism or scheme that hides machine details from users. Hence, it may be desirable to bear in mind this factor in the design of the algorithm before encoding in high level notations. Since several primitives have to be actually emulated by software techniques at the lower levels of system programs, and since the primitives are used quite frequently, hardware assistance, as described earlier, is resorted to.

The different types constructs, concurrently executing processes, and inherent non-determinism in several situations pose another significant challenge: How is the correctness of the programs to be assured? To cope with such added complexity, researchers have proposed several notations that incorporate means to verify program correctness. A major portion of distributed computing research is aimed at developing good models in which the correct development of complicated programs is made easier. For example, the development of correct programs by stepwise refinement of the precise specifications of the problem is an important research area [CM88, LL89, LG89]. Several methods among these are designed using notations that do not favor any particular architecture (e.g., [CM88]), and are suitably convertible to a specific machine organization. However, they usually do not include a rigorous measure of efficiency that may affect the choices made during the stepwise refinement process, and eventually, the efficiency of the implemented programs. Just as there are strong arguments to indicate that the more theoretical models for parallel computations that are aimed at providing efficiency measures of algorithms may be inadequate, arguments may be devised to indicate that implemented algorithms developed using formal techniques may well prove very inefficient. However, these models do allow the measurement of performance issues such as the total number of messages etc. which does measure efficiency to a certain degree in cases that the cost of message transmission is overwhelmingly important. Other approaches in programming notations seek to reduce programmer-induced errors by reducing the degree to which concurrency issues are handled by the programmers. Thus, the programmer is required in such situations to indicate *what* is required of the programs as opposed to *how* to achieve what is required (e.g., consider declarative notations). The underlying system converts these specifications into algorithms that actually execute (i.e., the specifications are executed) — and clearly, the efficiency of this approach depends strongly on whether the system is able to do its task well. Note that the development of correct and efficient programs is not obviated since the underlying system must itself have the necessary algorithms built-in.

Following the different philosophies, several types of notations are available to the parallel programmer depending on the level of explicit control they allow. In decreasing order of explicit control are control, pattern, data, and demand driven notations. The control driven types that are geared towards shared-memory architectures include Fortran, Concurrent Pascal, APL, Ada, Modula, Multilisp etc.; message-

allow the maximum allowable parallelism to be realizable. Justification is provided that this is not necessarily wasteful in processor utilization.

A schedule to execute the DAG is clearly restricted by the following: If node i is executed at time t on processor p, then every node j from which a dependency arc to i exists in the DAG, must have been executed by time $t-1$ on p or by $t-\tau-1$ on some other processor. It is an NP-complete problem to decide, given a DAG, an integer τ, and deadline time T_{max}, whether there exists a schedule with a time no greater than T_{max}. Thus, finding an optimal schedule is computationally intractable for an arbitrary DAG. Hence, the paper offers a means by which a good approximate schedule is implied.

Given a DAG G=(V, E), and an integer τ, the following function $e : V \rightarrow \omega$, is computed as follows on the set V:

- $e(v) = 0$ if v is a source.
- otherwise: Let all the predecessors u_i of v be ordered in non-increasing order of e (i.e., $e(u_1) \geq e(u_2) \geq \ldots \geq e(u_m)$). If $k = \min(\tau + 1, m)$, then $e(v) = \max(e(u_1) + 1, e(u_2) + 2, \ldots, e(u_k) + k)$.

The time taken for an optimal schedule, $opt(v)$, for a node v, obeys the relation $opt(v)/2 \leq e(v) \leq opt(v)$. A more involved algorithm, which is in P, can be used to evaluate the function e on the nodes of a DAG even if the execution times of the nodes are given as functions dependent on the nodes (i.e., $x : V \rightarrow \omega$), and the delay is also provided similarly (i.e., $\tau : V \rightarrow \omega$). Further approximations are also available for communication delays that are based on the arcs rather than the vertices. Thus, the values of e for the result nodes may now be used as a close approximation of the possible performance characteristics of an algorithm on architectures that have τ as their communication delay, assuming that reasonably good schedules are found.

The function e implicitly provides an assignment of processors to nodes, although the value of e itself need not correspond to the time that the node is actually evaluated. Let us examine this in the simplest case of the fixed communication delays. Note that an unlimited number of processors is available and the network is assumed to be synchronous. Also, it is assumed that whenever required, a processor can broadcast, in one step, all the values that it has evaluated to all processors that may need them with τ as the transit time required for these values to reach their destinations. Firstly, all the source nodes are assigned different processors that compute their values in parallel, simultaneously. Inductively, all the nodes that need to be evaluated before the current one are assumed to have been evaluated already. However, consider the number m of such nodes. If they number less than $\tau + 1$, it makes sense to (re)compute them at the same place that the current node is being evaluated since the communication overhead would delay the execution if they were evaluated elsewhere. Lacking more information, the node u_1 is evaluated before the others, followed by u_2, and so on since the idea is to get low values for $e(v)$. The remaining nodes u_i are obtained from other processors that have evaluated them. Notice that there is a large possibility that a fair amount of wasteful recomputation of the nodes may occur.

The method described is applied to DAGs that are binary trees, the FFT and the pyramid; the latter is very similar to the diamond DAG of [PU87]. The results obtained are very good and are optimal in some cases. Notice that the DAGs on which the method has been applied are very well structured. Among the cases worth considering are those where the DAG is not well structured.

10 Systems Aspects

Experience with sequential computing indicates that simply using some fast architectures in conjunction with a set of fast algorithms cannot make the use of computers efficient. Even sequential computers, when considered in detail, are quite complicated, and it is hopeless to expect that computer users can cope with all the particulars. Systems-oriented work helps to alleviate these problems by presenting a more tractable entity to the users. This is not only done by hiding several details but also by taking-up a large amount of the tasks that require in-depth knowledge about the system (e.g., task scheduling etc.). This is the realm of systems programming. In this section, systems work is taken to include programming notations, associated compilers, and operating systems.

nodes provides a lower bound on the number of communication arcs, and in the case of constant out-degree graphs such as the diamond DAG, this method provides a tight lower bound.

Let us examine an easy example of communication-time tradeoffs with which this paper is concerned. Consider a four-node diamond DAG G=(V, E), with $V = \{a, b, c, d\}$ and $E = \{(a, b), (b, d), (a, c), (c, d)\}$. In a two-processor environment, if one of them computes all the nodes, the time taken is 4 units and the communication overhead is 0. If only 3 time units are permitted, the first processor can evaluate nodes a, b, d during the 3 units of time while the second processor evaluates node c in the time slot 2. This results in 2 units of communication within the 3 time units permitted. By allowing recomputation, if the second processor evaluates node a in the first time slot as well, the communication overhead is reducible to 1. Notice that the depth of the DAG implies that the total time taken will be at least 3 time units.

The communication-time tradeoffs of this nature are examined closely for a diamond DAG. Unlike some simpler DAGs, such as the binary tree, the diamond DAG is shown not to have the property that the best lower bounds on the total communication, c, and the total time, t, are simultaneously realizable. The first result exhibited is: Any schedule and assignment of processors for the evaluation of an $n \times n$ diamond DAG requires communication c and time t such that $(c + n)t = \Omega(n^3)$. The term n occurs on the left side of the equation to account for the case that a single processor is involved. A sequence of lemmas, each of which is easy, proves the theorem. However, the proofs rely heavily on the particular geometric and graph theoretic properties of the diamond DAG. These factors limit the wider applicability of the proof techniques, although several other specific graphs have been examined. The common methods of scheduling processors to evaluate the diamond DAG are described in the light of this result.

The second result presented relates to the time-communication delay (denoted by d) of the diamond DAG. It is shown that $(d + 1)t = \Omega(n^2)$ where the unit term on the left accounts for the sequential case. In the simple case of scheduling where no recomputation is permitted, the proof follows quite easily from the previous result. This is because all the nodes of the DAG can be covered by $O(n)$ source-sink paths, and the $\Omega(n^3/t)$ communication nodes in the graph (proved earlier) imply that at least one path has $\Omega(n^2/t)$ communication nodes. However, when recomputation is permitted, a different D relation is required which is not conducive to a similar analysis. Again, using specific geometric properties of the DAG, the theorem is proved.

It is necessary to interpret the usefulness of these results. Depending on the actual communication characteristics of the underlying architectures, the results impose limits on the number of processors that may be usefully utilized in the solution of the algorithm. If such methods can be further extended, theoretical studies in the scheduling of processors for parallel algorithms will benefit. Similar tradeoffs have been examined elsewhere as well. The paper mentions the $ct = \Omega(n^4)$ and $ct = \Omega(n)$ lower bound results for dynamic programming DAGs and binary tree DAGs, respectively. Results involving other characteristics of architectures, such as granularity of the messages and message start-up times etc. would be very useful, and some of these were discussed in Section 4.

Let us consider the related paper [PY88]. As in the earlier case, sequential algorithm implementation is regarded as the choosing an algorithm and then analyzing the performance. For the parallel case, the task is considered to proceed as follows:

- choose an algorithm DAG.
- choose the architecture on which it is to be executed.
- find a suitable execution schedule.
- analyze the performance by the time taken to compute the final result.

The first choice is presumed to have been made. To obviate the need to study a particular architecture, the authors have chosen to abstract-out a key feature of an architecture to parametrize the results to suit any architecture. The choice that has been made is the upper bound on the interprocessor communication time, τ since it is probably the single most important factor. The paper is concerned mainly with the final two steps suggested above. It integrates the communication factor τ as part of the scheduling. A simplifying assumption made in the paper is that there are a sufficient number of processors available to

An embedding $< \phi, \rho >$ of G = (V_G, E_G) into H = (V_H, E_H) is defined by an injective mapping $\phi : V_G \rightarrow V_H$ and a mapping $\rho : E_G \rightarrow E_H^*$. The mapping ρ maps $(u, v) \in E_G$ to a path between $\phi(u)$ and $\phi(v)$ in E_H. The *dilation* of an embedding is the longest path in H that is an image of an edge in G. Clearly, the dilation of the embedding relates to the number of cycles of execution in H for the simulation of one cycle of execution of G. The *expansion* of an embedding is the quantity $|V_G|/|V_H|$ which is a measure of processor utilization. A related measure is the *load* of the embedding which is the maximum number of nodes of G mapped to a node of H. The *congestion* (or *load-factor*) of an edge $e \in E_H$ is $\lambda(e)$ where $\lambda(e)$ equals the number of edges in G that have e as a part of the paths that they are mapped to by ρ; the congestion of the embedding λ is the largest congestion among the elements of E_G. Since it is quite possible that every edge is utilized in G in each cycle, the number of cycles needed for every such cycle is λ. If d and λ are the dilation and load factors of an embedding, respectively, and T cycles suffice to simulate any cycle of G, then it is clear from the above discussion that $\max(d, \lambda) \leq T \leq d\lambda$.

Another notion in the simulation of networks is that of work-preservation [KLM+89]. If T steps of an M-node virtual network G are simulated in $O(TM/N)$ steps on an N-node network H, it is a work-preserving simulation since the processor-time product remains the same asymptotically. More generally, the slow-down factor M/N can be different: $q(N)$ $(= M/N)$ could be constant, polylog or polynomial in N leading to the notions of real-time, NC or polynomial work-preserving simulations.

A considerable amount of research on the simulations of networks has been done over the past several years. The research makes use of the above figures of merit to evaluate the simulations. The practical use of these is difficult to gauge at the present time. As remarked earlier, this work could prove fruitful if applied to transformations of distributed algorithms as well.

9.2 Some Aspects of Scheduling on DAGs

We examine two papers ([PU87, PY88]) that are concerned with the efficient scheduling of processors that are arranged in some interconnection architecture for algorithmic DAGs such that an analysis of the performance is made easier. Both papers strive to abstract away from the real architectures by isolating some key features and can thus be regarded as steps toward obtaining architecture-independent analyses of parallel algorithms. Both are concerned with fixed, regular families of DAGs which are already available to the scheduling algorithms. Hence, they do not provide a framework in which to design the algorithms, but nonetheless, an attempt to analyze extant algorithms is made.

