Sādhanā, Vol. 17, Part 1, March 1992, pp. 193–220. © Printed in India.

Scalable concurrent computing

NALINI VENKATASUBRAMANIAN*, SHAKUNTALA MIRIYALA[†] and GUL AGHA

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
*Present address: Hewlett Packard Company, 19111 Pruneridge Avenue MS44UT, Cupertino, CA 95014, USA
[†]Present address: Vista Technologies, 1100 Woodfield Rd Suite 108, Schaumburg, IL 60173, USA

Abstract. This paper focusses on the challenge of building and programming scalable concurrent computers. The paper describes the inadequacy of current models of computing for programming massively parallel computers and discusses three universal models of concurrent computing – developed respectively by programming, architecture and algorithm perspectives. These models provide a powerful representation for parallel computing and are shown to be quite close. Issues in building systems architectures which efficiently represent and utilize parallel hardware resources are then discussed. Finally, we argue that by using a flexible universal programming model, an environment supporting heterogeneous programming languages can be developed.

Keywords. Scalable concurrent computing; massively parallel computers; systems architectures; heterogeneous programming languages.

1. Introduction

Computers have penetrated into many branches of learning, industry and art. The increasing scope of application domains for computers has brought new demands for computer technology. At the same time, computers and programming methodologies have undergone dramatic changes in the past couple of decades, both in the conceptual view of a program as well as the physical layout of the machine. An analysis of these changes suggests that computer systems may be divided into five generations as shown in table 1 (Hwang & Briggs 1984). The foundational basis of the first four generations of computers is the stored-program concept or the *von Neumann* model. This paper describes fifth generation computer systems and focusses on developments in software technology necessary to realize it.

The outline of this paper is as follows. Section 2 describes the von Neumann model. This model has fundamental limitations for scalability which must be overcome. By scalability, we mean the ability of the system to display an increase in performance

Table 1. Classification of computer systems into generations.

Generation	Hardware advances	Software advances
First	Electromechanical devices	Machine language
Second	Transistors	Assembly, Fortran, Algol
Third	Small and medium scale integrated circuits	Multiprogramming
Fourth	Large scale integrated circuits	Advanced compilers
Fifth	Very large scale integrated circuits	Parallel programming

with the addition of computational resources. We then review concurrent computer architectures and their scalability characteristics. Section 3 reviews universal models for concurrent computing. Specifically, we describe 'actors', a scalable model of concurrent programming which overcomes the limitations of the von Neumann model. A number of problems, such as composability of independent systems, debugging and effective system management etc., arise in concurrent systems. Section 4 addresses problems in realizing concurrent computation, namely, new techniques for managing the computational resources of large numbers of concurrent processors. Section 5 discusses the use of actors to provide interoperability in programming environments – i.e., to provide a software technology which permits use of multiple programming paradigms. The final section discusses some ongoing research projects and future directions in scalable concurrent computing.

2. Concurrent computer architectures

In the von Neumann model, the hardware consists of a *processor*, a *memory module* (also called *storage*) and a *link* to communicate between them as shown in figure 1. A *program* is a sequence of instructions stored in memory and executed by the processor. In order to execute an instruction at least three communications have to pass through the link – the *instruction* (control), *operands* (data from memory to processor) and *results* (stored back in memory). However, having a single communication bus poses bandwidth limitations on the amount of information that can be transmitted between the memory and processor; this imposes a performance bottleneck, traditionally known as the *von Neumann* bottleneck.

The hardware components in a von Neumann computer operate by performing a

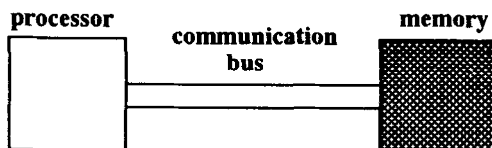


Figure 1. The von Neumann model.

sequence of transitions on a *state*[1]. The sequence of transitions is specified by the user through a program as a function of the state. Note that the programmer specifies not only the transitions which must be performed but also a *strict order* on the execution of these transitions by means of program instructions. In the model presented above, a single processor is the sole unit of computational power which results in the following two limitations:

Physical limits on how close components can get. Technological advances have led to the miniaturization of components as well as increase in processing speed. In fact, processor speeds have doubled every eight years.

Economic considerations on the cost of a single, highly sophisticated processor. One solution is to have a number of inexpensive units, all working simultaneously so that the overall throughput of the machine is increased. This is called *parallel processing.*

Industrial automation accompanied by the expanding size and nature of problems which have to be handled by computers has placed a premium on speed, efficiency and accuracy of computation. Weather prediction, nuclear physics, modelling physical and geological phenomena, genetic code mapping, space exploration, flight and space vehicle control, expert systems for medical diagnosis are representative problems. Applications run on computers may be numeric or non-numeric. In general, the use of computers is expanding from just number crunching to include *symbolic processing,* processing of non-numeric data such as pictures, text and sentences, and open information systems which are characterized by continuous information flow between autonomous units.

Nondeterminism is a key feature of the real world systems directly modelled in an open system. Nondeterminism has many sources. For example, when searching a very large database for some specific information, it may be necessary to explore many possible choices. It is not clear when we will arrive at a solution or which paths to investigate in the process. Searching many paths for a possible solution places a premium on performance. To satisfy this growing demand for performance we can exploit parallelism.

Computational models are inspired both by concern for representing real world problems and by architectural considerations. In building architectures which exploit parallelism, there is an obvious tradeoff between the number of processing units and their complexity. Variations in the degree of coupling of these two factors has given rise to a number of computational models for enhancing performance. At one end of the spectrum, there are machines with large numbers of very simple processors. The degree of concurrency is very high, but the power and complexity of each individual processing unit is low. The Connection Machine (Hillis 1985) is one such computer available in the market. On the other hand, there are multiprocessor architectures which use sophisticated processing units. The Encore Multimax is an example of a commercial multiprocessor machine.

2.1 *High performance uniprocessor architectures*

Supercomputers exploit parallelism by using multifunction pipelines; i.e., their central processing units (CPU) consist of multiple functional units. Multiple functional units are processing units that can perform more than one function (operation) simultaneously. Each functional unit is equipped with a set of registers to store the

[1] A state is the data stored in the memory of the computer at a given point in time.

data resulting from computation on that functional unit. The multiple functional units also share a set of general purpose registers that communicate with main memory. Results from one functional unit can be used as input to other functional units via a high speed internal bus. This brings us to the concept of *pipelined computation.* In pipelined parallelism, a complex operation **O** is broken up into a sequence of simpler and faster operations $o_1, o_2, \ldots, o_n$. Different operations can execute on different data units simultaneously; for instance, the output from o_1 can be automatically piped into the functional unit executing o_2, while new data is input to o_1. A number of existing computers exploit pipelined parallelism in their processing units, for example, CRAY, CDC-7600 and IBM 360/91.

Studies on supercomputer development over the past years have indicated that there is only a marginal increase in the sequential speed of supercomputers. The earliest Cray machines operated at 160 megaflops (floating point operations per second), while the more recent Cray X-MP operates at a peak performance of 210 megaflops. Current supercomputers have multiple processors (up to 4) in addition to multifunctional units, but do not provide an order of magnitude change in performance.

The *workstation* industry has benefited heavily by the introduction of RISC (Reduced Instruction Set Computer) processors which simplify the design of processors by transferring some of the work to software. For example, the original VAX uniprocessor exhibits a performance of the order of 1 MIP (million instructions per second) for typical uniprocessor RISC-based workstations. Today, this figure has increased to 20–25 MIPS. Typically, RISC processors also exploit pipelined parallelism.

2.2 *Parallel architectures*

Parallel architectures are classified based on the manner in which they exploit concurrency into *data parallel machines* and *control parallel machines. Data parallelism* allows the same operations to be independently performed on each element of a large aggregate of data. By contrast, *control parallelism* allows multiple threads of execution. Control parallelism is more general: implicit in control parallelism is the fact that each thread of execution may involve distinct data.

2.2a *Data parallel models*: Machines based on the data parallel model are frequently known as SIMD (single instruction multiple data) machines (see figure 2).

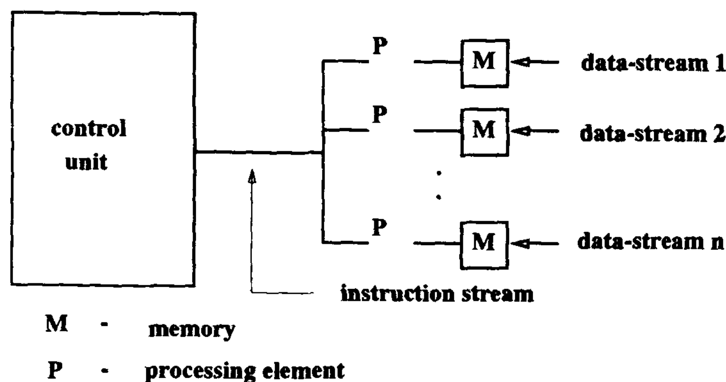
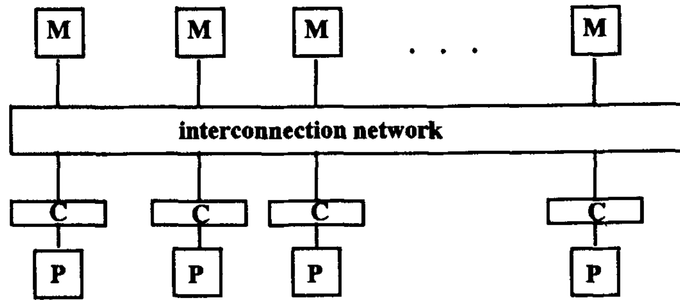


Figure 2. The data parallel model.