The aspects of performance considered in [PU87] are the time and communication requirements of an algorithm. The time is measured as the processing time of a schedule with the communication delays ignored. A *schedule* consists of a set of pairs (p, u) where p is from the set of processors and u is a node from the algorithm DAG. Also, for each such pair, a time is associated that indicates when the node is evaluated by the processor. These evaluation times have restrictions that may be expected due to the DAG and the constraint that a processor is able to evaluate only one node at a time. Given a schedule, a dependency relation D is defined on the processor-node pairs as follows. Let (v, u) be an arc in the DAG (i.e., v is a child of u and its value must be available when u is evaluated). If p computes u, consider all pairs (p', u) such that p' computes v *before* p computes u; the recomputation of some values is possible. Then, by choosing one pair (p', v), we relate (p, u) to it in the dependency relation D. One such tuple exists in D for every (p, u) and v. The dependency relation characterizes a particular choice of the communication pattern or the communication dependencies using the given schedule. For a tuple $((p', v), (p, u))$ in D, if $p' \neq p$, it is clearly the case that a communication from p' to p with the value of v is required if D is the characterization. Such tuples in D denote *communication arcs* of the schedule, and the communication choices made, and the total number of such arcs is the communication required to execute the DAG. Note that there is an implicit assumption that a fully interconnected set of processors is available, although, for a particular DAG, all the physical links may be unnecessary. Another useful measure examined is the length of the chains of communication arcs; the largest among these chains measures the total communication delay. For the case of the diamond DAG that is considered in [PU87], a *communication node* is a node in the DAG if it is evaluated by a processor before the far end of an incoming arc of the DAG is not evaluated by the same processor earlier. Thus, such nodes imply at least one communication arc. Counting the number of communication

level with the maximum number of nodes (a node level is the length of the longest path to it from the lowest level) provides the number of processors needed to achieve the exact lower bound on time given by the depth of the DAG. By the careful scheduling of processors, such as Brent's scheduling principle [KR88], it is often possible to achieve the asymptotic time-bound of the depth while reducing the processor-time product. Thus, given the algorithm as a DAG, it is of interest to see how the processors should be scheduled to obtain better efficiencies.

Another benefit of using the DAG representation of algorithms is that it provides hints as to what the performance is likely to be in the case of particular architectures with a given interconnection structure for the processor and memory elements. Consider an arc (x, y) in the DAG and let the nodes denote the values to be evaluated. The arc implies that x must be evaluated before y and that the value of x is required for the computation of y. If x is not computed at the same processor as the one that computes y, a communication of the value of x is implied. We have seen that the communication aspect of parallel computing is quite crucial, and this approach makes this overhead more explicit. For example, clusters of nodes with several arcs amongst them are better evaluated at the same processor. Further, it may be worthwhile recomputing some values (i.e., evaluating the same value at several processors) instead of communicating them to other processors from a single location. The dependencies, together with such criteria, offer some hints about how to schedule processors for evaluating the nodes. Also, given the communication costs, it may be possible to ascertain the best scheduling for some regular graphs for regular interconnection architectures (as we describe below). In this regard, the example of maximum-finding of Section 3 may be reconsidered. In general, the match between the algorithm DAG and interconnection architecture provides a good measure of both performance and ease of implementation.

Distributed algorithms are often judged by the number of messages that they require in a computation. However, it may be more useful to consider the symmetry of the communication pattern as well. Thus, algorithms that do not use a centralized process to act as the communication hub are preferable — and this is not due to fault-tolerance criteria alone. Since these algorithms are useful for the parallel computing systems in general, it may be well worth studying them from this angle as well. However, aside from some simple architectures such as rings of processors (which are not very interesting as far as tightly-coupled systems are concerned), not much has been done in this direction.

This section examines some research on concerns just described.

9.1 Simulations among Networks of Processors

By careful consideration of the interconnection network of processors and the communication costs, it is possible to design very efficient algorithms that run on that specific architecture. This is achieved by sharing the computation load evenly among the processors, ensuring that the processes that communicate frequently are executed on proximally located processors, and the processors and communications are scheduled to ensure that necessary inputs to a processor are available at the requisite spatial and temporal locations ([KLM+89]). The algorithm DAG itself is embedded within the interconnection structure of the architecture. Thus, graph-theoretic studies on embeddings of classes of graphs on each other often proves to be valuable. Results of the following form are available: Given a class of graphs Ψ on n nodes, how to construct a universal graph H on n nodes with the fewest edges necessary so that every graph in G$\in \Psi$ is a subgraph of H. Thus, every graph in Ψ can be simulated in H without additional communication overheads.

In this subsection, we are mainly concerned with [KLM+89]. If the same algorithm now requires to be evaluated on a different architecture, it is useful to be able to port it so that it runs with reasonable efficiency elsewhere. One way to achieve this is to find generic methods to simulate algorithms developed for a given network on another one efficiently. Although this may not produce the best results possible, it does provide portability with a fair degree of efficiency. The model used for this research uses a parallel synchronous network of processors in which each processor can communicate with each of its neighbors in one clock cycle. For the purposes of this report, embedding an algorithm graph G in a graph H implies a simulation of an algorithm DAG G on an architecture with the interconnection graph H. The simulation is restricted to one where all the steps in H pertaining to a single step in G are completed before the next step of G is started upon.

of implicit synchronization after every step. The model actually is concerned with processes rather than processors which provides the freedom to schedule available number of processors under certain conditions. The interleaving semantics of distributed systems are obeyed with certain timing constraints. The timing is defined by the logical clock times as in [Lam78]. The three types of events that are permitted within a process are local events that concern the local memory, and read and write events on the global memory. To be close to the models used in the PRAM domains, the complexity measures are defined by the number of processes and the number of rounds (see Section 8). Note that it is easy to find analogues of Brent's scheduling principle in the model using processes and rounds in place of processors and time. As may be expected, correctness issues become a major concern since the model is very similar to the models used in distributed computing.

The interprocess communication is done through the shared memory: process p_i may send a message to process p_j by placing the message in some pre-determined memory location that is read by p_j. Since the model is asynchronous, p_j normally has no means of finding-out whether a message has been placed without actually checking to see if it has. That is, busy or idle waiting has to be done on the value of a memory cell to effect communication. In a real system, as discussed earlier, this can lead to very high message traffic, and increased congestion may result. Alternatively, schemes where the sender places the message in the recipients' local memory, or interrupt signals are generated by the memory locations in question can be envisaged.

In the case of PRAM algorithms with oblivious communication characteristics, it is possible to design APRAM algorithms quite easily by constructs similar to send and receive primitives in distributed computing. The analysis also becomes simpler, though not the correctness proofs. For example, it is easy to see that an APRAM algorithm for the summation of n numbers exists that uses n processors and completes in $O(\log n)$ rounds. In the usual manner of introducing processor efficiency in PRAMs, this algorithm can be executed with $O(\frac{n}{\log n})$ processes with the same round-complexity.

In the case of algorithms where the communication characteristics are non-oblivious, the situation is more difficult. No longer is it possible to design APRAM algorithms with simple send-receive constructs — the recipient may keep waiting for messages that are never sent. The problem is that the PRAM algorithms assume synchronous behavior. Although this can be implemented in the APRAM by introducing a 'barrier' synchronization statement after each step which essentially synchronizes processes, the cost would be prohibitive. Hence, the approach that has to be resorted to is redesigning algorithms quite substantially to reduce the number of times that the barrier is invoked. In the example of graph connectivity provided in the paper, the algorithm has had to be redesigned quite extensively. Furthermore, the round analysis and correctness proof are not as straightforward as it is for the case of the PRAM.

It is not obvious that the APRAM model differs very significantly from the models used for distributed algorithms. Instead of using synchronization pairs in the form of send-receive pairs, the other model uses messages. Furthermore, the complexity measures are very similar. For the APRAM model, further research in the direction of results on complexity in the case that a subset of processes slow down considerably, is proposed, and this may prove to be quite useful.

9 DAGs and Processor Scheduling

Given the abstract model of a machine, a designer of algorithms is provided a repertoire of primitive operators that the machine can handle. After the design phase, the resulting algorithm can be represented as a DAG (constructs such as loops may be 'opened-up'). The nodes of the DAG correspond to unit-time primitive operations from the available instruction-set, and the arcs constitute the dependency relations.[19] In research concerning programming notations, compilers and systems-related environments, algorithms are widely viewed as DAGs ([AG89]).

The DAG for an algorithm reveals several interesting features. If the models of computation are the PRAM and the RAM, the depth of the DAG is a lower bound on the parallel time-complexity of the algorithm, and the number of nodes is clearly the sequential time-complexity. The number of nodes in the

[19] Equivalently, the nodes may be regarded as values that need to be evaluated by the primitive operators. The similarity to dataflow programs should be clear.

Note that although the Asynch-PRAM does not permit a more MIMD style of developing programs which have a greater potential for improved performance, the structured approach does offer an idea about the effect of synchronization costs and make the analysis easier. Variants of the model permit a subset of processors to synchronize, vis-a-vis all-processor synchronization, which more closely approximates the less structured models.

The paper refers to all-processor synchronization models and appropriately names them *Phase-PRAM* models. It is possible to allow modeling the memory access time delay d by including a $2d$ time cost for the reads and d for the writes. For several cases where there is no possibility of conflicts that may lead to inconsistencies in the stored values in the memory, it is possible to pipeline the read and write phases (e.g., consider the case for oblivious algorithms mentioned above). An all-processor synchronizing Asynch-PRAM with memory-access times included is termed a *Phase-LPRAM*.

Several straightforward lemmas are proved first:

- a PRAM algorithm running in time t on p processors can be simulated on a similar (EREW, CREW, or CRCW) Phase PRAM or LPRAM in $O(Bt)$ time with p/B processors. This is achieved by having each Phase processor simulate B steps of the algorithm before synchronizing (in the Phase LPRAM, the reads and writes are pipelined) — thus balancing the overheads.
- a Phase PRAM or LPRAM program with p_0 processors that takes $t+B(p_0)s$ time where s is the number of synchronizations, can be simulated by another one with $p < p_0$ processors in $O((p_0/p)t + B(p)s)$ time. A more general version of Brent's scheduling is discussed below.
- with $d \leq B$, a locally oblivious Phase PRAM can be simulated by a Phase LPRAM with the same time and processor complexities. This is achieved, as mentioned above, by pipelining the reads, computes and writes in each interval. Since many of the algorithms refer to locally oblivious Phase PRAMs, the paper restricts further attention to Phase PRAMs alone.

Once the model has been defined, designing and analyzing algorithms for the model is a more routine task. First the more common algorithms are designed; if the complexities compare well with the best known results, or with 'obviously' good methods, the model gains more credence. Consider the summation of n input values. By using a B-ary tree approach where each active processors sums-up B values each time, it is not difficult to envisage the algorithm running on n/τ (where $\tau = B \log n / \log B$) processors of a Phase PRAM or LPRAM in $O(\tau)$ time (initially, each processor adds-up τ values prior to synchronizing). This algorithm is optimal since the following result is proved (via induction, for example), and the result is applicable to any single-output associative function that depends on all inputs and whose basic steps involve only binary operators. Given n numbers in the global memory, and the following repertoire: $L \leftarrow G$, $G \leftarrow L$, $L \leftarrow L + L$, and *synchronization*, (L is local memory, G is global memory, and $+$ is a binary associative operator corresponding to the function), the $+$ operation on all the n numbers requires $\Omega(B \log n / \log B)$ time on a CRCW Phase PRAM irrespective of the number of available processors. Using known PRAM results, one can see the effect of the overheads in the form of the $B/\log B$ factor in the complexity. This factor plays a significant role in the other basic algorithms on list ranking, FFT, bitonic merge, multiprefix, integer sorting etc. that are developed for the Phase PRAM. In many cases, the original PRAM algorithms have provided some guidance; and the proximity of the two models is evident.

The paper also provides an example of the development of an algorithm for on-line load balancing for the Phase PRAM. The question of load balancing is quite important and has applications which pervade several aspects of parallel computing [AG89, KLM+89, PY88]. In this context, the paper also provides a general version of Brent's scheduling principle: a Phase PRAM program using p_0 processors, s synchronizations, x work, and $t + B(p_0)s$ time, can be simulated by another with p processors in $O(x/p + t + B(p)s)$ time. Also, several simulation results with formal circuits and languages are provided but are not germane to the purposes of this report.

8.2 The APRAM

Another model that is very close to the models used in distributed algorithms is the *APRAM* model. The model has a unit-cost access available for memory references as in the PRAM, but there is no concept

the PRAM repertoire of primitives. Situations can be envisaged where the repertoire of instructions available to the model is governed by the efficient algorithms that the underlying architecture can provide. The algorithms developed using these instruction sets, when converted to real time steps, would model the real costs quite closely.

Several other models addressing issues of fault tolerance, redundancy, and other important features, have been left out of this report due to time and space constraints.

8 Asynchronous Models

As remarked earlier, parallel computing systems that have a substantially large number of processors are necessarily asynchronous systems. This asynchronism is of different types: message-order asynchrony, asynchrony in delay times, clock-speed mismatches between processors etc.. Foreseeable technology also does not permit the realization of maintaining synchrony in large systems (e.g., consider maintaining global clocks). Such systems could be assumed to be totally asynchronous. That is, the time intervals as measured by one component of the system may bear no relation to that measured by another — similar to the assumption made in several distributed algorithms. In real systems, though, the situation may not be so pessimistic; there often is some relation so that one may envisage some degrees of synchronism. As far as tightly-coupled systems are concerned, clock synchronization is an important factor that affects performance. It is usually a fairly expensive operation to actually effect a global clock since this means that, in effect, a synchronization is done at each point that the clock-tick occurs (in fact, that is precisely what a global clock achieves). In a large-scale system of processors such as the Connection Machine, the global clock is only a simulated one: clock-tokens are sent through the system and the receipt of a token by a processor indicates that the clock has 'ticked'. That is, the effect of system-wide timing is maintained via synchronization. In this section we shall examine two models ([Gib89, CZ89]) that address the question of asynchrony in tightly-coupled systems.

8.1 The Asynch-PRAM

Two major difficulties that [Gib89] identifies in mapping PRAM algorithms to real machines is the efficient implementation of the read and write primitives on the interconnection architecture (see Section 5), and enforcing the necessary synchronization implicit in the algorithms onto the machine. It is the second aspect that concerns the paper. To represent the requirement, the paper parametrizes the cost of global synchronization as a value, B, which then appears as a parameter that affects the algorithmic complexities. The machine model used is basically the PRAM, with the difference that synchronization costs are made explicit. Also, the methods to develop the algorithms help make the correctness and complexity analyses simpler. The results in the paper provide a good means to exhibit the effect of synchronization requirements on the complexity measures of the algorithms.

In the *Asynch-PRAM*, the computation is conducted in a series of *intervals*, and within each the processors are permitted to run asynchronously. The intervals are separated by synchronization points which have a time-cost of $B(p)$ which depends on both, the number of processors p, and the particular architecture family considered. The only (reasonable) restrictions on B are that $B(p+1) \geq B(p)$ and $B(p) \geq p$; note here that if 'busy waiting' occurs for synchronization, this may not be a reasonable abstraction. The time cost of an interval is the maximum number of instructions executed by a processor. To permit the asynchronicity within an interval, a processor i does not read what a processor j writes in the same interval (every PRAM step has three phases of read, compute and write). Thus, the communication between processors only straddles synchronization points (i.e., is between intervals), rather than at every step. Hence, CRCW, CREW and EREW PRAMs may be defined for operations pertaining to an interval.

The results in [PY88] can be obtained by a minor variant of the Asynch-PRAM. Setting B to the greater of $2d$ and the synchronization cost, where d is the delay incurred between the processors and the memory, oblivious algorithms can be restricted to let all the reads be issued first, followed by the computes, and finally the writes. Synchronization steps are executed between these phases. Such *locally oblivious* Asynch-PRAMs yield results from [PY88].

The notion of *conservative* algorithms is introduced: they are the algorithms that produce a congestion that is upper bound by the congestion implied by the embedded data structure on which they execute. Thus, the capacities of the cuts need not be considered for such algorithms since the capacities are taken into account within the data structure, and the conservative nature of an algorithm is determined by the loads implied by the algorithm and the data structure embedding. Therefore, T steps of a conservative algorithm executing on a data structure with a load factor λ executes in at most $T\lambda$ steps. Network-flow algorithms, that can be run on a Connection Machine [Hil85], are an example of this since the set of memory accesses of the algorithms are a subset of the pointers in the input data structures. Several algorithms are developed to exhibit the effectiveness of the DRAM model and the notion of conservative algorithms.

The DRAM is close to the machine architectures. While resulting in pragmatically efficient algorithm development, the complexity of the development and analysis is fairly substantial. Firstly, the data structure embedding must be considered using the underlying network. For reasonably structured networks, the capacity assignments are easy to develop. Hence, an algorithm is developed that has conservative characteristics. Thus, although the development proceeds quite slowly, important quantitative guidelines are made available to design congestion-free algorithms.

7.5 Other Models

Models abound, and many are interchangeable by the correct assignments made to their parameters. In this section, several other models are examined briefly.

The *Direct Connection Machine*, or the DCM, consists of RAMs with unit cost local memories, that communicate using only message passing primitives that are also unit cost. Aside from the usual operations, send-receive primitives are available with a unit buffer size. This is clearly quite close to some of the other models discussed above. It is also quite close to the models used in distributed computing research. Memory cells in real computers are usually grouped together into modules with limited-access ports. To model this architectural aspect, research with models that have PRAM characteristics except for memory grouped into modules — usually the same number as the processors — has been fruitful. Besides the DCM mentioned above, the model appears in guises such as the *Module Parallel Computer*, and the *Seclusive PRAM*. All such models usually assume complete interconnection graphs.

In [Sny86], a general guideline to develop models is suggested. A *type architecture* is suggested that bridges the extremes between programming notations and real machines, and its fundamental utility is the accurate modeling of algorithmic costs. The paper proposes a *candidate* type architecture (CTA) that is defined as a finite set of sequential computers, with a global controller, that has its elements connected by a fixed, bounded-degree graph. The sequential nature of the computers does not preclude pipelining etc. within the general framework of sequential execution; the connectivity implies efficient communication between adjacent nodes in the graph without actually requiring a particular physical network; and the global controller is an artifice to ensure the logical existence of weak controls such as a reset signal that may be implemented in any physical manner desired. The CTA is a fairly general model. However, the paper does not indicate the manner in which complexity measures are to be made. Perhaps this is because the measures depend on the particular type of system that the CTA is expected to model.

Sometimes, the architecture prompts changes in the model that makes the model stronger. For example, the development of fetch-and-add primitives has prompted the development of PRAM models that are augmented with fetch-and-op primitives. Clearly, these models are at least as powerful as the usual PRAMs, and they are certainly more powerful in the sense of having the augmented capabilities. The fetch-and-op primitives behave like atomic read-modify-write actions.

Section 4 describes how reasonable architectures have $\Omega(\log p)$ shared memory access times. For $n = p$, PRAM algorithms directly implemented on these reasonable architectures require $O(\log n)$ time per PRAM step. Several reasonable architectures are able accomplish certain other operations that are more complicated as compared to memory access alone within the same time bounds. For example, the parallel prefix computation takes $O(\log n)$ time on reasonable architectures. It then makes sense to regard these operations also as unit steps in the PRAM computations. Thus, recently, *scan* models have been suggested where steps such as parallel prefix, routing etc. are charged unit PRAM costs. That is, these *scan* operations augment

Some of the tradeoffs developed in [PU87] are also obtained for the LPRAM — although the LPRAM does not allow the pipelining of communication and computation steps. Global communication bandwidth is the only aspect of communication measured in the LPRAM; the effects of communication latency are not (see the BPRAM). Drawing from other practical studies (e.g., [SS86, Ath87]), it is seen that in message passing systems, the latency period for short messages is dominated mainly by a fixed start-up time.[17] Thus, the difference in the latencies to send messages to neighbors and distant processors is negligible. Hence, the LPRAM may be regarded as a reasonable model.

7.3 The BPRAM

The *Block PRAM* model (BPRAM) is developed in [ACS89] to study the effects of communication latency when these are incorporated into the PRAM model. The model has global and local memories (see the LPRAM model), and also has a block transfer capability between the two (see the BT model) that requires $l + b$ time for a block of b locations where l is the latency associated with global memory access. The model is EREW with respect to the blocks, the I/O is done in global memory, and the complexity measures are the same as the PRAM model.

Several obvious bounds are easily derived for the BPRAM (e.g., $T_{BPRAM} = \Omega(l)$, $w_{BPRAM} = \Omega(lp + T_{RAM})$, and others for specific problems). Problems such as matrix operations, rational transpositions, permutation networks, sorting etc. are studied in the context of the BPRAM. In these, the appropriate use of spatial locality properties lead to interesting bounds. Many of these relate to parallel efficient algorithms (see Section 3). The utility of the abstraction is apparent.

7.4 The DRAM

The *Distributed RAM* model (DRAM) is an abstraction developed in [LM88] to design algorithms that take into account the important effects of network congestion (see Sections 7 and 8) in the underlying architectures. The model is quite different as compared to the parallel models discussed thus far in that it does not assume an underlying shared-memory architecture.

The DRAM consists of several RAM processors that have local memories, and the processors are able to access remote processor memories though that incurs a communication cost. A set of memory accesses with one for each processor, both local and remote, is performed in one step. Remote accesses are effected by routing messages through a network. Communication contraints of the network are modeled by assigning a capacity $cap(S)$ to each subset S of processors that is intended to model the number of wires between S and S'.[18] For a set M of memory accesses, $load(M, S)$ is the number of accesses in M from a processor in S to a processor in S', or vice versa — this indicates the amount of traffic that M imposes on the $S - S'$ cut. The *load factor* of M on S is $\lambda(M, S) = load(M, S)/cap(S)$, and the load factor of M on the DRAM is $\lambda(M) = \max_S \lambda(M, S)$.

The time taken to perform a set M of memory accesses is taken to be $\lambda(M)$. Though $\lambda(M)$ is a lower bound on the time taken in the real network, the proximity of the upper bound of message routing on the network determines how well the DRAM models the real network. When the message routing closely approaches the lower bound (e.g., within a polylog factor), the DRAM is deemed a good model.

A data structure is embedded in a DRAM by placing the records in the RAM memories, and the structuring pointers between the records. The pointers traversing processors are embedded in the underlying network, and similar notions of load factors etc. that are discussed above may be developed. This is a justifiable notion since communication is needed to access records using the pointers. The congestion produced in the network is related to the cut capacities. Firstly, if too many pointers are embedded in one network link, parallel access of the records involving those pointers results in high message traffic. Furthermore, the algorithm itself, independent of the pointers, may generate a high degree of traffic, and this is the focus of the research in [LM88].

[17] Message traversal time is usually represented as $l + mb$ where m is the message length, l and b are system parameters with $l \gg b$.

[18] Usually, the networks are likely to be well-structured and, hence, the assigned capacities are likely to have closed-forms for facility in the analysis.

controllers). The effect of these factors is significant — for example, consider attempts to multiply matrices using dot product algorithms when the matrices are stored in row major order in one case, and column major order in the other.

The model studied in [ACS87] assumes that the basic RAM model has its memory access time characteristics changed in that accessing memory location x requires time $f(x)$. Also, a block transfer capability is provided: a contiguous block of l memory locations beginning at location x may be transferred to another location beginning at location y in a time interval of $\max(f(x), f(y)) + l$ units. The function f is assumed to be non-decreasing, and the blocks range $x, \ldots, x - l + 1$ and $y, \ldots, y - l + 1$. The model is denoted by $BT_{f(x)}$ to signify a block transfer RAM with access-time function $f(x)$; the RAM model is equivalent to BT_1. Semiconductor memory circuits and secondary storage access times motivate the study of the $BT_{\log x}$ and BT_{x^α} (where α is a constant), respectively. Several problems are studied in the context of the models — some serving to illustrate the basic features of the models. For example, the *touch* problem requires that given n inputs, an algorithm should touch each input where touching denotes bringing the element to the lowest location in the memory. It is proved that any algorithm that touches cn inputs requires $\Omega(n \log\log n)$ time on BT_{x^α} for $0 < c \leq 1$ and $0 < \alpha < 1$. The same time complexity is also shown to be an upper bound for the problems of touch, dot product of two vectors, deterministic CFL recognition, merging etc.. This clearly exhibits the effect of unequal memory access times. Bounds for more complicated algorithms are shown, and also models with other functions f are examined.

To a certain extent, the model does exhibit the effects of temporal and spatial locality: the algorithm designs need to exploit these to achieve the bounds in many cases. For a closer rendition of real world situations, models with f provided as a step function would be more suited since a particular portion of the memory hierarchy in real systems typically has the same access time (e.g., access times for locations within a disk are quite close to each other compared to the difference between accessing the main memory and disk memory locations). However, if the model is very close to the real machine, at a particular point one may as well work on the real machine — the whole idea of an abstraction becomes futile.

The above nicely illustrates models and their use. The closer a model gets to a real machine, the more difficult is the development of algorithms for it — though the results are better. We examine several abstractions that modify some of the models discussed in Section 3 to obtain closer abstractions to real parallel machines.

7.2 The LPRAM

Communication complexity of parallel algorithms developed for the PRAM model is usually a topic on which practitioners have very critical opinions. The *Local-memory PRAM* abstraction (LPRAM) is developed in [AC88] to study, ostensibly, this issue. The model seeks to separate the access time for global shared memory as compared to that for the local memory of a processor in a parallel system. This is an important point of departure that the PRAM exhibits from the real world situation, and the LPRAM seeks to rectify that. The model is reasonable since, in practice, it is often the case that efficiency of a program is considerably enhanced by separating global shared variables the the locally owned variables.