M - **memory**

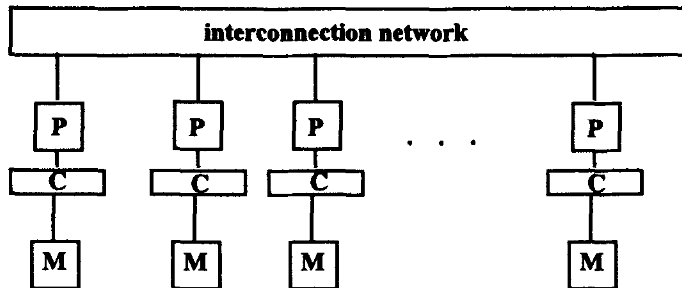
P - **processing element**

C - **local cache**

Figure 3. The shared memory model.

Each instruction stream operates on multiple units of data such as a vector or array instead of on a single operand. A single control unit coordinates the operation of the multiple processors. Synchronous computers using global clocks are quite special purpose and rather restrictive in their model of concurrent computation. This approach is exemplified by the connection machine (Hillis 1985) and conventional array processors.

2.2b *Control parallel models*: Control parallel computers can be divided into two broad classes: *shared memory* machines and *message-passing* concurrent computers (also called *multicomputers*) (figures 3 and 4). Shared memory computers have multiple processors and share a global memory. For efficiency reasons, each processor also has a local cache.[2] Separate caches create the problem of maintaining consistency



M - **memory**

P - **processing element**

C - **local cache**

Figure 4. The message passing model.

[2] A cache is a fast, expensive memory unit between the processor and the main memory and is used to hold frequently accessed data.

between caches when processors may modify shared data in their local caches simultaneously. The shared memory computers which have been built typically consist of 16 to 32 processors. Large numbers of processors create increased contention for access to the global memory. The contention for shared information increases as the computing resources increase. As a result, shared memory architectures are not scalable (Dally 1986).

Multicomputers have evolved out of work done by Charles Seitz and his group at Caltech (Athas & Seitz 1988). Configurations of multicomputers with only 64 computers exhibit performance comparable to conventional supercomputers. Multicomputers use a large number of small programmable computers (processors with their own memory) which are connected by a message-passing network.

Depending on the amount of memory per component, multicomputers may be divided into two classes, namely, *medium-grained multicomputers* and *fine-grained multicomputers*. Two generations of medium-grained multicomputers have been built. A typical first generation machine (also called the cube or the hypercube because of its communication network topology) consisted of 64 nodes and delivered 64 MIPS. Its communication latency was in the order of milliseconds.[3] On the other hand, a typical second generation medium-grained multicomputer has 256 nodes and is projected to carry out 2·5 K MIPS and has a message latency in the order of tens of microseconds. Third generation machines are currently being built and are expected to increase the overall computational power by two orders of magnitude and reduce message latency to a fraction of a microsecond (Athas & Seitz 1988).

The frontiers of multicomputer research are occupied by work on fine-grained multicomputers. Two projects building experimental fine-grained multicomputers are the J-Machine project by William Dally's group at MIT (Dally & Wills 1989, pp. 19–33) and the Mosaic project by Charles Seitz's group at Caltech (Athas & Seitz 1988). A number of other innovative architectures such as dataflow, reduction machines and logic programming engines have been proposed (Arvind & Culler 1986; Shapiro 1987).

Concurrent computing involves *partitioning* and *communication* of tasks. Partitioning involves the splitting up of a process into many threads of computation which may execute in cooperation. Threads belonging to the same process interact with each other and the underlying system must provide for communication between them. It is also necessary that the threads executing in parallel synchronize or cooperate with each other without conflicts. Thus at a higher level *coordination* is required.

Partitioning may be *explicit* where the user divides the tasks into a few large threads or heavyweight objects which may execute in parallel. This is referred to as *coarse grained partitioning*. In this approach, however, concurrency inherent in each of the tasks is not fully exploited. The need for a high degree of parallelism leads to *fine grained partitioning*. Typically, partitioning a task into the grained threads is performed by the compiler. Here, the task is split into many small threads or lightweight objects (fine grained parallelism). Very large scale integration (VLSI) is an elegant medium for expressing this form of parallel computation. The computational

[3] Communication latency is a measure of the time delay involved in transmitted information from source to destination. As communication latency increases, more time is required to transmit information from one processing unit to another. As a result, the performance of the system drops.

quanta obtained by partitioning a task may need to communicate with each other and this communication introduces additional overhead. The presence of smaller threads introduces more communication which may offset the benefits of parallelism.

Communication is needed to transfer information from one processor to another. Communication strategies may be synchronous or asynchronous.

- *Synchronous communication*: Both the sender and receiver must be available before communication begins. A real life example of synchronous communication is a telephone that waits for users at both ends to be available at the same time before conversation can occur.
- *Asynchronous communication*: Here, the sender and receiver can operate independently, the sender can send information without waiting for the receiver to be ready to accept the information. Communication that takes place through a postal system is a real-world example of asynchronous communication. Any communication between two people or two points goes through some post office. Note that it is not necessary to have a centralized controller in such a system. The presence of buffers at the receiver allows us to store the messages arriving from possibly different senders. This kind of communication is called *buffered asynchronous communication.*

When there are two or more processes executing in parallel, data dependencies dictate the threads which need to cooperate or *synchronize*. This cooperation or information transfer is achieved by means of efficient synchronization primitives implemented in hardware.

With increased parallelism, we believe that the current wave of technologies and mechanisms will cause a shift in programming paradigms. In particular, from a programming language standpoint, we are observing a shift from textual to visual (graphic) programs (Miriyala 1991). From an implementation standpoint, there is a shift from static to dynamic resource management (Venkatasubramanian 1991).

3. Universal models for scalable concurrent systems

3.1 *Inadequacy of existing software strategies*

Computers available in the market today are not sufficiently powerful to compete with the pressing demand for computational speed and power. Hardware advances have been so rapid that today it is possible to think about packing massive computational power into a few inches of silicon. The advent of VLSI technologies has also brought forth highly concurrent hardware. This alone is not sufficient to satisfy our requirements. The computational capacities of these hardware modules can be exploited only through efficient software methods to program these machines. The challenge, therefore, lies in the fact that software technologies have not scaled up in proportion to potential hardware advances. Radical software techniques have to be developed to concurrently program thousands, and in the future, millions of processors to work in concert.

Most familiar programming languages are based on the von Neumann model of computation. A major drawback of von Neumann languages (languages based on the von Neumann model of computation) is that the architectural model of the machine is the basis of the programming model of the language. Thus the programmer

is subjected to a serious constraint in forcing his/her programs to reflect the underlying architecture when one should be worrying about the most effective way to specify the problem without imposing any artificial constraints. Only then can he/she fully harness the computational power of the underlying hardware.

Another handicap of traditional languages is their difficulty in combining existing program modules in a number of ways to achieve different functionalities.[4] Furthermore, programs written in these languages are designed for deterministic computations and traditional programming strategies break down when we have to deal with nondeterminism.

Scalability, in the context of parallel computation, implies that given a program with sufficient parallelism, it will be possible to increase the performance of the system by adding physical resources. In a scalable system, no alteration of the application program is necessary to exploit the benefit of the added resources. Scalability is a fundamental requirement of *open systems*, i.e. of information systems which permit continuous influx of input from new and different sources.

Realistic modelling of natural phenomena and real world systems as open systems is possible because an open system has the following attributes:

(1) decentralization;
(2) inherent concurrency;
(3) organizational cooperation.

One can view a parallel computation model as a set of abstractions that capture the semantics and functionality of concurrent program execution. This problem has been approached from different perspectives and various models of concurrency have been proposed by language theorists, complexity analysis and architects of parallel machines. We analyse some of these models with a view towards determining their suitability as a general purpose parallel programming model for scalable systems.

There are a number of closely related approaches to developing universal models of concurrency. A conservative language strategy to defining a model of concurrency is to start with sequential processes and add communication primitives. One language-based model that does this is Communicating Sequential Processes (CSP) developed by Tony Hoare and others (Hoare 1978). In CSP, communication is *synchronous*, i.e., a source process cannot send a communication until a destination process is ready to accept it, and the process topology is *statically* determined, i.e., the process configuration cannot be altered during program execution. The significant limitations of such a model include the need to specify all concurrency explicitly, the need to predetermine resources to be used, and the need to fix synchronization points.

We focus on the three universal models – a programming model, a complexity model, and an architectural model. A universal model facilitates the development of various applications without any concern for the underlying architectural configuration. If a universal model can be used, it insulates hardware and software developments from each other: it is possible to use the advances in hardware (software) without having to be overly concerned about equivalent advances in software (hardware). Interestingly, the proposed universal models have similar underlying features. We will discuss the following classes of universal models:

---

[4]This property is called *composability*.

A programming model: *Actors* is a model of concurrent object oriented programming with active agents developed primarily by programming language theorists.
A complexity model: The *bulk synchronous parallel* (BSP) model proposed by Valiant (1990, pp. 103–111) is an intermediate model to bridge the gap between concurrent programming languages and parallel architectures. It deals with complexity issues involved in *efficient universality* and is a useful model for designers of parallel algorithms.
An architectural model: The *parallel machine interface* (PMI) developed by Bill Dally and his group at MIT (Dally & Wills 1989, pp. 19–33) is an implementation oriented machine model. The PMI aims at achieving efficient hardware implementations of primitive abstractions.

It is possible to measure the cost of implementing different operations in various programming models on the basis of the primitive operations on each of these universal models. We discuss them in greater detail below.

3.2 *The actor model*

The *actor model*, first proposed by Hewitt (1977) and later developed by Agha (1986) captures the essence of concurrent computation in distributed systems at an abstract level. In the actor paradigm the universe contains computational agents called *actors*, which are distributed in time and space. Each actor has a conceptual location (its *mail address*) and a *behavior* as illustrated in figure 5.

The only way one actor can influence the actions of another actor is a send the latter a communication. An actor can send another actor (or itself) a communication only if it knows the mail address of the recipient. On receiving a communication, an actor processes the message and as a result may cause one or more of the following events:

- creation of a new actor,
- change its behavior, and,
- send a message to an existing actor.

In asynchronous communication, the sender and receiver need not coordinate message delivery. If two messages sent to an actor arrive at their destination simultaneously, there must be some mechanism to serialize the incoming communications and execute both messages. To ensure this, every actor is equipped with a mailbox that queues any communications received. Therefore, communication in the actor model is *asynchronous* and *buffered*. The size of the buffers is theoretically unbounded.

Although actors is inherently an asynchronous model, it is possible to simulate synchronous models as specializations of the actor model. A important characteristic of communication in the actor model is the ability to communicate mail addresses. Thus the interconnection topology of the system is capable of changing continuously. This adds to the reconfigurability and flexibility of the actor model; for example, it allows resource management decisions such as object to processor mapping to be directly programmed. Another property of the actor model is the *guarantee of delivery*. i.e., messages in the system will eventually reach their destination actor. This implies that mail in transit cannot be indefinitely buffered.

Actor execution is graphically expressed in terms of event-diagrams (see figure 6). An event-diagram is a pictorial representation of the arrival order of events within a thread of execution and the causal relationship between different threads of

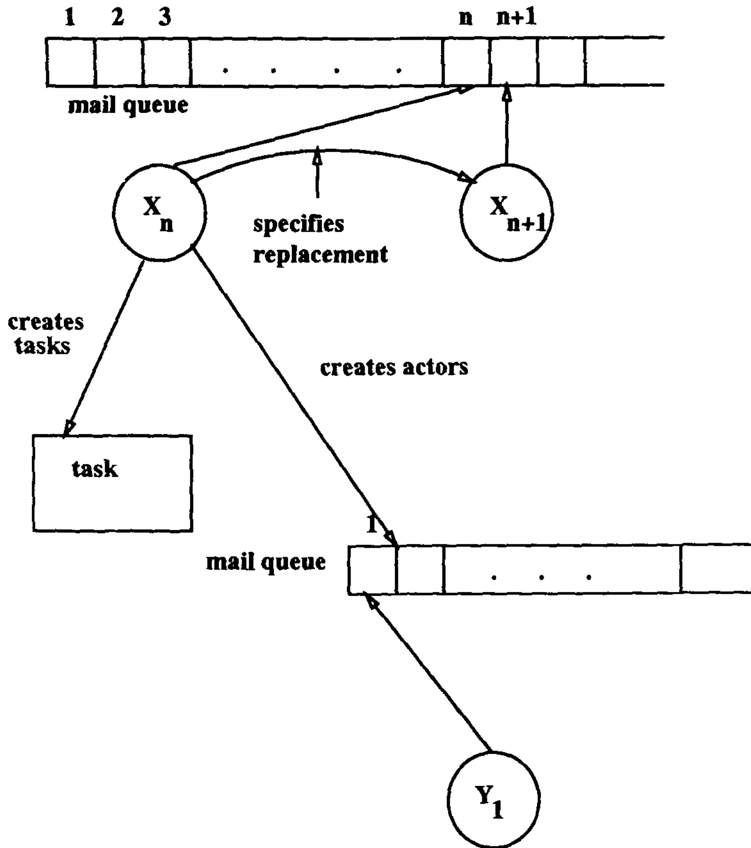


Figure 5. An abstract representation of actor transitions: When an actor processes the nth communication, it determines the replacement behavior which will process the $(n+1)$th communication. The mail address of the actor remains unchanged. The actor may also send communications to specific target actors and create new actors.

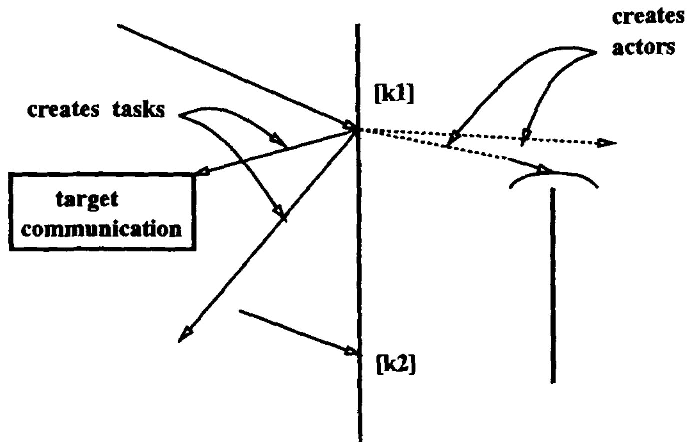


Figure 6. Each vertical line represents the linear arrival order of communications sent to an actor. In response to processing the communications, new actors are created and different actors may be sent communications which arrive at their target after an arbitrary delay.

computation. The thick vertical line in figure 6 represents the actor on a linear temporal scale with time floating from the top of the line to the bottom. Events that occurred earlier lie above those that occur later. Event diagrams bring out the concept of local time and local state in the actor model. Causality connections form the fundamental synchronization mechanism in the actor model.

The actor model is well-suited for fine grained computation because of its dynamicity and its ability to create new actors inexpensively. It may sometimes to necessary to sequentialize task execution using one of the following methods:

(1) *Introduce causality constraints* between different threads of execution. This does not reduce the grainsize of a task, but causes sequentiality that allows controlled resource allocation.
(2) *Execute a given thread of control sequentially* instead of in a functional fashion. Either user-defined or compiler-derived annotations could be used to specify the sequentiality. For example, we could have both sequential and parallel versions of a recursive method. The parallel version creates new tasks for every recursive call. The sequential version is called when the argument to the recursive call falls below a certain value. The sequential method definition executes sequentially to completion without creating new tasks.

Baude & Vidal-Naquet (1991, pp. 184–195) have shown that the asynchronous, message passing actor model is as powerful as the traditional PRAM or Parallel Random Access Memory model (used to analyse the complexity of parallel algorithms) which is a synchronous, shared memory model.

In the PRAM model, program execution is deterministic and can utilize an unbounded number of logical processing units. The logical processing units communicate via a global shared memory. Efficient parallel programs are defined as those programs that demonstrate an exponential speedup with a polynomial increase in the number of processing units. Such *efficiently parallelizable programs* are classified under the NC class of programs.

Efficiently actor parallelizable problems or NC_{actor} are defined as those problems for which there exists an actor program whose time complexity (the length of the execution chain created by the actor) is a polylogarithmic function of the input size and whose size complexity (a measure of the space needed to process the message) is a polynomial function of the input size. Baude & Vidal-Naquet (1991, pp. 184–195) provide a simulation of actors by PRAM and vice-versa. In particular, they show:

$$NC_{\mathrm{PRAM}} \subseteq NC_{actor}$$

In summary, the actor model is a convenient perspective for the programmer of the parallel machine due to its flexibility and simplicity (programming with abstractions). However, it does not model some architectural characteristics – such as the overheads involved in message passing.

3.3 *The bulk synchronous parallel model*

Valiant (1990, pp. 103–111) proposed the bulk synchronous parallel (BSP) model as a suitable bridge between parallel languages and architectures. The BSP model consists of three units:

- components – which are computing or memory units;
- router –responsible for point to point message passing;

- periodic synchronization – that performs synchronization of all or a subset of the processors with a periodicity L.