The LPRAM is a form of the CREW PRAM restricted to perform either a communication step or a computation step — all processors do these in concert. A communication step involves writing into, followed by reading from, a location in global memory, from and to a local memory location, respectively; a computation step is a binary operation on two local values. A DAG is used as an abstraction of an algorithm (see Section 7). The measures of complexity are the number of processors, the number of computation steps, and the number of communication steps. For a given DAG, the minimum over all the schedules provides a measure of the complexities — however, it may not be possible to minimize all the measures simultaneously since tradeoffs similar to those in [PU87] exist. Representative results include multiplying two $n \times n$ matrices in $O(n^3/p)$ time, $O(n^2/p^{2/3})$ communication delay using $p \leq \frac{n^3}{\log^{3/2} n}$ processors, and sorting n words with comparisons only in $\Theta(n \log n/p)$ computation time, $\Theta(\frac{n \log n}{p \log(n/p)})$ communication delay using p processors with $1 \leq p \leq n$.

chosen from a class of universal hash functions, is used to distribute the variables in the modules (i.e., map the logical locations to the physical ones); the number of random bits required is a small $O(\log^2 N)$. The combining switches behave as follows. Each ensures that the stream of messages leaving it are sorted by the tag values. To effect this, a message packet is not transmitted unless it is ascertained that every other packet sent by it will have a higher tag. If necessary, the switch awaits further information from preceding switches. To avoid unnecessary delays, when a switch chooses to send a message with some tag on one outgoing channel, it also sends out a ghost message on the other with the same tag so that the subsequent switch gets to know the lower bound on the tag value that it will receive from the preceding stage. If two requests arrive for the same location, the two are combined into one request for the subsequent stage. For the return path in case of a read request, two bits per message are stored at a switch (to indicate, for example, which path the return message should be routed to). This is sufficient because the deterministic nature of the routing ensures that the return messages arrive in the same order as they went. Since the total emulation time is $O(\log N) = O(n)$, only $O(n)$ bits need to be stored in the switches which is $O(1)$ in terms of message sizes.[15]

The emulation is shown to succeed in a probabilistically short time — the details of which cannot be provided concisely. The methods used are similar to those used in [KU86] in building sequences of message delays and exhibiting that a long emulation time implies the existence of an unlikely sequence of entities. From the practical point of view, it should be noted that the constants involved and the size of the queues required are reasonable.

7 Incorporating Communication and Retrieval Overheads

Models are abstractions of machines or systems and have the important role of simplifying the usage of the entities they represent.[16] The key features of the underlying entity are recognized and abstracted into the model, and the less relevant details are ignored. Thus, the model plays a descriptive role. Conversely, it may happen that models dictate the construction of the entities that they are supposed to represent (see Snir's paper in [OCP88]) — efforts to build shared-memory systems that provide the capabilities of shared-memory models may be viewed in this light.

The level to which abstraction is done depends, on one hand, on how difficult the abstracted details are to work with. On the other hand, the application for which the model is created dictates how oblivious of the details of the real entity the model can possibly be. It is often the case that the development of a suitable model is uncertain while the development of algorithms for the model, once it is available, is more certain. Whether or not the developed algorithms are of use depends largely on how well the model abstracts the key features of the underlying system. Notice that although the RAM abstraction is usually deemed acceptable, it does not capture several important details of real systems that render unusable a large number of algorithms that have been developed for it. We examine this more closely.

7.1 Hierarchical Memory in the RAM

An assumption is made in the RAM abstraction that all memory accesses take only unit-time. In reality, the existence of a memory hierarchy of local registers, caches, main memory, disks, tapes etc. make the access times different for different memory locations. Temporal and spatial locality become important concerns in algorithms designed for such cases. Also, block transfers of memory elements, where a consecutive block of memory elements may be copied from one part of the memory to another, is of consequence. This is an abstraction of paging etc. that pertains to real systems (e.g., those that use virtual memory and DMA

[15] It would appear that the queue size is not entirely independent of the size of the computer. Strictly speaking, that is true. However, since logarithmic costs [AHU74] are not being used, we always do consider $O(\log N)$ bits as $O(1)$. In several distributed computing algorithms, this is not permitted, and research has been done on systems that have components that are unable to store even their own identities.

[16] The term model is used quite generally here. It is not restricted to a machine necessarily — in fact, a model may be an abstraction of a piece of software, or indeed, another model. For example, in a layered approach to system development, the upper layers are provided the key features of the ideal machines represented by the lower levels so that the task of development is simplified.

it is shown that most of the graphs satisfy the requirement. Thus, if the assignment as represented by such a graph is chosen randomly, the chances are that the algorithm works as stated. A good assignment scheme is assumed to be available to achieve the bounds stated in the paper, but the explicit construction of such graphs efficiently, is still an open question.

The approach in [AHMP87] is essentially the same, and the models used are the same. The paper is an improvement of the results stated for [UW84], although the question of an explicit construction, and therefore a practical deterministic simulation, is not resolved. The paper exhibits a non-uniform emulation technique by which any step of the PRAM can be emulated in $O(\log n)$ steps on a Module Parallel Computer as long as m is polynomial in n. Also, it is shown that a bounded-degree network of processors requires $\Omega(\log^2 n/\log\log n)$ steps to emulate an arbitrary step of a PRAM when point-to-point communication is assumed. Note that any step of a fully interconnected network, such as the Module Parallel Computer assumes, can be simulated by a bounded-degree network in $O(\log n)$ steps.

6.2 Probabilistic Methods

The paper [KU86] provides a probabilistic algorithm to simulate arbitrary T steps of a (n, m) PRAM, with m that is polynomial in n, on a bounded degree network of n processors, with local memory only, in $\Theta(T \log n)$. The $\Omega(T \log n)$ is a standard lower bound. The probabilistic approach provides an algorithm that always terminates, and is always correct. The running-time varies depending on the random choices made during the execution of the algorithm. The model used to describe the method is a distributed memory system with a local memory at each processor, and with bounded-degree synchronous communication links.

The method employed in the paper uses *parallel hashing* that uses a function from a set of universal hash functions to distribute the variables, and also a probabilistic routing algorithm. The size of the set of hash functions chosen is $m^{O(\log m)}$ implying that $O(\log^2 m)$ bits are needed to describe the function chosen; if a totally random distribution were used, $O(m \log m)$ bits would be needed. The routing involves sending messages to a random location first followed by the shortest routing to the actual destination (see Section 4). The number of random bits required, a scarce resource, is $O(n \log n)$. A priority is associated with each message that diminishes with the actual transitions made. Contending messages are sent according to priority. The queue sizes could become $O(\log n)$, and they need to be maintained as priority queues to handle cases of contention. Thus, it becomes clear that the overhead hardware required to support the message traversal system is quite complicated. In these respects, the method developed in [Ran87], which we examine next, is better.

The [Ran87] paper provides an efficient probabilistic simulation of a CRCW PRAM on a butterfly network of processors. The approach used is easy to understand and is also quite practical. The major results are that any N-processor PRAM step can be emulated in $O(\log N)$ time steps of the bounded-degree butterfly network with N processors with high probability.[14] Further, the queue size at each node for the routing of the messages is $O(1)$ in size. Also, the queues are very simple FIFO channels. The routing itself is deterministic and oblivious — thus, the randomization is needed only once, offline, to distribute the variables used by the PRAM algorithm. The oblivious nature of the routing indicates that the input algorithm does not affect the nature of the routing. To scale-up parallel computers in size, it is essential that the complexities of the individual components should not also require scaling-up, and it is seen that the technique accomplishes this exactly. Other schemes with similar time bounds do not succeed in obtaining the limited queue size that this paper describes (contrast with [KU86]).

Let us examine the emulation for a write request. The emulation may be considered to run on three logical stages of size $N = n2^n$ each, where the neighboring stages share a column of the butterfly. The functions of each stage are, of course, done by the single physical stage of the butterfly. The first stage may be regarded as the processors, the second as the routing network, and final stage as the memory modules. Messages are routed from the processors to the first column of the router (at the same row), then to the destination row and final column of the router, and finally to the destination (row and column).

The messages carry three fields: the type (request, ghost, or end-of-stream); the tag (which indicates the destination location — logical and physical); and any data that the message carries. A hash function

[14] The upper-case letter N has been chosen since we will let $N = n2^n$ for the subsequent discussion.

6 Simulating Shared Memory

From the preceding sections, it is clear that a parallel computer of reasonably large size with a physically shared-memory is infeasible. Hence, if it is required that the algorithm and program design be done on a shared-memory model of computation, it becomes necessary to map the results on to more realistic machines or machine models. In this section, some of the results and techniques to simulate PRAM instructions on more reasonable models are examined. Two papers that provide a deterministic approach to the simulation are examined first. This is followed by two probabilistic approaches. Although the simulation techniques are not on the same model, other results relate the simulations to one another.

6.1 Deterministic Methods

A complete network of processors, each with a local memory, is considered in [UW84]. In this model, no shared-memory of any sort is permitted. The system is synchronous, and the local memory modules are assumed to be able to handle only one service request at a time. When several processors make requests to the same memory module, only one arbitrarily chosen request is granted. The processors may access in unit time any memory module that is free of contention. The organization is similar to the MPC of Section 6. The results exhibit a method to simulate an arbitrary EREW PRAM algorithm in an on-line simulation. The basic result is that a single PRAM step (i.e., a step for each of the processors in the PRAM) can be simulated in the model just described in $O(\log n(\log\log n)^2)$ time where there are n processors, and the size, m, of the shared-memory accessed in the PRAM is polynomial in n. This bettered the previous best trivial bound of $O(n)$. However, the method is *non-uniform* — which means essentially that the existence of such a simulation is shown, but an explicit method by which to carry-out the simulation is not available currently. It is also shown that T steps of a PRAM require $\Omega(T\frac{\log n}{\log\log n})$ steps on the fully connected MPC for on-line simulations.

If $m > n^2$, without the use of hashing or duplicated copies of variables, one local memory has at least n data items. This implies that a pathological program may slow down the simulation to $\Omega(n)$ steps of the MPC for each step of the PRAM by accessing variables in that memory module alone. If m is about the same size as n, routing by the use of sorting described in Section 4 achieves the desired results. In other cases, probabilistic methods are available.

The basic method used in [UW84, AHMP87] involves the use of several the copies of each variable, and copies are read and updated using a majority consensus protocol in conjunction with timestamps. The method uses aspects of superconcentrator graphs, and thus introduces the non-uniformity. However, the randomization needed to realize the graph is used just once for any pair of n and m values.

Let $\Psi(u)$ be the set of memory modules with a copy of a shared variable u, and U be the set of shared variables accessed ($|U| = m$). An *assignment* scheme S assigns sets $\Psi(u)$ to each $u \in U$, and also provides the read and write protocols that essentially use the majority consensus. Since a protocol need not update all copies, at least one copy must have the latest value to ensure a consistent simulation. Specifically, in the case of [UW84], $2c-1$ copies are present for each variable, and c copies are accessed to read or to write values. It should be clear that the use of timestamps unequivocally provides a means to ascertain the correct values.

The algorithm to simulate one step proceeds in $2c$ phases. At most $k = n/(2c-1)$ variables are accessed at a time by forming k groups of $2c-1$ processors each. In each phase, from each group, one processor has its request attended to. All the processors of a group attempt to access the appropriate variable assigned to the group, and at the end of every attempt, a check is made to ascertain whether the access has indeed been accomplished. The number of attempts in every phase is bounded, and it is proven that at most $k/(2c-1)$ items remain that could not be accessed (out the ones assigned for the phase). Hence, the final phase which is allowed to run to conclusion, needs to access at most k items. The desired bounds are obtained with $c = \Theta(\log n)$.

For the above algorithm to work within the stated time bounds, it is imperative that the assignments $\Psi(u)$ for the elements of U be good. That is, the assignments should ensure that the copies are well spread-out among the modules so that for any given access pattern, the model does not create excessive contention at a single module. By examining sets of $(|U|, n)$ bipartite graphs whose edges denote the $\Psi(u)$ assignments,

in real computing systems. These are also the same problems that are addressed for database systems. If the emulation techniques for implementing PRAMs on more realistic models is examined, it will be observed that the essential problem is one of managing the data in the memory, and techniques very similar to those used in distributed databases are used. In [SS88], the similarity is exhibited further. The only difference between the areas of database concurrency control and parallel computation management is that in the former, it is not known *a priori* to the controller which instructions (or transactions) will be executed, while in the latter, the analysis of programs done by the compiler may be used. The compiler output allows any controller in the system to arbitrate requests etc. so as to achieve a consistent execution.

In terms of *fault-tolerance*, robustness issues in the context of PRAM algorithms have been examined in [KS89], and the motivation and techniques are developed from the communication aspects of distributed computing. The paper addresses only fail-stop behaviors where the processors fail but are not able to recover. Also, malicious failures are not considered. If the notion of failures is extended to include the effect of processors failing and then re-entering the system, notice that the parallel algorithms that would result would have use beyond that of failure-prone systems alone. In a realistic system, it may be expected that the operating system would dynamically re-assign processors to the processes. If such dynamic re-assignment occurs in an ordinary PRAM algorithm, the situation would not only mimic processors failing, but also mimic the case where the processors exhibit unpredictable asynchrony during the execution.

There are other aspects of distributed computing that are finding a lot of use in parallel systems, but to limit this report, neither have they been described nor expounded upon.

5.3 Complexity Measures

Space complexity measures in distributed systems are very similar to traditional complexity theory methods. The space occupied in the execution of an algorithm is measured in the usual manner. An added complication is the size of buffer spaces that arise in models that consider message-passing. This also arises when real computers are considered in the context of routing messages in processor networks.

Since messages are costly in distributed systems, measures have been developed to gauge the complexity of algorithms that run on such systems as regards their overheads with messages. As an example, [Awe87] provides the following model. Consider an undirected graph (V, E), where the nodes are processors and the edges are bidirectional communication links. Processors are assumed to know their own processor identities, but are unaware of who their neighbors are. The communication complexity is defined as the total number of elementary messages sent during the algorithm execution, where each elementary message is at most $O(\log V)$ bits long (this is a common restriction; e.g., in complexity theory, it is assumed that the inputs have $O(\log n)$ bits). Time complexity is also measured using the maximum elapsed time on some global clock assuming that message delays and propagation delays are unity. The global clock, it should be pointed out, is used solely for performance evaluation and not for correctness criteria.