To achieve *efficient universality* results, we must be able to model the performance of the system as a relation. If h is the number of messages sent or received in a computation on a given processor and g is the sum of all computations in the system per second divided by the number of data words delivered per second, it is assumed that all communications are delivered in time gh. Let the startup latency or cost be s. The router sends and is sent at most h messages in a superstep. Such supersteps are also called *h-relations*. Therefore, the cost to realize an h-relation is $gh + s$. This technique of modelling the performance of a system as a relation offers parameterized controllability and is one of the basic features of the BSP model that makes it amenable to complexity analysis.

Memory components in the BSP model are distributed across the processing components and therefore access to every computational unit is equally likely. To allow efficient symbolic to physical address mapping, the BSP model adopts a pseudo-random mapping or hashing mechanism. The assumption is that known hash functions are used and they can be computed locally and inexpensively.

Synchronization in the BSP model is carried out periodically using *bulk synchronization*. The process of bulk synchronization, from which the model derives its name, is as follows. Initially, the mean delay to execute each operation is roughly estimated. After the estimated time has elapsed, an elected processor sends out a sync detect signal to all the other processes along a spanning tree. If synchronization is achieved, we can proceed with the next phase of computation. Otherwise, a new timeslot is allocated and the process is repeated till synchronization is achieved.

The tasks generated are executed in a sequential fashion but all the tasks generated can execute in parallel. The tasks involved in a bulk synchronization must wait until the synchronization has been achieved for further continuation of the executing task. However, it is possible for processing components to switch this mechanism off and execute without waiting for synchronization. As a consequence, task granularity in the BSP model is controllable by varying the periodicity of synchronization, L. As L increases, the granularity of the program also increases. The type of algorithms most naturally expressed using the BSP model are PRAM programs.

The BSP is a possible model for the designer of a parallel algorithm. Valiant (1990, pp. 103–111) demonstrates how the BSP model can be embedded on theoretical models like PRAM as well as architectural models like networks of systems. However, the level of abstraction in BSP makes it difficult to specify optimizations that may be necessary for efficiency – in particular, no specific language interface or architectural issues can be addressed. Furthermore, the model limits communication costs in an architecture by assuming the feasibility of efficient bulk synchronization. Thus BSP would need to be modified before it could be used as a model of a realistic scalable system.

3.4 *The parallel machine interface*

In Dally & Wills (1989, pp. 19–33), a universal machine model has been proposed to support various parallel models of computation, the *Parallel Machine Interface* (PMI). Any machine model, sequential or parallel, must have abstractions representing hardware components, i.e., memory, instructions and instruction sequencing. One such general-purpose abstraction in a sequential machine that has been very successful

is the notion of stack-based storage allocation. Complex modelling primitives lack flexibility and generality, and are less amenable to optimizations and therefore, it is important to keep the modelling primitives simple and straightforward.

What are the kinds of abstractions needed to support a parallel model of computation? Three basic requirements of any parallel model are *communication, synchronization* and *naming*. Most proposed models of parallel computation like dataflow (Arvind & Culler 1986), static message passing, shared memory, parallel logic programming (Goto *et al* 1988, pp. 208–229) and concurrent object oriented programming improvise on similar implementations of these primitive abstractions. The goal is therefore to design efficient hardware implementations of these primitive mechanisms that are portable across different programming systems.

A parallel machine interface isolates the issues of programming models from the details of machine organization and implementation. This abstract model of parallel systems is then embellished with special features in order to support a particular programming paradigm efficiently on a specific target architecture.

A parallel architecture interface, called Pi has been proposed based on the PMI (Wills 1990) and the implementation of several machine models has been illustrated. An abstract machine architecture has also been proposed for the same and we will use this abstract architecture implied by Pi in the discussion below.

The PMI views storage as a collection of data units or *segments* which are logically related. It makes no assumptions about how segment names are interpreted in an implementation. However, some translation mechanism must be built to support segment addressing and access. The cost of accessing a segment is directly dependent on the translation scheme chosen.

The PMI currently assumes a message passing model of communication, but mechanisms like communication via shared memory or RPC (remote procedure calls) can also be implemented. The communication network does not take a stand on routing and buffering mechanisms. As in the actor model, the PMI assumes that all messages are eventually delivered (albeit with an arbitrary delay due to network latency) and that there is no message order preservation.

The PMI models the three primitive operations in a parallel system as follows:

(1) *Communication* is represented by means of a message send. Other forms of communication like shared memory reads and writes are more complex mechanisms implemented in terms of message sends.
(2) *Synchronization* is of two kinds: *data synchronization*, which is needed when there is a data dependence between executing tasks, and *control synchronization*, which requires that a task must complete before another can begin execution. The mechanism used to implement safety and correctness in synchronization is *actor firing* and progress after synchronization is guaranteed by *I-structure accesses* (Arvind *et al* 1987).
(3) *Naming* issues in most models can be implemented as some form of translation from a logical name to a physical address in the memory of the system. The model assumes that the translation is embedded within the message injection and reception mechanisms.

In addition, the Pi model defines primitives to provide information regarding the relative proximity of objects to a particular node.

The abstract machine implied by the Pi model consists of a set of nodes interconnected by means of a communication network. Although the Pi model is fine-grained (where grain size is measured in terms of the node size), it should be observed that fine-grained systems are *upward-compatible* with larger grained systems.

Table 2. Universal models of parallel computation.

	BSP	Actor	PMI
Storage model	Memory and components	Actors as Abstract Data Type (ADT)	Logical collection of data units called *segments*
Communication	Asynchronous point to point	Asynchronous point to point	Asynchronous point to point
Synchronization	Bulk synchronization	Causality and history-sensitivity	Access attributes
Naming	Pseudo-random mapping of symbolic names to physical addresses	Unique actor names	Nametable for address translation
Process/object Topology	Unclear	Dynamic creation Dynamic configuration	Dynamic creation Dynamic configuration
Expressing Parallelism	Sequential tasks Specified	Parallelism default Implicit synchronization	Fine-grained threads Explicit synchronization
Locality	Hash function distributes memory components	Actors are a unit of locality	Primitives to support spatial locality

In other words, architectures for the fine-grained computation can support larger grain size efficiently (but not necessarily vice-versa). Using the Pi model, it is possible to measure the cost of implementing different operations in various programming models in terms of the primitive operations on the PMI. The translation between PMI and actors is one-to-one. It is worth noting that costs measured in terms of the PMI model apparently closely correspond to actual machine costs.

3.5 *A comparison of universal models*

Table 2 gives a summary of the three different universal models we have considered. As is evident from all the models, some attributes of parallel computation are mandatory to any model irrespective of whether it has an analytical or architectural flavour, i.e., memory, communication and synchronization. However, the degree of specification with respect to other attributes depends largely on the level of abstraction that the model is trying to achieve.

3.6 *A hierarchical view of parallel computation*

Good scalability characteristics are achieved only if the problem size scales up with the number of processors. In other words, merely increasing the number of processors keeping the problem size constant is expensive. Furthermore, scalability and complexity analysis on these models must also consider the implications of the algorithm and its mapping on the underlying architecture (see, for example, Singh *et al* 1991). A hierarchical view of a parallel machine is shown in figure 7.

The purpose of the hierarchical model is to provide layers of abstraction in a parallel system. When dealing with high level issues like algorithm design or programming language issues, we would like to abstract away from implementation

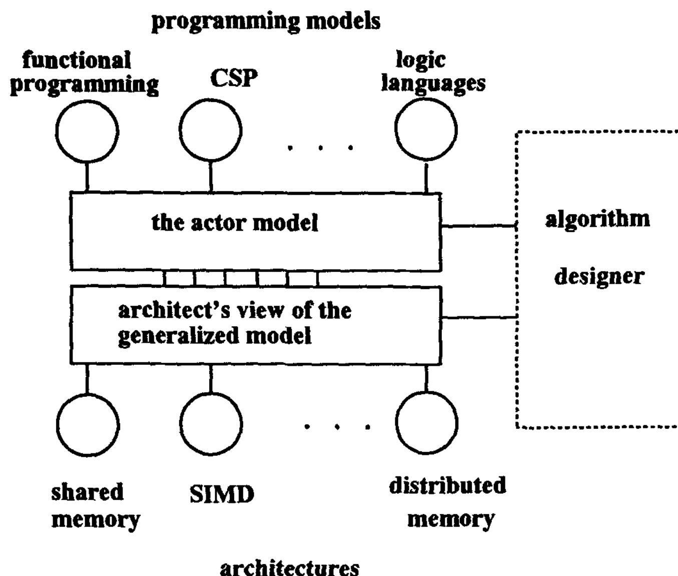


Figure 7. A hierarchical view of parallel systems: Existing models do not consider the implications of algorithm design in the scalability and complexity analysis.

or architectural details like the communication subsystem or resource management. However, in order to discuss resource management issues without referring to any particular programming language or architecture, we must isolate features specific to parallelism from issues specific to the framework in which parallelism is being exploited. The hierarchical view in figure 7 illustrates the different levels at which a parallel system can be viewed for different purposes (Venkatasubramanian 1991).