The use of message complexity is not necessarily reasonable. It may well happen that all the messages are sent over one link — thereby making the time complexity a lower bound on message complexity. In this context, [PU87, KLM$^+$89] also have a bearing.

Time analysis, which is perhaps one of the more crucial measures, is difficult since it is unclear in many instances what it is that is attempted to be measured. To add to the problems of asynchronous executions, busy-waiting on variables leads to more complications. One approach taken is to assume bounds on individual actions instructions which then provides an upper bound on the entire algorithm by the use of a worst-case analysis. The concepts of rounds is also used often in asynchronous systems. A round is said to have occurred if every process in the system executes at least one instruction. In the case of waiting, a busy-wait constitutes an instruction — and not the individual attempts. In the context of time analysis, note that the large amount of literature in distributed computing that arises from questions of termination (e.g., consensus problems) are actually studies of the extreme cases of time complexities.

Finally, consider the anomalous behavior of a system where the causality is violated by a signal or message that is external to the system. In such a situation, suppose we still want the clock condition to hold, it is easy to see that the causality relation among the events should include the external signals as well. Let the local logical clocks represent inaccurate readings of a real physical clock (i.e., the asynchrony among the clocks is not complete). Then each physical clock $C_i(t)$ is a function of the real time t, and is approximately the real time itself.[12] That is, there exists a small constant κ such that for all i, $|dC_i(t)/dt - 1| < \kappa$. Moreover, the clocks among the different processes are approximately the same — that is, there is a sufficiently small constant ϵ such that for all i, j, $|C_i(t) - C_j(t)| < \epsilon$. Note that anomalous behavior occurs *between* processes. Hence, if there is a causality relation from event a to event b, and a occurred at t_a, then b must have occurred at $t_a + \nu$ where ν is the physically least amount of time for information traversal from a to b.[13] That is, $C_i(t_a + \nu) - C_j(t_a) > 0$ for where a occurs at processor p_j and b at p_i. Therefore, within a processor, $C_i(t+\nu) > -C_i(t) > (1-\kappa)\nu$, and hence, $\frac{\epsilon}{1-\kappa} \leq \nu$ ensures that anomalous behavior is obviated. In practice, $\kappa \leq 10^{-6}$ seconds is achieved. To ensure correct behavior, a low ϵ is obtained by sending messages frequently enough, if only to resynchronize clocks.

5.2 A Merging of Concepts

One of the underlying themes in this report has been that concepts in distributed computing have importance in parallel computation when realistic computers are considered. As an illustration, this section provides exemplar instances of such a merging of concepts. Although the material is to be found in several papers, a good survey may be obtained from [LG89].

PRAM models use several variants of modes in which the processors may access the memory elements. Although the concurrent access modes may be considered, for the most part, a method of abstraction, there has been considerable interest in distributed computing in the area of *register construction* where some of the variants are realized via software mechanisms. Also, the areas of *mutual exclusion*, though more applicable to process control, have techniques that have a close relation to this problem.

The notion of *atomicity* that pervades most distributed systems is quite important in parallel computation as well. So as to reduce overheads, several actual instructions may have to be done together as a task, or transaction etc., and these have to be atomically executed. As mentioned earlier, models with test-and-set, fetch-and-add and so on, are examples where the atomic operation of the instructions have been relegated to hardware and have provided a richer model for theoretical study as well.

It was seen that parallel computation systems need to address the issue of processor scheduling in a real system, and the effect that such scheduling may have on the efficiency of implemented programs. In distributed systems, the actual scheduling is not presumed. The processes have instructions that have to be executed with some *fairness* that will ensure certain *progress* conditions. The idea in fairness, as the name suggests, is an operational mechanism to ensure that no process is neglected. That is, processes should get processors scheduled to them, and this should not necessarily occur in a deterministic manner. Several modes of fairness may be effected by *randomization* in the scheduler construction, and there are results to show that the two notions are essentially equivalent [Fra86].

Randomization also has a part to play in *symmetry* considerations in distributed systems. In cases where the individual processes are not allowed to know their own identities within the system — such examples occur even in computational VLSI in the interest of building scalable systems — randomization is often the only alternative to break symmetry. Breaking of symmetry provides some ordering within a system so that actions that require inherent asymmetry, such as assigning resources, can be effected. In parallel computation theory, randomization is used very often for precisely similar reasons — to introduce a degree of asymmetry. Since it is often the case that deterministic methods are very difficult to analyze and design, recourse to probabilistic methods is taken to provide load balancing, resource allocation and the like.

Parallel systems have concurrent accesses made to variables that may be shared. This leads to problems if there are concurrent accesses, if the messages are not delivered in order etc. — something that is common

[12] $C_i(t)$ is a piecewise continuous and differentiable function for our purposes. It may have discrete jumps when the logical clock C_i is set forward.

[13] ν is a lower bound given by the distance between the physical location of the processes divided by the fastest speed of propagation of information — the speed of light. In reality, this a pessimistic lower bound.

Rediflow, WRM, and dataflow machines. Most of these have fairly large processors which can store their own programs. The processors need to communicate using messages, and they may need to synchronize from time-to-time, again using messages, since they do not have global clocks. The tasks that each processor executes need to be substantially large to compensate for the effect of message-passing latency. The dataflow machines are somewhat different from the rest in that they are optimized to execute tasks that are represented by banks of tokens.

MIMD designs with shared-memory have been quite closely studied over the past few years. The CMU Cmmp, CMU Cm*, HEP Denelcor, NYU Ultracomputer, BBN Butterfly, IBM RP3, UI Cedar, and CHoPP are a good representative set of examples. High switching costs due to sophisticated switches coerced by certain systems-related concerns (see Section 8) can increase the cost of the machines, and also render them unscalable. Among the innovations that have been considered are the hardware implementation of costly software coordination constructs so as to lower the overheads introduced due to programming. Examples include combining networks and atomic fetch-and-add constructs. Programming these computers is far more easy since the multiprocessor organization often takes care of memory access requirements and a more uniform logical structure is provided to the programmer.

Some other approaches include the VLIW architectures which use several instructions packed into one word to reduce the von Neumann bottleneck; reconfigurable architectures such as the TRAC and PASM; multiple SIMD machines that have several portions that run separate SIMD machines in an obvious attempt to obtain the best of MIMD and SIMD architectures.

5 Relevant Aspects of Distributed Computing

In the recent few years, literature that attempts to bridge the gaps between distributed computing research and parallel computing research, has appeared. In general, parallel computing systems assume synchronous, shared-memory systems with negligible message-transmission overheads; distributed systems assume asynchronous, distributed memory systems with negligible computing time overheads. For reals systems, a more reasonable approach is somewhere in-between with tradeoffs in the message and time complexities implicit. This section begins with the notion of logical time in the context of asynchronous distributed systems.

5.1 Time in Distributed Systems

In [Lam78], time is defined in logical terms since there is no concept of a global clock as in, say, the PRAM model. The events occurring in a distributed system are partially ordered using a relation called the *happened-before* relation that is abbreviated to $\rightarrow$. For two events a and b, $a \rightarrow b$:

- if a and b are in the same process and a precedes b in time.
- if a involves sending of a message and b involves the receipt of the message.
- if there is an event c such that $a \rightarrow c$ and $c \rightarrow b$.

Furthermore, this relation is the smallest possible one, and is clearly the causality relation. The asynchrony between local clocks in a distributed system indicates that $a \not\rightarrow b$ and $b \not\rightarrow a$ imply that a and b are *concurrent* events.

Based on the above definition of the happened-before relation, *logical* clocks are defined that can number the events in a system. Each process p_i has a clock C_i to number the events a_{ij} that occur in it. A global clock C is defined as $C(a_{ij}) = C_i(a_{ij})$. A reasonable analogy to physical time is imposed by the *Clock Condition*: For any events a and b, $a \rightarrow b$ implies $C(a) < C(b)$. Note that asynchrony indicates that the converse need not be true. This condition is easily implemented by the local clocks numbering local events in a monotonically increasing manner in time, and ensuring that a receipt event for a message has a lower number as compared to the event that sent it. The latter is accomplished by timestamping every message by the number of the event that sent it, and coercing the recipient's clock to beyond, at least, that timestamp. A total order imposed on the processes may be used to break ties and thus achieve a total order on the events of the system.

large number of routing techniques that are available for these networks exhibiting varied characteristics of time, determinacy etc. Depending on the application at hand, available resources of time, money etc., it is possible to choose the requisite interconnection architecture, and the related algorithms from the literature.

4.2 Routing

Routing of messages between the nodes of a network is an important facet of parallel computing since its efficient implementation implies efficient communications and memory-access mechanisms. Routing also plays the crucial role in the emulation of ideal computers on more realistic ones as we shall see in the next section. In this subsection, some of the basic paradigms in the area are described from [Ull84a].

In a bounded-degree network, at least $O(\log n)$ levels of switches are necessary to be able to connect together every processor to every memory element (when there are n of each). Hence, at least $O(\log n)$ time must elapse between the request for a value and the arrival of the requisite value. Assuming an EREW request pattern, this is easily achieved when the access pattern is known to a global controller *a priori*. If the switches use local controls, the routing information (e.g., the destination locations) is carried within the messages, and decisions on which direction to route are made at the switches. The routing schemes may be susceptible to deadlock situations where none of the messages are forwarded due to a cyclic wait-for graph within the interconnection network (the buffer sizes within the nodes are of bounded size).

Several kinds of routing have been studied. *Permutation* routing has every processor generating a unique memory location request. To effect this, messages carry the destination numbers, and an efficient sort on these numbers provides the necessary routing. Hence, a fast sorting scheme on the interconnection network implies a fast permutation routing scheme. *Partial* routing also requires the processors to produce unique location requests, but all the processors need not make requests. A probabilistic algorithm provided in [VB81] describes how this is achieved. The idea is to send every request to a random location first, followed by routing to the appropriate location. This is necessary to minimize contention of edges since the actual mechanism used itself may be deterministic — a pathological program could always choose to make requests maliciously. The scheme prevents such an occurrence since the random step cannot be predicted by any program. The algorithm is described for the case of a hypercube, but is actually more general. On a d-dimension hypercube, the probability is less than $(.74)^d$ that more than $8d$ steps are taken for the routing although, in practice, approximately $2d$ steps suffice. *Many-one* routing allows the processors to produce requests that are not unique, and also, every processor need not produce a request. The efficient implementation of techniques to broadcast information obtained by one processor to several others (very similar to simulations between models of the PRAM), and using sorting as well as partial routing, this kind of routing can be achieved probabilistically in $O(\log n)$ time.

4.3 Some Implemented Designs

There is a wide variety of implemented architectures and designs available — both at the prototype stages and at the commercial levels. A broad survey that includes several recent designs is available in [AG89].

SIMD parallel architectures are usually easier to build and their control mechanism is simpler. They usually appear in the form of vector processors and the related systolic arrays.[11] Generally speaking, the applications that can be profitably run on these systems have well-structured communication and dependency patterns (see Section 7), and have similar operations that need to be performed on several sets of data. A reasonably comprehensive list consists of the Cray series, CDC Cyber, IBM 3090, NEC SX, UI Illiac IV, IBM CF11, MIT/TMI CM, Goodyear MPP, ICL DAP, CMU/GE Warp, and Purdue/UW CHiP. Note that a globally synchronized clock, perhaps emulated, is required, and these machines have been the most successful among the parallel computers deployed so far. Also, the level of the granularity of the tasks is small, and hence, a fine degree of parallelism is possible. This is partly due to the efficient communication that can be effected in most of the designs.

Parallel architectures of the MIMD variety are of several kinds. The Caltech Cosmic Cube, and the related iPSC, FFS/T, NCube/ten are examples. So are Tandem (bus-based), Teradata, DADO, PAX,

[11] Pipelined processors are relegated to sequential computation for the purposes of this report.

of a network that does not achieve the minimum bound. That grids cannot sort so fast can also be obtained from the AT^2 bound for sorting described earlier, and the fact that $A = \sqrt{n}$. In fact, if $T = O(\log^k n)$ then the AT^2 bound implies that $A = \Omega(\frac{n^2}{\log^{2k} n})$. Note that this implies that the area on a chip is dominated by the wiring rather than processing elements for fast sorting networks. Using bounds that involve volume — similar to AT^2 bounds — it can be shown that $V = \Omega(n^{3/2})$ which implies that using all three dimensions will also not solve the high overhead of wiring. Thus, for large ensembles of processors, the communication becomes slower since the wires are unable to take short routes.