We now consider parallel resource management with the actor model as the programmer's view of the world and the parallel machine interface as the underlying model of a highly concurrent machine.

4. Resource management for scalable systems

In this section, we focus on the actual implementation of scalable parallel programs on a highly parallel machine. In particular we elaborate on techniques for the effective utilization of hardware.

Resources refer to the hardware and software components of a computer system which are required in order to solve a specific problem. *Resource management* involves techniques and mechanisms used to efficiently allocate, utilize and coordinate these resources. Parallelism creates new complexities for resource management, for example, the communication network is one of the most critical of execution resources. In fact, a bottleneck is scaling concurrent computers is not limitations in the computational power of individual processors, but the costs and delays associated with transferring information from one processor to another. Thus resource management strategies must try to reduce the communication traffic.

Another source of complexity in parallel systems is that different resources may be needed simultaneously and these resources must *all* be available. Furthermore, in a large scale concurrent system, dynamic allocation of resources is necessary because

computations may need to be automatically divided into large numbers of sub-computations. In actor systems, such division happens each time concurrent sub-requests are made as a result of processing a given request (Agha & Hewitt 1987). With every fork in the computation, execution resources must be provided to the sub-computations created. Often the behavior of sub-computations cannot be determined in advance. Different strategies for subdividing resources lead to different results. An improper allocation may lead to a condition where a useful subcomputation cannot proceed due to the non-availability of resources, called the *dangling sub-computation problem.*

Dynamically spawning off tasks may give rise to expanding resource requirements limited only by their physical availability. By serializing executions, resources can be exclusively allocated to the executing process and there are no resource contentions. However, little parallelism is exploited by this scheduling strategy. The other extreme aims at extracting all the parallelism inherent in the application by using a suitable programming paradigm. In such cases, studies have determined that many programs are "embarassingly parallel" (Sargeant 1986). Excessive parallelism leads to inordinate resource utilization and may have an even more serious outcome – deadlock due to unavailability of resources for any of the parallel tasks to continue execution. There is an obvious crossover point between the degree of parallelism exploited and effective resource utilization in a concurrent system. The goal, then, is to be able to set up a flexible *throttle* which will allow us to increase or reduce the degree of parallelism at will (Arvind & Culler 1986; Sargeant 1986).

Another aspect of resource management is controlling the use of resources at the application level. Some problems exhibit an exponential growth in the number of possible paths which could lead to a solution; it is infeasible to explore all of them. This in turn means that differing amounts of resources must be allocated to different paths. The assessment of how fruitful a particular path is changes over time as one assesses the intermediate results along the path. This is one reason resources need to be controlled and re-allocated dynamically.

Other complications in a real implementation arise from the presence of prioritized execution of processes, conflict resolution, deadlock handling and synchronization. Effective resource management in parallel machines is a combination of:

Static analysis: The text of the program is analysed before execution at compile-time to detect infomation such as useless concurrency via dependence analysis, rough estimates of resource requirements, data placement and partitioning.

Heuristics: It is often sufficient to approach the resource management problem conservatively. In fact, mechanisms implemented to guarantee completeness or totality may degrade machine performance, thereby defeating the purpose of an efficient resource management strategy. Heuristics designed to achieve effective resource management concentrate on detecting sections of program execution critical to performance and target their restructuring efforts at those sections where the payoffs would be significantly large.

Reconfiguration strategies: The system must be capable of dynamically detecting "congestion points" and handling them, when appropriate, by alteration of the existing system configuration.

Resource management issues can be divided into actor/process management, memory management and I/O (input–output) management.

4.1 *Actor management*

Actor placement and migration is governed by mechanisms implemented to handle locality and load balancing.

4.1a *Locality*: A process in one processing unit may need to communicate with a process or object on another processing unit. For example, in figure 8, processes P1 and P3 on different processing units may need to communicate with each other. Similarly, processes P2 and P4 also interact with each other. The allocation of processes as illustrated in figure 8 reduces processor utilization caused by tasks waiting for the result of a communication. Frequent communication between processes on different processors also increases network contentions. Minimizing network traffic is important even in the presence of high speed networks, where the network bandwidth is the major limiting constraint on communication. Locality directed scheduling policies, as in figure 9, reduce interprocessor communication, thereby decreasing network traffic.

Memory locality is a property of the pattern of a program's references to memory. References which are tightly grouped together in terms of addresses are said to have spatial locality. References close together in time are said to have temporal locality. Three distinct classes of locality can be identified in user programs:

(1) temporary objects that are usually intermediates during computation;
(2) fairly long lived data structures whose lifetime is a significant portion of the program's lifetime;
(3) permanent structures such as the runtime system's routines and structures.

The above hierarchy of relative object lifetimes is well portrayed and exploited in the generational garbage collection schemes discussed later.

4.1b *Scheduling*: The scheduling problem in multiprocessor machines is that of distributing multiple threads of control to execute on the available processing resources. Threads which share a lot of data and threads which communicate frequently might yield better throughput if scheduled to execute on the same processor. But this means that the execution of these tasks is serialized, inhibiting possible parallelism.

A straightforward policy is to statically schedule tasks to execute on specific processors despite the fact that better choices could be made dynamically after the execution scenario is more well defined. An alternative is to perform dynamic

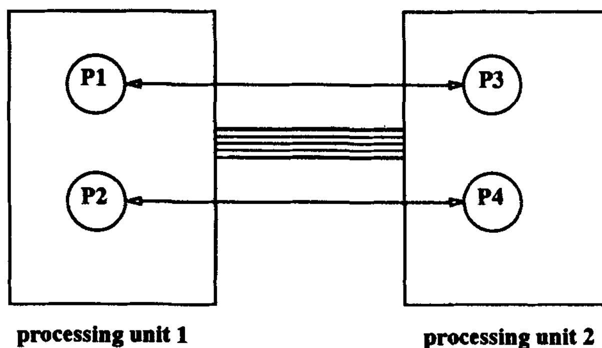


Figure 8. A system that does not exhibit locality. Communication which occurs across processors blocks the network which is a critical resource.

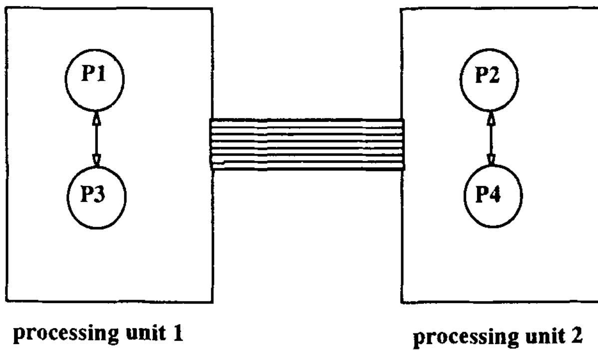


Figure 9. A system that exploits locality. Locality based scheduling strategies avoid unnecessary communication overhead.

scheduling by having a global pool of tasks which need to be executed, from which processors can pick their next to execute. However, a centralized job queue may disrupt the locality inherent in the application.

4.1c *Load balancing*: Load balancing is the task of keeping the processors of a parallel machine uniformly busy. Figures 10 and 11 illustrate this concept.

For effective load balancing, each processor needs to know the degree of load in every other processor, or at least, some controller needs to maintain load information in order to make load balancing decisions. In the latter case, there is likely to be a contention for this centralized resource (i.e. the controller). If there is a static interconnection scheme between processors in a system, each processor need only maintain nearest neighbor information.

Locality and load-balancing are contradictory constraints that need to be weighed against each other for determining an optimal tradeoff between dispersion and aggregation (Athas 1987). Locality aims at placing related objects in close physical proximity, thereby mitigating the frequent communication costs. Load balancing efforts are geared toward splitting and distributing work and do not encourage "clubbing" of computational nodes or threads. In order to make effective use of resources, we must exploit the right amount of locality needed to mask the communication latency. The degree of object mobility required to achieve this balance and mechanisms to deal with this have been explored in Jul (1989).

4.1d *Network activity*: An important concern of network behavior in a parallel system is that the links of the network should be kept uniformly busy with data

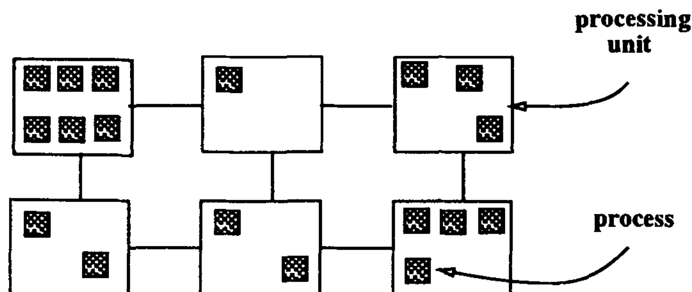


Figure 10. Processors before load balancing. Note that the throughput of the system is limited by its slowest component. Processes within a processing unit execute sequentially and the heavily loaded processor becomes a bottleneck.

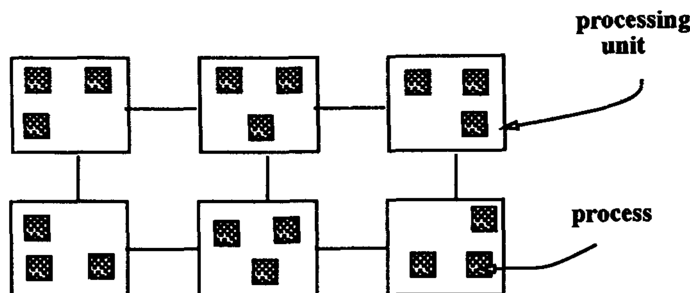


Figure 11. Processors after load balancing. The uniform distribution of computational units improves the system throughput.

objects. Non-uniform distribution of objects may result in bottlenecks at frequently accessed nodes, thereby delaying other communications. Often a process will create a stream of information used by another process. Such processes are called *producer* and *consumer* processes respectively. An important factor for proper utilization of the network is that message dispatch and reception rates of a producer and its consumer of the communication be compatible. Consider the scenario where we have a producer–consumer relationship between objects. In order to exploit concurrency, we assign them to different nodes. If these nodes are adjacent to each other in a mesh, we have a single link between them and this may result in non-uniform traffic in the network. To avoid this problem, we can place the two objects on nodes which are far apart. However, note that message reception at the consumer is serialized and placing the nodes far apart in the network would block the network which is a shared resource in the system. Thus there is a contradictory argument that presses for locality. If the producing and consuming rates are well-matched, we could exploit concurrency without excessive traffic on the network.

4.1e *A distributed namespace*: We assume that communication in a distributed memory system can be modelled as a series of message sends and that messages are routed by means of a fast, efficient routing network to their destinations. Local communication is accomplished through primitive messages[5]. We advocate having a uniform address space across all nodes in a network. All entities in the system are referred to by a virtual name that is uniform over the entire system, giving us a uniform namespace over the entire system. A uniform namespace brings us additional flexibility in resource handling: it permits us to migrate objects to support actor placement, migration and garbage collection. The cost of this flexibility is the overhead associated with name translation (virtual name to physical address). Each access to the object must query the translation table and it is therefore vital that all translations be completed with a relatively low latency. Obviously, we should avoid virtual name to physical address translation for objects which reside locally and as far as possible for remote objects through intelligent compilation techniques and optimizations.

4.2 *Memory management*

Storage management is used to allocate space for an object when it is created and subsequently reuse this allocation space when it is no longer needed. Memory which is not accessible is referred to as *garbage* and the reclamation of garbage is *garbage*

[5] Primitive messages are messages that a processor sends to itself.

collection. An efficient storage management scheme plays a crucial role in enhancing the efficiency of a programming system. The two major issues involved are storage allocation and reclamation. Automatic storage management schemes require that the runtime system be capable of recognizing a shortage of memory and reclaiming unused memory for reallocation. The actor-based storage model adopted here assumes the following operating systems support services:

(1) a mechanism that can allocate and deallocate contiguous blocks of memory;
(2) abstractions needed to access objects in memory with a single virtual address across a distributed global object namespace;
(3) effective support for relocation of objects either within the heap or across the network (object migration).

4.2a *Garbage collection*: There have been a number of schemes and algorithms for performing garbage collection (Ungar 1984, pp. 157–167; Baker & Hewitt 1987). The following steps are either implicitly or explicitly performed in all the algorithms.

(1) Identification of accessible objects;
(2) reclamation of the inaccessible memory objects;
(3) compaction of memory to improve locality.

Some of the traditional garbage collection algorithms take time proportional to the size of memory which makes them inefficient as the size of memory increases. The additional time taken to detect and reclaim all the unreachable memory cells may also destroy the interactive response of the system. Storage reclamation involves both temporal and spatial overhead. In addition, stringent time restrictions imposed by interactive programs necessitates efficient memory management mechanisms.

General requirements of a garbage collection algorithm:

(1) good interactive response;
(2) capability to reclaim circular structures;
(3) requires minimal hardware support;
(4) meshes well with virtual memory;
(5) causes only a small impact on execution overhead.

With the help of operating system services, a number of low-level memory management details can be abstracted thus enabling us to design.a generalized garbage collection algorithm. We will now briefly describe the traditional GC mechanisms and extend them to parallel machines.

In *Reference Counting* schemes, every cell (object) in memory is associated with a field called the *reference count* which is a count of the number of references to the cell. The reference count of a cell is continuously updated as pointers to the cell are created or destroyed. When the reference count of a cell becomes 0, the object is no longer referenced and can be collected. Reference counting mechanisms are incremental in nature. Therefore, computation is not interrupted for a significant period of time. A major flaw, however, is their inability to handle circular garbage (because the reference count of any cell in a circular structure can never be zero). Also, when an object is collected, additional work is done in decrementing the reference counts of any child cells. This process can be potentially unbounded. There have been attempts to modify reference counting algorithms to accommodate various implementation issues (Deutsch & Bobrow 1976; Bevan 1987, pp. 273–288; Watson & Watson 1987, pp. 432–443; Goldberg 1989, pp. 313–320; Ichisugi & Yonezawa 1990).

The *Mark and Sweep* algorithm detects garbage by halting the mutator (application program) and then performing two (optionally three) phases. A predetermined root set is used to *mark* all the reachable objects in the first phase, the second phase reclaims dead objects one at a time by walking down the entire memory space and a third, optional phase compacts memory to avoid fragmentation. This algorithm collects all available garbage and uses only one extra bit per word for reclamation purposes. However, the running time of mark and sweep is proportional to the size of memory, and hence this method is impractical for large memory sizes.

The *Stop-and-Copy* garbage collector divides the available memory in a system into two regions – the *from-space* and the *to-space*. Only one of these regions, i.e., the from-space, is available to the mutator at any time. The copying algorithm (Fenichel & Yochelson 1969; Cheney 1970) traverses all the reachable records in the from-space starting at a root pointer set and copies these records into a separate area of memory, the to-space, that is currently unused by the mutator. Forwarding pointers are placed in the from-space to redirect any references to the swapped record. At the end of collection, the only records in to-space are the reachable ones. The from- and to-space pointers are swapped and the mutator now works on the new from-space. Note that the amount of work performed is proportional to the number of reachable cells whereas the traditional mark-and-sweep algorithms take time proportional to the size of memory. The performance of the copying algorithm is unaffected as the size of memory increases.

Generational Garbage Collection (Liebermann & Hewitt 1983) is an extension of the copying garbage collector that takes advantage of two significant observations:

(1) high rate of infant mortality – A young object is more likely to die than an object that has survived for a while;
(2) newer records are more likely to point to older records than vice-versa. An older record points to a newer record only if it is altered (reassigned) after it is initialized. This flavour of reassignment is rather unlikely in many languages. Generational garbage collection is claimed to be highly suitable for such languages.

Records with a similar age are grouped together in a contiguous area of memory. Once we have segregated objects on the basis of their age, reclamation efforts can be concentrated on the youngest generation. The drawback of this scheme is what is known in garbage collection literature as the *Tenuring Problem.* Old objects that die may take a long time to be reclaimed. Even worse, they may never be reclaimed until the occurrence of an offline collection. If objects are promoted too quickly, there are more objects present in higher generations. Collecting higher generations results in significant pauses and clever promotion policies are necessary to mask this unavoidable tradeoff.

4.2b *Multiprocessor* GC *algorithms*: The desire to exploit concurrency among tasks in a job complicates storage management. This is due to the need for synchronization between the various tasks in order to ensure their correctness and consistency. Programming environments for highly parallel machines, it appears, are largely centred around dynamic resource management schemes which require sophisticated memory allocation and reclamation schemes. Furthermore, one can think of storage reclamation itself as a concurrent set of processes.

All this concurrency requires coordination between the parallel subtasks of a single job which in turn requires efficient and reliable communication. Deficiencies of

traditional algorithms in a parallel setting have been detailed upon in Hudak & Keller (1982, pp. 168–178). Many of the insufficiencies arise from the fact that the traditional algorithms halt the computation process, preventing realtime response by the system. In other words, the collector (thread of control that is responsible for garbage collection) is initiated after the mutator(s) (threads of control corresponding to the application program) have been halted.

The main overhead in parallel GC occurs in maintaining intraprocessor and interprocessor references to an object. If this is not done, it is possible for an object to be collected even while a reference to it exists, violating the correctness criteria of the algorithm.