A system with several processors is usually configured in one of the following two ways.[10] The shared-memory (or dancehall, or multiprocessor) configuration consists of a set of processors on one side, a set of memory units on another, and the two connected to each other by an interconnection network. The algorithms developed for shared-memory architectures (e.g., PRAM algorithms) are directed toward such configurations. The other configuration consists of several autonomous processor and local memory unit pairs that are linked to one another in an interconnection network, usually without additional intermediate elements. This configuration is known as a distributed memory (or boudoir, or multicomputer) organization. The logical organization of the two is not so different as might appear at first. Shared memory configurations with end-around interconnections between processor and memory units, as is often provided, results in an organization that is essentially the distributed memory organization. And, using routing techniques to be discussed below, the distributed organization resembles the shared-memory configuration. In fact, algorithms written for either may be executed in the other with only minor modifications [AS88] — although the differing communication costs may render them impractical for use.

As remarked, the two configurations have few logical differences and the real differences lie in the physical aspects. The communication time on multicomputers is far greater than that on multiprocessors. This is in part due to the fact that the former being much more scalable [AS88], have several more processors in practice, and consequently, several interconnections transcend chip boundaries. Multiprocessors, though permitting fast interprocessor communication, are hindered by economic and technological factors that prevent their having large processor ensembles. The interconnection networks in multiprocessors use switches that could be quite sophisticated. In fact, the switches become small processors themselves which makes the two configurations quite similar.

4.1 Interconnection Networks

Interconnection networks are concerned with the delivery of the maximum of relevant data quickly, reliably, and at a low cost between elements of a processor ensemble. The performance criteria for these networks include *latency* (transit time for a message), *bandwidth* (amount of traffic supported), *reliability* (existence of redundant paths, and necessary algorithms), *functionality* (the sophistication of the switches), *connectivity* (the degree of each node), and *hardware cost* (fraction of total cost of system represented by the network). The choices available to build a network include the topology (static or dynamic with subclasses in each), synchronous or asynchronous operation modes, switching methods (packet or circuit switching), and the type of control to route messages (centralized or distributed). It is not possible to detail these criteria and techniques — the combinations are many, and a large amount of literature exists. Any standard parallel architectures text discusses these issues.

Static topologies are useful for algorithms that have comparatively determinate communication patterns. Some examples in decreasing order of minimum latency, maximum bandwidth per processor, connectivity, and wiring costs, are the all-to-all, hypercube, cube-connected cycles, and tree networks. Dynamic topologies are more suitable for algorithms without regularly structured or determinate communication patterns. They use switches to effect the communications, and are more general-purpose. Several examples are available in this category as well; in decreasing order of minimum latency, maximum bandwidth per processor, connectivity, and wire economy are the cross-bar, bus, and the multistage networks. For switch costs, the multistage network lies between the inexpensive bus and the more expensive crossbar networks. There are a

[10] Approaches such as pipelining or vector arrays may be considered to be extensions of uniprocessor systems. Issues germane to uniprocessors apply to them, and it is generally the case that well-structured algorithms (e.g., those involving matrix operations) that are fairly specific to these architectures are the ones relevant for study in their context.

extent in the computational theory for integrated circuit technology. Let the solution to a problem require area A, and the time taken to solve it using the chip be T. Using area-time diagrams and information flow arguments [Ull84a], it is possible to derive lower bounds on the AT^2 characteristics of several problems. For example, sorting n numbers, each of $(\log n + 1)$ bits, requires $AT^2 = \Omega(n^2 \log n)$.[8]

The area constraint implies that efficient ways to lay-out the circuit elements must be examined. Regular structures are easier to lay-out, and they scale-up more easily as well. A high degree of connectivity between circuit elements is inhibited by at least two factors. Firstly, the pin limitation constrains the interconnectivity of elements that lie on different chips. Note that even if the pins are used in a multiplexed manner, the rate of information flow across the boundary of a chip is limited by the communication rate on a pin and the number of pins. Secondly, the area on a chip is usually dominated by the wiring rather than the processing elements [Sei84]. Thus, increasing the number of interconnections increases the wiring overhead and hence, reduces the number of circuit elements that may lie on a chip. It may be concluded that sparse, regular interconnections between elements, especially considering processing elements, will be the dominant pattern for parallel computer architectures. Again, newly emerging technologies are unlikely to alter the situation significantly.

Graph separators are very useful in computational VLSI. A graph G with n nodes is said to be $S(n)$ separable if $n = 1$, or for $n = n_0$, at most $S(n_0)$ edges exist whose removal separates G into two $S(n)$ separable graphs $G_1(n_1)$ and $G_2(n_2)$ with $n_1, n_2 \geq n_0/3$. Separators assist in VLSI layout algorithms by splitting a graph that is to be laid out into smaller, more manageable graphs that are recursively laid-out. Upon a return from a recursive call, the two graphs are adjusted to make the necessary connections implied by the separator.

The communication efficiency of algorithms that are implemented on an architecture that has processors interconnected in the topology of some graph is related to the separators of the graph as well. Broadly speaking, a graph with a small $S(n)$ value will hinder algorithms that require a high degree of communication — a high degree of message traffic between the groups of processors in G_1 and G_2 with a small $S(n)$ value illustrates this nicely. In [Ull84b], this notion is formalized further in the description of *communication flux*. Consider a hypergraph on n nodes where the nodes represent processors and the hyperedges denote the interconnections. A hyperedge with several nodes denotes that all the corresponding processors can use the same interconnection amongst themselves (e.g., as in a bus). A hyperedge is assumed to be able to handle all the communication that its constituent nodes can generate. Consider a family of processor networks $\{N_i\}$ where $|N_j| > |N_k|$ for $j > k$. If $|N_i| = n$, the communication flux $f(n)$ for $\{N_i\}$ is given by $f(n) = \min_S \{q/|S|\}$ where $q = \sum_{hyperedges}(\min(n_1, n_2))$ where $n_1 \in S$, $n_2 \in S'$, and $|S| \leq n/2$. The flux is very similar to the notions of electrical and magnetic flux prevalent in physics. Flux is related to the amount of communication that is needed by a family of circuits: it may be defined as the time needed to supply each node with a quantum of data. That is, the flux for a family is the minimum of the fluxes considering every subset and complement pair from the nodes of the graphs in that family. Use of this definition shows, for example, that binary tree graphs have $f(n) = O(1/n)$.

To illustrate the use of communication flux formulations, consider fast sorting networks.[9] Such networks, given one element per processing node, can sort the values in polylog time, say $O(\log^k n)$. A fast sorting network must have $f(n) = \Omega(\frac{1}{\log^k n})$ — otherwise, by placing all elements destined for S' (for a suitable S) in S, one could coerce a flow of information across the $S - S'$ boundary that would exceed the volume implied by the time bound. Conversely, AKS sorting networks are able to achieve the fast sorting time bounds, and therefore must have flux function values meeting the minimum requirements. Other examples include the butterfly and hypercube networks. A grid organization of processors with a flux function of $\sqrt{n}$ is an example

[8] By taking a cross-section parallel to the time axis, with the area plane being cut on the shorter side, it is possible to apply information flow arguments relating the two dimensions of the cross-section area: T and $\sqrt{A}$. Hence, AT^2 characteristics are obtained by squaring. Cross-sections on the other axes yield AT characteristics. In the context of area and time considerations, as is often the case with parallel computation, probabilistic techniques prove much better. However, the results are not guaranteed in the usual sense of probabilistic algorithms. It may be remarked here that parallel algorithms are complicated enough to justify the use of probabilistic techniques which are usually simple. Also, the chances of error can be made very small — i.e., errors are 'impossible' for any practical situation.

[9] The importance of fast sorters goes beyond sorting alone (e.g., see routing further ahead). In the context of communication, fast sorting networks are those that support large amounts of communication between subsets of processors.

summation problem has the speed-up $S_p(n) = \Theta_{HL}(p)$ and efficiency $E_p(n) = \Theta_{HL}(1)$. This is achievable when l items are added sequentially within a processor, and the remainder of the partial sums are added in the usual tree fashion. When the times for the local and global computations are close, the desired complexities are achieved: $S_p(n) = \frac{T_1(n)}{T_p^{local}(n)+T_p^{global}(n)}$ and $T_p^{local}(n) = \Theta(l)$, $T_p^{global}(n) = \Theta(\log p)$; that is, when $l = \Theta(\log p)$.

In the context of data movement mentioned above, studies in the permutation of data are important. Related concerns of sorting and routing are extensively studied problems. We shall examine some aspects below. In this regard, note that data movement patterns that are known *a priori* can be performed quite fast. However, those movements that are determined dynamically, and hence cannot be analyzed in advance, have algorithms with poorer performance. In using algorithms as subroutines, therefore, these considerations have to be taken into account, which is unlike the case for sequential computing, or in many cases of PRAM algorithms.

4 VLSI and Architecture

The study of parallel architectures must begin with integrated circuits. Parallel architectures not subject to integrated circuits technology resemble message-passing distributed systems due to the large communication delays [AG89, AS88]. Components integrated on a chip are able to communicate among themselves much more rapidly and permit the development of tightly-coupled systems. However, even with the use of integrated technologies, it is not possible to place an arbitrary number of processors in close proximity, and it is also not possible to provide communication without resorting to message-passing techniques. Using some of the new technologies (e.g., optical computing elements [Fei88]) is also unlikely to permit building machines that are fundamentally different as compared to the more traditional methods.

In VLSI manufacturing, both the technological and economic factors play a role. While it may be technologically possible to develop certain kinds of chips, the cost factors preclude their manufacture. Hence, in the ensuing discussion, the difficulty in producing certain kinds of integrated circuits is a result of either or both these factors. Without delving into excessive details, we begin with some relevant issues in VLSI architecture from [Ull84a].

The area of a VLSI chip is a coveted commodity. Very large chips being infeasible, efforts to reduce the required area for a chip are important. Likewise, designs that imply the need for a large chip area are not very useful. A related aspect is the pin limitation. The number of I/O pins possible on a chip is small, and these pins are usually constrained to be located at the chip boundaries. One of the reasons is that pins drive external signals, and hence, require high power dissipating driver circuits on the chips. In general, since the perimeter of a chip increases only as the square root of the chip area, doubling the number of pins implies the quadrupling of chip area. For all practical purposes, integrated circuits should be regarded as units with, at best, a few hundred pins. Even the newer technologies such as Wafer-Scale Integration and Opto-electronic devices may alleviate the problem only slightly.

The propagation of a signal outside a chip takes a long time as compared to that within the chip. With large ensembles of processors, the chip area constraints imply that the processors cannot be located proximally and interprocessor signals may have to traverse sizable distances. Such traversals, besides being limited by the propagation speeds, are also subject to dissipative effects. Hence, the signals have to be boosted after short intervals by circuit elements that act as relays — and these elements add to the delay due to finite switching speeds. Consequently, signals such as a global clock synchronization signal, are infeasible unless a fairly substantial slowdown is tolerable. Thus, in case of a global clock, the clock rate would have to be quite low. In fact, in large synchronous systems, the synchronous clock for the entire system is actually an emulation of a clock signal by means of tokens where the receipt of a token signifies a tick of the global clock. That is, clock synchronization is achieved in essentially the same manner as in distributed systems.

The above discussion indicates that the solution for a given problem integrated onto a chip is likely to exhibit a relation in the area of the circuit logic required and the time taken for the solution. That is, it may not be possible to simultaneously optimize the area and time. This notion has been formalized to a certain

3.3 Distributed Systems

At the other end of the spectrum, in loosely-coupled systems with message-passing primitives for communication, the criteria used for the design of algorithms is substantially different. The spatial separation of processors is substantial, interconnection networks that deliver messages have large latency times, and the memory is distributed into local modules with no shared stores. Such systems require algorithms that minimize the overhead of costly messages — often at the cost of comparatively cheap processing time. The systems exhibit varying degrees of reliability, connectivity, and asynchrony since the synchronization costs of maintaining global clocks is prohibitive. The reason that we are concerned with such systems, and their the associated algorithms and complexity measures, is that real parallel computing systems exhibit several similar characteristics. The following largely follows [LG89, LL89].

Generally, distributed systems are concerned with processes rather than processors, with the former run by some scheduling procedure on the latter. Distributed algorithms are characterized by: independent inputs to processes with the output at multiple sites; several processes executing under separate control; processes starting at different times and running at different speeds; processes failing; each process being aware of only a part of the system; The motivating factors of study in distributed systems are: precise statements of canonical problems suitable for research; precise and careful descriptions of algorithms to solve the problems; rigorous proofs regarding the algorithms; complexity analysis; and impossibility results.

In most algorithms, the complexity analysis is concerned with the dominant cost factor of the messages. In many cases, the efficacy of an algorithm is judged by the number of messages that it generates [LL89]. The measurement of the time-complexity of such systems is not entirely obvious. Several approaches are common. For example, in some cases, algorithms are compared by their round-complexities (see Section 9). In others, the maximum time-interval for any one event is taken as the basic unit [LG89]. These methods take into account the asynchrony of the systems under study. The main goal of the research is the development of correct algorithms, and a secondary goal is their efficiency which is reasonable for the systems under the purview of distributed algorithms. However, to be of use to parallel computation, this must be remedied by making efficiency considerations more prominent.

The correctness proofs for the distributed algorithms can become very complicated. Among the factors contributing to this are the interleaved actions, inherent non-determinism, actions at several sites, failures, reactive computations etc. In fact, rigorous study of these systems is not only desirable but quite necessary as well.