- *GC in shared memory machines*

One of the earliest parallel mark-and-sweep algorithms on a shared memory has been discussed in Steele (1975). A dedicated storage reclamation processor handles garbage collection of a memory space that it shares with a mutator. In the general case, this algorithm could be extended to handle any number of mutators that share a single address space. Hence, this algorithm is applicable to a tightly coupled system. The correctness proof for a similar mark-and-sweep based algorithm was developed in Dijkstra *et al* (1978).

Locking mechanisms must be implemented to prevent two processors from simultaneously manipulating the same memory location. Extensions of the stop-and-copy algorithm and generational GC implemented on shared memory machines are presented (Appel *et al* 1988, pp. 11–20; Pallas & Ungar 1988, pp. 268–277).

- *GC in distributed memory machines*

In distributed architectures, each processing unit is associated with its own local memory. A global communication network connects the different processing units together. As intra-processor communication is orders of magnitude faster than interprocessor communication, we are forced to exploit locality and manipulate data in the local memory in a distributed memory machine.

There are two levels of garbage collection on distributed memory machines, i.e., local and global garbage collection. Global garbage collection is a systemwide GC involving all processors in the system. Local garbage collection is carried out locally on each processor without inhibiting the processing activities on another processor. There are costs associated with both local and global GC. The interval of a global garbage collection is dictated by the first processing unit that needs memory. Other processors have to cooperate even if they have sufficient local memory to continue work. The overhead of synchronization – orchestrating a global start and stop could be substantial, especially when the collection process has to be real time. In other words, the deviations in the time taken by different processing elements to run out of space must be more well-balanced. Hardware techniques like logic and signal lines or software mechanisms like barrier synchronization can be used to reduce the synchronization overhead. Local garbage collection requires a reference management table to determine whether an object is wholly local or not. The entries in this table should be periodically updated and eventually reclaimed.

The authors' group is currently studying algorithms for distributed memory management and a hierarchical approach to memory management has been presented in Venkatasubramanian (1991).

4.3 *I/O management*

While advances in processor architecture and memory/register management strategies have enhanced performance in parallel machines, commensurate improvements in I/O performance have not been made. The disparity between the performance of the I/O subsystem and the other subsystems in current parallel architectures must be reduced. This problem is acute in the data parallel model explained earlier, where the physical dimensions of the machine pose a limitation on the amount of information which can enter or leave the surfaces of the machine.

In a three-dimensional physical world with n processors along each dimension, a cube has a total of n^3 processors with 6 faces. As all the input/output to a system configured as a cube occurs through the face of the cube, the n^3 processors communicate with the outside world through the $6n^2$ processors on the face of the cube. It must be noted that this is the hypothetical best case for a cube configured system. This yields an I/O limitation of $6n^{2/3}$ for loading information from external sources to the processors in a cube simultaneously from all the $6n^2$ entry points into the cube. To illustrate the above discussion, consider a cube with a million processors. Thus an order of 17 basic time units would be needed to load one unit of information into each of the processors in a million processor multicomputer with optimal I/O.

It is possible to avoid the I/O bottleneck for specific applications (Kung 1986, pp. 49–54), but the bottomline remains that we must focus efforts on improving the I/O subsystem performance to the level that it does not affect throughput. Real-time visualization and animation are examples of I/O limited applications.

Interoperability in programming environments, i.e., the ability to switch from one programming framework to another, will play an important role in future distributed systems. Multiparadigm programming environments are software technologies that will permit a smooth transition among programming paradigms.

5. Multiparadigm programming environments

In order to address the problems of concurrent computation, a number of important programming paradigms have been developed. Our research focuses on providing a flexible basis for providing efficient execution of important programming constructs inspired by different linguistic paradigms in a single framework.

5.1 *Declarative programming*

In *declarative programming*, there is no notion of instruction sequencing – we "declare" what is to be computed rather than how this computation must be done. Separating logic from control is an important step toward achieving abstraction and modularity. All control or computational issues governing machine behavior are relegated to the language implementation. Two approaches to declarative programming are functional and logic programming.

Declarative programming is a more radical approach to concurrent computing. Unlike the CSP model where the concurrency and state transformations are explicit, declarative models are implicitly parallel. An important feature of declarative languages is that variables in these languages correspond to values; they have no notion of a computational history or state. The absence of state resolves a number

of determinacy issues, for example, there are no cache consistency problems. However, the ability to create shared, modifiable data structures cannot be cleanly integrated into declarative models. State must be perpetually passed through procedural abstractions such as functions or relations. This is inadequate for programming in the large system which requires support for data abstraction.

Functional and concurrent logic programming suggest important linguistic constructs and programming techniques. Important language features in these paradigms, such as higher-order functions and pattern-matching, can be defined in terms of actor primitives (Agha 1989, pp. 1–19) and used as needed.

5.2 *Concurrent object oriented programming*

Object Oriented Programming (OOP) is another programming paradigm whose roots go as far back as Simula (a simulation language), with significant contributions made by developments in languages such as Smalltalk and Flavors (Wegner 1990). Modern object oriented programming has been influenced by a number of individuals and concepts which makes it difficult to give a universally acceptable definition of object oriented programming. A simplified definition is given in Madsen (1987):

A program execution is regarded as a physical model, simulating a behavior of either a real or imaginary part of the world.

Here, the universe is viewed as being composed of many passive entities with functions which transform them. Structuring mechanisms in OOP include classification and specialization. Classification enables us to group diverse objects based on some common characteristics. Specialization supports the specification of objects as modifications of existing specifications. Abstraction and inheritance are language mechanisms which allow this structuring. Object-oriented programming provides encapsulation: the user sees a simplified interface consisting of a set of operations permissible on a structure.

5.2a *Abstraction*: Abstraction is a powerful tool which allows programmers to manage complexity by dealing with high level concepts before dealing with their representation. Object oriented languages are designed to support both data and procedural abstraction. *Procedural abstraction*, common to all modern programming languages, allows different actions to be grouped into a single name. *Data abstraction* is a data structuring and packaging mechanism in which small subsystems which have certain common characteristics are grouped together to form a larger subsystem.

In other words, an object is a unit of encapsulation which hides from the user, implementation details (i.e., the representation) of the data structure as well as the operations performed on it. The user need only be concerned about what operations may be performed on the object. An advantage of object-oriented programming is the easy maintenance, modifiability and portability of the code. Actors are similar to objects in that they encapsulate a local state and a set of operations. However, actors differ from objects in several ways. Perhaps the most important distinction is that actors are inherently concurrent – many actors may be active at the same time. By contrast, in sequential object oriented languages, only one object may be active at a time.

5.2b *Inheritance*: Inheritance is a mechanism which facilates code sharing and reusability. Code is structured in terms of superclasses and subclasses. Inheritance

mechanisms allow a class to borrow the functionality (methods) of its superclass and specialize it. Code sharing can lead to name conficts – the same identifier may be bound to procedures in an object and in its class. Object oriented languages differ in how such name conflicts are handled. One proposal, advanced by Jagannathan and Agha (see, for example, Agha & Jagannathan 1991), is to allow programmers to define different possible inheritance mechanisms in a single linguistic framework by providing the ability to define and manipulate the underlying name bindings.

The power of Concurrent Object Oriented Programming (COOP) comes from the ability to develop abstractions which hide the details and complexities of concurrency. COOP systems are larger systems of communicating actors and they can be built in terms of actor primitives. The inherent locality in distributed systems is modelled naturally through a modular design. A judicious choice of abstractions and primitives which express concurrency in a manner transparent to the user is of crucial importance. It has been observed that the main contribution of OOP to concurrency is to provide reusable abstractions which can hide the low level details of partitioning, synchronization and communication from the user (Lim & Johnson 1989). Thus, COOP systems illustrate the power of the actors as building blocks. Another approach to COOP can be found in ABCL (An Actor Based Concurrent Language) (Yonezawa 1990).

5.3 *Reflection*

In the normal course of execution of a program, a number of objects are implicit. In particular, the interpreter or compiler being used to evaluate the code for an object, the text of the code, the environment in which the bindings of identifiers in an object are evaluated, and the communication network are all implicit. As one moves from a higher level language to its implementation language, a number of objects are given concrete representations and can be explicity manipulated at the lower implementation level (for a more detailed discussion see Agha 1990, 1991, pp. 1–59).

The dilemma is that if a very low level language is used, the advantages of abstraction provided in a high-level notation are lost. Alternately, the flexibility of a low-level language may be lost in a high-level language. A *reflective architecture* addresses this problem by allowing us to program in a high-level language without losing the possibility of representing and manipulating the objects that are normally implicit (Maes 1987). *Reification* operators can be used to represent at the level of the application, objects which are in the underlying architecture. These objects can then be manipulated like any other objects at the "higher" application level. *Reflective* operators may then be used to install the modified objects into the underlying architecture. Reflection thus provides a causal connection between the operations performed on this representation and the corresponding objects in the underlying architecture.

In a COOP system, the evaluator of an object is called its meta-object. Reflective architectures in COOP have been used to implement a number of interesting applications. For example, Watanabe and Yonezawa (see Yonezawa 1991) have used it to separate the logic of an algorithm from its scheduling for the purposes of a simulation: in order to build a virtual time simulation, messages are time-stamped by the meta-object and sent to the meta-object of the target which uses the time-stamp to schedule the processing of a message or to decide if a rollback to a previous state is required. Thus the code for an individual object need only contain the logic of the simulation, not the mechanisms used to carry out the simulation; the specification of the mechanisms is separated into the meta-objects.

In summary, reflection provides a way of integrating various language paradigms. By using the underlying representations, one can define programming constructs and their interrelation as first-class objects. Thus a multiparadigm programming environment may be created for heterogeneous computing. Components of a system may use language constructs which are more suited to the computations they are carrying out.

6. Discussion

There are a number of ongoing efforts to achieve the fifth generation computing objective. The US High Performance Computing and Communication (HPCC) Initiative (1991–1996), funded at a level of over 4 billion dollars, attempts to address fundamental problems in high performance computing. Major emphasis is placed on the development of hardware and software technologies required for scalable, parallel computing systems with a performance of trillions of operations per second on a wide range of important applications. The European ESPRIT project is placing emphasis on models of computing based on the human brain and parallel architectures.

Similar efforts are being conducted in Japan in the New Information Processing Technology funded by the Japanese Ministry of International Trade and Industry. Areas of research include establishing sound theoretical foundations for flexible information processing technologies using distributed and cooperating agents or actors, development of massively parallel computing technologies (a million processor architecture) and new application domains for such technologies.

The US HPCC program has identified a large number of problems critical to science and engineering, the so-called *grand challenge* problems. Addressing these grand challenges will require orders of magnitude increase in computational power. For example, in biomedical investigations, research on genes causing cancer requires management of data of the order of billions of molecular units. The fastest supercomputers available in the market today will require hundreds of years of processing time to yield the necessary information. Similarly, fundamental research in physics and chemistry, and superconductor research require accurate simulations to predict the transformations of materials under varying conditions. Weather prediction and information about severe changes in atmospheric conditions are other computationally intensive problems.

Scalable concurrent computing is a fundamental enabling technology required to deliver the promise of high performance computing. We are just beginning to address some of the problems involved in using massively parallel processing.

The authors would like to acknowledge Chris Houck, Subrata Mitra and Rajendra Panwar for their comments on the paper. The work described in this paper has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Development Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90–07195).

References

Agha G 1986 *Actors: A model of concurrent computation in distributed systems* (Cambridge, MA: MIT Press)

Agha G 1989 Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE'89) Lecture Notes in Computer Science. Vol. 366* (Berlin: Springer-Verlag) pp. 1–19

Agha G 1990 Concurrent object oriented programming. *Commun. ACM* 33 (9): 125–141

Agha G 1991 The structure and semantics of actor languages. In *Foundation of object-oriented languages. Lecture Notes in Computer Science. Vol. 489* (Berlin: Springer-Verlag) pp. 1–59

Agha G, Hewitt C 1987 Actors: A conceptual foundation for concurrent object-oriented programming. In *Research directions in object oriented programming* (Cambridge, MA: MIT Press)

Agha G, Jagannathan S 1991 Reflection in concurrent systems: A model of concurrent continuations, Technical Report, University of Illinois at Urbana Champaign

Appel A, Ellis J, Li K 1988 Real-time concurrent collection on stock multiprocessors. In: *SIGPLAN'88 Conference on Programming Language Design and Implementation* Atlanta, GA

Arvind, Culler D E 1986 Dataflow architectures. In *Annu. Rev. Comput. Sci.* 1: 225–253

Arvind, Nikhil R, Pingali K 1987 I-structures: Data structures for parallel computing, Technical Report Computation Structures Group Memo 269, Massachusetts Institute of Technology, Cambridge, MA

Athas W 1987 *Fine grain concurrent computations*, Ph D dissertation, Computer Science Department, California Institute of Technology (also published as technical report 5242:TR:87)

Athas W, Seitz C 1988 Multicomputers: Message-passing concurrent computers. *IEEE Comput.* pp. 9–23

Baker H, Hewitt C 1977 The incremental garbage collection of processes, Technical Report Memo AL-454, Massachusetts Institute of Technology, MIT Artificial Intelligence Laboratory

Baude F, Vidal-Naquet G 1991 Actors as a parallel programming model. In *Proceedings of the 8th Symp. on Theoretical Aspects of Computer Sciences: Lecture Notes in Computer Science. Vol. 480* (Berlin: Springer-Verlag) pp. 184–195

Bevan D I 1987 Distributed garbage collection using reference counting. In *Parallel architecture and languages Europe: Lecture Notes in Computer Science. Vol. 259* (Berlin: Springer-Verlag) pp. 273–288

Cheney C J 1970 A nonrecursivelist compacting algorithm. *Commun. ACM* 13: 677–678

Dally W J 1986 *A VLSI architecture for concurrent data structures* (Kluwer: Academic Press)

Dally W J, Wills S 1989 Universal mechanisms for concurrency. In *Parallel architecture and language Europe: Lecture Notes in Computer Science. Vol. 365* (Eindhoven: Springer-Verlag) pp. 19–33

Deutsch P, Bobrow D G 1976 An efficient, incremental, automatic garbage collector. *Commun. ACM* 19: 522–526

Dijkstra E W, Lamport L, Martin A J, Scholten C S, Steffens E F M 1978 On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21: 966–975

Fenichel R R, Yochelson J C 1969 A lisp garbage collector for virtual memory systems. *Commun. ACM* 12: 611–612

Goldberg B 1989 Generational reference counting: A reduced-communication distributed storage reclamation scheme (1989). In *SIGPLAN'89 Conference on Programming Language Design and Implementation* (Portland, OR: ACM Press)

Goto A, Sato M, Nakajima K, Taki K, Matsumoto A 1988 Overview of the parallel inference machine architecture. In *Fifth generation computing systems* (Tokyo: ICOT)

Hewitt C 1977 Viewing control structures as patterns of passing messages. *J. Artif. Intell.* 8: 323–364

Hillis D 1985 *The connection machine* (Cambridge, MA: MIT Press)

Hoare C A R 1978 Communicating sequential processes. *Commun. ACM* 21: 666–677

Hudak P, Keller R M 1982 Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on LISP and Functional Programming* (Portland, OR: ACM Press)

Hwang K, Briggs F 1984 *Computer architecture and parallel processing* (New York: McGraw Hill)

Ichisugi Y, Yonezawa A 1990 Distributed garbage collection using group reference counting, Technical report, University of Tokyo, Dept. of Information Science

Jul E 1989 *Object mobility in a distributed object-oriented system*, Ph D thesis, University of Washington

Kung H T 1986 Memory requirements for balanced computer architectures. In *Proc. of the 13th Annual Symposium on Computer Architecture* (New York: IEEE Press)

Lieberman H, Hewitt C 1983 A real-time garbage collector based on the lifetimes of objects *Commun. ACM* 26: 419–429

Lim J, Johnson R 1989 The heart of object oriented concurrent programming. *Sigplan Notices* 24(4): 165–167

Madsen O L 1987 *Block-structure and object oriented languages* (Cambridge, MA: MIT Press)

Maes P 1987 *Computational reflection*, Ph D thesis, Vrije University, Brussels, Belgium (Technical Report 87-2)

Miriyala S 1991 *Visual representation of actors using predicate transition nets*, Master's thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Urbana, IL

Pallas J, Ungar D 1988 Multiprocessor smalltalk: A case study of a multiprocessor-based programming environment. In *SIGPLAN Conference on Programming Language Design and Implementation*

Sargeant J 1986 Load balancing, locality and parallelism control in fine-grained parallel machines. Technical Report UMCS-86-11-5, Dept. of Computer Science, University of Manchester

Shapiro E 1987 *Concurrent prolog: Collected papers, Series in Logic Programming* (Cambridge, MA: MIT Press)

Singh V, Kumar V, Agha G, Tomlinson C 1991 Scalability of parallel sorting on mesh multicomputers. In *Proceedings of the International Parallel Processing Symposium*

Steele G L 1975 Multiprocessing compactifying garbage collection. *Commun. ACM* 18: 495–508

Ungar D M 1984 Generation scavenging – a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA

Valiant G 1990 A bridging model for parallel computation. *Comput. ACM* 33: 103–111

Venkatasubramanian N 1991 *Hierarchical memory management for parallel machines*, Masters's thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Urbana, Il (forthcoming)

Watson P, Watson I 1987 An efficient garbage collection scheme for parallel computer architectures. In *Parallel Architectures and Languages Europe, Lecture Notes in Computer Science. Vol. 259* (Berlin: Springer-Verlag) pp. 432–443

Wegner P 1990 Concepts and paradigms of object-oriented programming. In *OOPS Messenger* 1(1): 7–87

Wills D S 1990 *Pi: A parallel architecture interface for multi-model execution*, Ph D thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts

Yonezawa A 1990 *ABCL: An object-oriented concurrent system* (Cambridge, MA: MIT Press)