For the purposes of pragmatism in parallel computation, it appears that the available and foreseeable architectures exhibit characteristics of both tightly-coupled and loosely-coupled systems. This leads to the conclusion that much of the research that needs to be done for these systems will not rely entirely on the extreme cases, but the available results in the extremeties may find good use when taken into consideration together.

3.4 Choosing Good Algorithms

Although the best choice of the algorithm for a particular application depends on several issues, it is generally the case that the algorithm with the best asymptotic bounds is suggested — with some important caveats. The size of the input makes a difference (e.g., simple linear sorting suggests itself for a small number of input values); imprecision in asymptotic behavior (e.g., the Ajtai-Komlos-Szemeredi sorter), worst case versus average case analyses, use of algorithms as subroutines (whereby, in parallel machines, data movement becomes important), and I/O factors are examples of concerns that motivate a choice.

In the choice of parallel algorithms for PRAM computations, some approaches specify the relationship between p and n. For example, as mentioned earlier, in practical situations we have $n \gg p$. Another factor, $l = n/p$, called the *data loading* factor, is used in such analysis. Algorithms are designed and the analysis is done using l. In this context, a dependent-size problem is one where n depends on p, and the independent-size problem is similarly defined. An important factor in a message-passing model is the initial data-distribution. In independent-size problems which are practical importance, the complexity measure may be provided as an expression involving l. In the case that we reach a *heavily-loaded* situation which is denoted by the subscript HL, where $n \gg p$, it may be possible to obtain a simpler description of the complexity. For example, the

sequential systems, is available.

3.2 Relevance of the Class NC

The creation of the class NC is an *ad hoc* decision about the measure of parallelizability of algorithms. In sequential algorithms, the development of algorithms is not restricted to the class P; better algorithms for NP-hard problems are also sought. Similarly, problems outside the class NC are not devoid of interest in the parallel context.

The class NC is, in some sense, too absolute a classification. Assuming that parallel systems will be deployed only in situations where sequential computing fails to suffice, it is important to examine parallel algorithmic efficiency vis-a-vis the sequential efficiency. In [KRS88], among others, some alternate concerns are studied as well. The emphasis is placed on the speed-up over sequential methods achieved through parallelism and the efficient utilization of processors. Thus, an important point of practical concern is raised, and the domain of interest shifts to achieving reasonably fast algorithms on a reasonable number of processors which contrasts with the class NC where a large number of processors are required to execute very fast algorithms.

Based on relative speed-ups and efficiency, [KRS88] identifies several classes of interest. In the following, a problem is solved by a sequential algorithm that runs in time t, and a parallel algorithm that runs in time T on P processors. The parallel algorithm: is polynomially faster if $T \leq t^k$ for some constant $k < 1$; is polylog faster if $T = \log^{O(1)}(t)$; has constant inefficiency if $T.P = O(t)$; has polylog inefficiency if $T.P = t.\log^{O(1)}(t)$; and has polynomial inefficiency if $T.P = t^{O(1)}$. By combining the speed-up and efficiency criteria, six classes are identifiable. The model of computation that this classification is based upon is the PRAM. Mainly complexity issues and containment relations among the classes have been studied. The algorithms are developed for the PRAM model, and only the focus is shifted from the class NC to more practical classes.

An example of closely related work is [RK89] where the focus is on the scalability of parallel algorithm based upon *iso-efficiency* functions. For a parallel algorithm, given a particular number of processors p, and a number e where $0 \leq e < 1$, there is a number w that must be exceeded by the problem-instance size n to achieve an efficiency of e. For a given architecture, if w needs to grow at the rate of $\Omega(f_e(p))$ to maintain the efficiency e as the number of processors p grows, the function f_e is the iso-efficiency function of the algorithm for the particular architecture. This research is more general as compared to similar research using PRAM models since it delineates between specific architectures. Overhead times are evaluated for some architectures by assigning values for the communication times in the architectures under scrutiny. Several experimental benchmarks culminate in the scalability analysis as described. However, much of this work emphasizes fine-tuning algorithms to particular architectures which, on occasion, may well be unavoidable. In a similar manner, developing algorithms for the PRAM model could also be regarded as fine-tuning with a different set of parameters that are considerably simpler to work with.

The importance of the number of processors and the speed-up is closely related to the problem-size as well, and this complicates parallel algorithm analysis further. Let the speed-up be given by $S_p(n) = T_1(n)/T_p(n)$ as usual. The efficiency $E_p(n) = S_p(n)/p$ normalizes speed-up to lie between 0 and 1. The effects of increasing p and the effects of increasing n are both important enough to warrant close study. Depending on the relationship between p and n, one algorithm may be preferred over the other.

Consider the Minimal Spanning Tree problem in the context of PRAM algorithms. Sequential $\Theta(n^2)$ algorithms are known. There is a parallel algorithm with $\Theta(\frac{n^2}{\log^2 n})$ processors requiring $\Theta(\log^2 n)$ time — the speed-up implies an efficiency of 1. However, assume that we *have* n^2 processors and want to make full use of their potential instead of wasting their time idling. A different algorithm achieves a better time of $\Theta(\log n)$ but a lower efficiency of $\Theta(\frac{1}{\log n})$. Even the use of scheduling p processors on the algorithm yields $\Theta(n^2/p)$ processors, $\Theta(p \log n)$ time , and the efficiency remains unchanged at $\Theta(\frac{1}{\log n})$. Thus we cannot hope to simply take an algorithm that appears good for some p and n values and apply task scheduling to obtain efficient results. To add to the problems, there exists yet another algorithm that finds the solution in $\Theta(\log n \log\log n \log\log\log n)$ time and $\Theta(\frac{n^2}{\log n \log\log n})$ processors. The best algorithm would take all the three and, depending on the number of processors p with respect to n, switch between them.

complexities are the ones that are of real interest. Fortunately, most problems of interest in P have such complexities. For NP-hard problems that are unavoidable, approximation methods that are tractable are often resorted to [AHU74]. This delineation of tractability is of use in the parallel context as well where the modest potential of parallel computation implies, at least in the context of problem scalability, that the interesting problems lie in the class P.

The dominant model of parallel computation is the PRAM [Coo85, KR88, GR88]. It consists of an unlimited shared-memory connected to several processors that run in lock-step synchrony. Each processor is a RAM with its own program and store. Memory-access, and any instruction from its available repertoire, takes a processor a single unit of time. Depending on the availability of concurrent reading or writing of the memory cells, CRCW, CREW and EREW PRAM models can be distinguished. The input instance of a problem is placed in the first few cells of the memory and the output is required is made available by the algorithm in a similar manner. The important measures of complexity are the time-complexity measured by the number of steps required to solve the problem, and the processor-complexity which is the number of processors required to compute the result — both as functions of the problem-instance size.

There are several admirable features of the PRAM model. It is a simple, straightforward extension of the RAM model. It allows algorithm-designers to concentrate on the parallelizability of the problem at hand. The model allows the development of algorithms without getting cluttered in the details of the target machines. As a result, a large number of interesting algorithms have been developed for the model as evidenced in [KR88]. Further, the model is equivalent to several other reasonable theoretical models.

Most of the algorithms developed for the PRAM model are for problems categorized by the class NC. Membership in this class requires a problem to have a polylog-time algorithm that runs on a polynomial number of processors. In a manner similar to the sequential case, NC is contained in P, and P-hard problems provably cannot have algorithms that will permit membership in NC (unless some very unlikely results hold). Theoretically, the class NC delineates the problems that have very fast parallel algorithms, and could be well regarded as the acceptable notion of parallel tractability to the theorists. As in the sequential domain, tractability is an *ad hoc* notion. The purpose of the hardness and completeness results in complexity theory is to avoid attempting better algorithms in those cases where it would be futile to do so [AHU74]. However, it should be noted that in the case of sequential computing, all the algorithms, whether tractable or not, are developed on the RAM model which promotes easy development, and the model reflects the real costs reasonably well.

In the parallel situation, it has been a general lament that the PRAM model is inadequate [AG89, OCP88, Qui87]. In fact, the model may actually be detrimental by misleading designers as the following simple example from [Sny86] demonstrates. Assume that the maximum of n elements needs to be found on a CRCW PRAM using a similar number of processors, that is, $p = O(n)$. Valiant's optimal algorithm for this model runs in stages: at stage s, the $n(s)$ distinct values are partitioned into the fewest number of r sets $S_1, S_2, \ldots, S_r$, where for all i, j, $|S_i - S_j| \leq 1$ with $p \geq \sum_{i=1}^{r} C(|S_i|, 2)$. For each distinct pair of elements out of each S_i, a processor determines the smaller one and annuls the corresponding entry in an appropriately initialized boolean array for the set S_i. Thus, for a set S_i, it takes $O(1)$ time to find the maximum. Analyzing this algorithm yields an $O(\log \log n)$ time algorithm which is optimal.

However, the assumption that a unit time memory-access is available to the processors proves to be the undoing of the algorithm. Even from the purely physical viewpoint, p processors require $\Omega(p)$ volume, and hence, $p^{1/3}$ access-time. Similar problems of a lesser degree are present in the sequential domain as well [ACS87]. Since any reasonable interconnection between processors and memory elements requires an access-time of $\Omega(\log p)$, the running-time of the algorithm as provided would yield a bad running-time of $O(\log p \log \log n)$. And yet, a reasonable architecture with a binary tree embedded within its interconnection structure can achieve a running time of $O(\log n)$ by simply percolating the maximum value to the root processor. Note that even in such architectures, the time taken to access an arbitrary location is $\Omega(\log p)$. Thus, by using the CRCW PRAM for our model, we are misled, and the modest potential available makes this a serious error. Although the RAM has similar drawbacks in that it does not reflect the memory access time accurately, in the domain of practical applications, the results are satisfactory. Moreover, there is no question of a modest potential since the sequential method is very cost-effective. In parallel systems, the high costs of building the systems will obviate their use unless significantly superior performance, as compared to

Not all problems are amenable to parallel solutions, and it is important to be able to recognize those that are. Furthermore, given that we are dealing with a real-world situation with several constraints on the computer technology, it is imperative to recognize which are the better algorithms for a particular problem. To improve the performance of the algorithms, use of special-purpose parallel systems may be considered, or general-purpose computing systems could be used. In programming the non-sequential machines, how much responsibility the programmer should have to exploit parallelism must also be considered since the remainder has to be borne by the system.

These issues are not yet fully resolved. Meanwhile, some of the celebrated objections to the efficacy of parallel computation (e.g., Grosch's Law, Amdahl's Law, Minsky's Conjecture etc.) have equally reasonable counter-arguments that indicate that this area of research is definitely worth pursuing [Qui87]. Furthermore, better performance requirements leave open no other choice.

The above discussion pertains mainly to tightly-coupled systems with a sizable number of processors. However, for loosely-coupled distributed systems, most of these issues are relevant as well. Distributed computing research is closely related to such systems [LG89]. The programs are usually viewed as a collection of communicating and cooperating processes, and the system is regarded as being asynchronous. Considerations for such systems include failures, non-determinism, correctness etc. If parallel computing architectures with a large number of processors is considered, most them are likely to have several characteristics of distributed systems such as message-passing for communication, partial asynchrony etc.[AG89, AS88]. Hence, the research in parallel computing techniques can make use of extant results from distributed computing research.

Research in distributed algorithms has arisen mainly from concerns in systems-related areas such as operating systems and database management. In these, the issues relate to correctness, portability etc. rather than efficiency.[6] The algorithms to control processes very often have large completion times, and using them directly to control systems in parallel computation would generally yield bad results. Hence, for the control of parallel computation systems, hardware support has been considered (e.g., in the form of the atomic test-and-set operations). In turn, these developments may provide further research avenues to explore in distributed computing research *per se* [LG89].

Thus, parallel computation brings together a large group of researchers since the concerns overlap considerably. It is important to view this research from the angles of hardware, software, theory, etc. to be able to develop easily usable parallel systems [OCP88].

3 Theoretical Aspects

This section discusses those aspects of theoretical research that pertain to the subject of this report. At the very outset it would be prudent to point-out that the section does *not* purport to provide an overview of the entire area of theory. Also, the matter discussed here is related to the more firmly established theory and does not deal with the more recent developments.

3.1 The Class NC

The theory of sequential algorithms is well-established and accepted. It is based upon the straightforward RAM model of computation which is equivalent to several other reasonable models [AHU74]. The RAM allows simple development strategies for algorithms and also for programming. Thus, most widely-used programming notations have the RAM as their underlying model. The costs incurred by the algorithm execution in the RAM are reflected reasonably well in real computations, and *vice versa* (see Section 6).

Notable in the development of sequential computation theory is the notion of NP-hardness [AHU74]. Problems that have deterministic polynomial-time solutions are regarded *ad hoc* to be tractable and form the class P.[7] Problems that are considered NP-hard can be proved to require non-polynomial-time algorithms (except in the unlikely situation of P=NP). For practical concerns, problems with low degree polynomial

[6] Although performance issues are important, they have often taken a secondary place.

[7] Tractability is robust over reasonable models. Conversely, a model is reasonable if the tractable problems in the RAM model are also tractable in it...

program $\Phi(P)$ in the real model, in an algorithm-independent manner. In sequential computing, the models implied by the Pascal programming notation and the RAM abstraction provide an example of the two levels.

Algorithm costs, developed from cost functions for the operations, are used for comparison, and usually the asymptotic complexities are considered, and usually, the quantities measured are the time-taken, the space-used.[4] An order $\succeq$ could be considered on programs, where $x \succeq y$ means that program x is no worse than program y in performance for whatever characteristic that is being compared. Several desirable properties can be described using the order. The high-level model is *expressive* if for any low-level program Q there is a high-level program P that solves the same problem, and $\Phi(P) \succeq Q$. Thus, expressive models permit the design of good algorithms without a guarantee of their ability to allow analyzing the developed algorithms. A high-level model is *faithful* if it is expressive, and for any two high-level programs P and Q that solve the same problem, $P \succeq Q \Rightarrow \Phi(P) \succeq \Phi(Q)$. Although faithful models allow the comparative analysis of programs, the actual time taken on the machine may be poorly indicated. Even if a high-level model is not faithful, the cost function may be useful if it can be used to ascertain which algorithms are good in a practical sense. Thus, a high-level model is *restrictive* if for any high-level program P and a low-level program Q for the same problem, there is a third program R such that $R \succeq P$ and $\Phi(R) \succeq Q$. The idea is that it is desirable to obtain practically good algorithms from theoretically good ones.

Usually, only a few of the above desirable traits are present in the high-level models. It is not necessary to find the actual costs incurred by every algorithm using the transformations, and results obtained in a few generic cases may be extrapolated to obtain the costs for most other cases. This is one of the most important reasons for considering such abstractions.

2.4 Issues in Parallel Computation

Parallel computation requires the examination of several more issues as compared to the sequential domain. The organization of this portion is based on [AG89, OCP88, Qui87].

Non-sequential computing has the key ingredient of using several processors. The number of processors in the system and the power of each are both important, and this is often limited by the available technology, cost factors etc. (see Section 4). Systems may be configured using a few powerful processors (e.g., Cray or IBM 3090) or numerous small processors (e.g., the Connection Machine or systolic arrays)? Technological and economic factors make it infeasible to have a system with numerous powerful processors. In this context, the overall goal of the system may dictate the choice: for example, if the system-throughput is the criterion then the powerful processor systems are indicated. The primitive operations desirable and feasible in the processors are also of concern. The memory organization, as in shared-memory or distributed memory, needs examination. For the related question of I/O, it is unclear how to effect it in the systems. A dedicated I/O processor which requires all the input information to be sent to relevant processors dynamically could be implemented, or several processors in the system could engage in I/O activity. Similarly, whether the instructions of the algorithms should be stored at each processor, or whether this information should be sent by some specific processor to the others, is unclear. Thus, it is apparent that several more issues are of concern in parallel computation.

Other design decisions are required to handle the communication among the processors. Communication may be effected by shared-memory or through message-passing. The protocols used for communication, the interconnection patterns, the electronic technology used for the inter-communication mechanism etc., are all pertinent questions that affect the correctness and efficiency of the implemented algorithms.

Cooperation among the processors is also quite important. How synchronization is achieved and the primitives provided for it together affect the efficiency significantly.[5] By increasing the granularity of the tasks, the relative effect that such requirements have on the performance is reduced. A natural question that concerns the factors that guide the choice of the granularity of the tasks. It may be the case that these decisions are made by the operating systems. If so, how should these systems be designed for peak efficacy?

[4] The comparisons need not be limited to asymptotic complexities. The models simply provide a *means* for any comparisons; what is to be compared depends on the applications.

[5] Research on synchronization and related topics has concentrated on the aspects as they related to operating systems rather than parallel computation.

model of computation are available at similar costs in the implemented programming notation. The notation strongly influences the form and efficiency of the implemented algorithms. For example, the efficiency of the codes for executing a matrix inversion differ significantly between development using Fortran and Lisp. The problem occurs because constructs that are not available have to be compensated for by circumspect programming. In fact, the translator may do this even if the requisite primitive construct *is* available. Although every notation favors some subset of algorithms, there is strong evidence that essentially sequential notations augmented superficially with parallel constructs favor inherently sequential algorithms [Sny86]. This is an important consideration since it has been argued that the program is the most important concern in parallel computing [Sny86, OCP88].

The machine itself may differ significantly from the models of computation but should provide the more important capabilities suggested by the models. For example, in the sequential domain, the low cost of memory access is usually available (however, see Section 6). Thus, inspite of widely differing machine architectures, the basic capabilities implied by the RAM model are usually provided for.

2.2 A Modest Potential

As in the sequential case, it will be important to develop universal methodologies for parallel computing. So far there is no consensus in sight. Before delving into details, it would help to see what it is that we hope to achieve using parallel systems.

Adapting from [Sny86], the important potential for non-sequential computing is in compute-bound problems rather than the I/O-bound ones.[3] These problems usually have polynomial-time sequential algorithms, $O(n^x)$ for $x > 1$. The goal of using parallel machines is often to solve larger problem instances within a fixed time budget. Consider an algorithm that requires $t = cn^x$ sequential steps. Assume, very optimistically, that the maximum p-fold speed-up is achieved using p processors. Keeping the time budget constant, let the increase in the problem-size as a result of using the p processors be given by a factor of m, and thus $t = c(mn)^x/p$. Hence we have $m = p^{1/x}$ which implies that an increase in two orders of magnitude for an $O(n^4)$ sequential algorithm requires, very optimistically, at least 10^8 processors! And we have not even begun to consider the significant overheads such as communication latency etc. It is clear that considering some of the more computationally intensive problems brings into focus the very modest potential available in parallel computation.

The argument presented indicates that the overheads in parallel computation critically affect performance. Thus, it may be expected that an acceptable abstract model of computation, when developed, will most likely be quite close to the underlying architectures than in the case of sequential machines.

2.3 Parallel Computation Models

This subsection draws mainly from Snir's position paper in [OCP88]. Among the several factors that dictate the choice of a model, the performance impact of a choice is one, and it is well gauged by complexity theory. The division of labor between programmers, compilers, hardware assistance etc. is essentially settled for sequential computation and a similar consensus is needed for the parallel computation realm.

The model of computation plays a significant role in both the design of algorithms as well as the programming aspect. A model specifies the basic operations available and the rules to compose them into valid programs. Similar to the composition rules used to specify program syntax and semantics, it is desirable to find cost functions that compose over the different portions of a program.

Several 'levels' of models may be considered depending on the level of the abstraction and the purposes that the model is expected to serve. Consider two models of a given machine at two levels: the 'virtual' level, and the 'real' level. The first is a high-level programming model which is an abstraction that is of use for programming convenience. The second is a low-level hardware model that serves as an accounting device for the basic operations to be performed by the actual machine. The real model need not actually model the hardware beyond the purposes of accounting. Translators map a program P in the virtual model into a

[3] Snyder's paper is dated 1986. Several different architectures and technologies have vastly improved data transmission and retrieval options (e.g., optical disks, optic fibers, disk arrays etc.[AG89]). In database systems, a typical I/O-bound problem domain, parallel systems are becoming increasingly significant.

skeptical scrutiny. That a definite problem does exist, is made apparent in this report, and is motivated mainly by [OCP88, AG89, Qui87, Sny86]. An example in this regard that is evident is the difference in complexity measures used to determine the efficiency of algorithms in the models for parallel computation on one hand, and distributed computing on the other. The former uses time complexity while ignoring the effects of communication overheads while the latter concerns itself with just the reverse [KR88, LL89]. The justification is that the two apply to different types of systems. However, real systems that have a sizable number of processors, actually lie somewhere between the two extremes implied by the complexity measures [AG89]. Hence, it may be appropriate to develop models that have features of both.[1] On the other hand, this incurs a loss in the simplicity of the models which is detrimental to the easy development of algorithms.

Understanding efficient computation on multiprocessor and multicomputer systems is much more difficult and involved as compared to uniprocessor, sequential systems. It will become clear in this report that the design and analysis of algorithms in the more complicated case is not easily abstracted from the issues of architectures, programming notations, systems, etc. Hence, we examine some of the issues raised when such 'extraneous' factors are considered as well. However, time and space constraints limit the research that can be described. Furthermore, owing to the lack of consensus in the research community [OCP88, AG89], the subject defies a thoroughly organized presentation.

This report is arranged as follows. Section 2 discusses some general issues germane to the field of highly parallel computing. The section motivates the reasons for examining the factors that constitute the following sections. Section 3 examines the more theoretical aspects of the work done on parallel and distributed algorithm design. VLSI and architecture constraints affect the development of algorithms — which is the subject of Section 4. Since real parallel computers have features in common with distributed systems, Section 5 considers some of the interface areas. The next few sections study how theoretical work is being re-examined to account for real-life constraints imposed by technology. Section 6 describes some efforts to reconcile shared-memory models with distributed memory models. Section 7 examines different efforts to account for technology-imposed constraints of communication and retrieval overheads in the design of algorithms. Section 8 examines models of asynchronous computing that have been proposed recently. Most parallel algorithms can be conveniently denoted as DAGs, and research related to the efficient execution of such structures on real computers is examined in Section 9. Some systems-oriented aspects of parallel and distributed computing are examined in Section 10. Section 11 concludes the report.

2 Basic Issues

We begin with a brief overview of the issues germane to sequential computing. This will make it easier to develop similar issues in the case of parallel computing.

2.1 Sequential Computing

Comparatively simple guidelines exist for the development of efficient computing in the sequential case. An algorithm is developed, it is then encoded in a high-level programming notation. The resulting program is translated into machine code for a target machine.[2] A closer look at these reveals several details [Sny86].

The algorithm itself is developed with respect to a reference machine-model which is the model of computation. The aim of developing a good algorithm is to obtain time or space efficient solutions for the problems at hand. The model of computation, with its associated capabilities such as the primitive operators or memory access times etc., permits a systematic development, and guides the choice among possible designs. For the algorithms to perform well on real machines, the abstract model should be realistic enough to embody significant costs and capabilities. On the other hand, to ease the design and analysis, it should abstract away details that are unnecessary. The consensus on the model for sequential computing has been the RAM.

The program itself is encoded in a notation that is easy to work with but that can also be efficiently realized on real machines. In sequential computing, it is generally the case that the unit cost operators of the

[1] Henceforth, parallel computing is taken to mean parallel and distributed computing.

[2] Although this does not appear to be the case in situations such as declarative notations, the algorithmic details are hidden in the underlying system that realizes the notation.

Pragmatism in Parallel Computing*

Nandit Soparkar

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

Abstract

This report surveys some aspects of the multidisciplinary nature that typifies parallel and distributed computing. Clearly, the scope of such a study is so large as to be impossible to condense in one report. Hence, this paper examines a few key references that indicate the trend in the area of obtaining good models for the design of parallel and distributed algorithms that work well in practice, and that are also conducive to rigorous complexity analysis. In doing so, it is necessary to take into account several related aspects of parallel computation such as the architectures, the systems, and the experimental results. The manner in which the concepts and results from heretofore separately studied areas help in the understanding of parallel and distributed systems is, hopefully, brought into focus in this report.

1 Introduction

Undoubtedly, the major thrust of computing research today is in the harnessing of the power offered by several processing elements coordinating to reach a common goal. There are large efforts to this end in the areas of systems design, algorithms, programming notations, computer architectures, applications, etc. This report examines a portion of the research in some of these areas.

As in the case of sequential, uniprocessor systems, the design and analysis of algorithms is certain to be a key element in the realization of usable parallel and distributed computing systems. Voluminous literature, mostly developed over the last decade, is available. Problems that were studied in the sequential context have been studied for the case of systems with several processors. Furthermore, issues peculiar to parallel and distributed computing have also been closely examined by researchers. As a result, a number of basic methods and design paradigms, strongly fortified by foundational theoretical models, is available [KR88, LL89, LG89].

The approach used to develop the algorithmics for parallel and distributed computing systems is based on certain models of computation. Clearly, the results accruing from such studies is restricted to the models used, and their extension to other models is not necessarily assured. This research can be viewed in two ways. Firstly, it promotes a better understanding of the problem structure and motivates further research. That is, the study is quite academic and theoretical in intent which is similar to the more theoretical branches of mathematics. The second manner in which to view the research is more applied — as a means to understand the problem and devise solutions that are useful for actual applications in computing [OCP88]. The original motivation for parallel and distributed computing research was certainly based upon practical considerations. It is the second, the more pragmatic approach to the research, that is the intent of this report.

With the focus on pragmatism, a question has been raised in the past few years about the validity of the models of computation used, and the related complexity measures, as regards their ability to represent real-life computing systems. Consequently, the results, paradigms and issues that developed have also been under

*This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant No. IRI-9106450,and grants from the IBM, NEC, and Hewlett-Packard corporations.

PRAGMATISM IN PARALLEL COMPUTING

Nandit Soparkar

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-19 April 1992

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712