International Symposium on New Information Processing Technologies '91,
Tokyo, Japan, March 13 – 14, 1991

PROGRAMMING MODELS FOR MASSIVELY PARALLEL SYSTEMS

W. K. Giloi and W. Schroeder-Preikschat

GMD Research Center for Innovative Computer Systems and Technology
at the Technical University of Berlin
Hardenbergplatz 2, D-1000 Berlin 12, Germany, e-mail: giloi@gmdtub.uucp

Abstract

New, very important application paradigms call for the development of massively parallel computer systems. During the nineties highly integrated processing nodes will become feasible having a performance range of 102 to 103 MIPS and MFLOPS, respectively. The entire system will be capable of processing several hundred thousand to several million bits with each clock cycle. These systems are by nature distributed memory systems in which the nodes communicate through message passing. Recently, new kinds of hierarchical interconnection structures have evolved that combine economy of realization with the high connectivity required for massively parallel systems. In order to provide for the necessary scalability of massively parallel systems, new programming models must be developed that allow programming to take place at a sufficiently high level of abstraction as needed to make such a system programmable and convenient to use. Moreover, the level of abstraction must allow application programs to be written independently of the actual size of the system. This requires the view of either virtual shared memory or virtual processors, as well as new concepts for extremely efficient distributed operating systems, highly optimizing compilers, and adequate support by the hardware. These issues are addressed in this paper. While for the virtual shared memory paradigm efficient solutions are already evolving, the virtual processor approach is entirely new. Some first implementation concepts are presented in the paper.

1 INTRODUCTION

Massively parallel systems are computers with a very large number of processors. What is "very large", however, is strongly dictated by the technology available. So far, Thinking Machine's CM-3 with 64K single-bit processors was considered by many as the epitome of "massively parallel." However, the next generation of largely parallel MIMD architectures with up to 2096 nodes [1] will be more parallel than the CM-3, since each processor processes 64 bits in parallel.

A Connection Machine with up to 256K processors has been announced. Will that be the new definition of "massively parallel" ? The New Information Processing Technologies Study envisions systems with millions of processors. With the up-coming 100-million-transistor technology (64-Mbyte chip) this seems perfectly feasible. Does "massively parallel" therefore begin with a million processors ?

One important, yet unanswered question is whether massively parallel systems are meant to introduce a new programming style or whether they are just a means to raise the peak performance limits of supercomputers by some orders of magnitude while adhering to a conventional programming style. In either case the success of such an endeavor will strongly depend on the solution to the question how to program efficiently a system with many thousands or even millions of processors ? All these are pertinent questions that as of yet have hardly been addressed. This paper tries to give some answers.

Whatever the programming paradigm is, the user should not be aware of the physical processors, let alone have to program them, as this very well might prove to be an unmanageable problem. Does the user have to see individual processors at all and, if yes, what for ? If the individual processors are not visible, wouldn't it be more efficient in terms of the communication and management overhead to have, say, 16K of 64-bit processors rather than one million single-bit processors ? Where are the applications that demand bit-serial processors ? The first massively parallel computer, the Connection Machine became successful in the marketplace only after it was converted into a 64-bit number cruncher. In order not to mix apples and oranges, it may be better to speak of the millions of bits processed with each clock cycle and leave it as a hardware detail how many bits wide the processors are made.

In this paper we will consider two scenarios. In the first case the computer hardware is massively parallel, while the programming model is of the conventional shared memory kind. Since it is impossible to efficiently realize a massively parallel architecture as a shared memory system, there exist only *virtually shared memory*. In the second case, we introduce the *massively parallel system* as a programming model rather than a hardware feature. In this case the processors of the massively parallel architecture are *virtual processors*, and the user can define any arbitrary number of them. This will be very suitable in programming paradigms where the problem consists of a very large number of objects, each of which is to be subjected to a primitive transformation. In this case, each object may have individually assigned one of the virtual processors, to carry out the transformation.

For both cases we shall show how such systems can be implemented. Both solutions, virtualization of memory or virtualization of processors, introduces additional overhead, which is the price to pay for the manageability of the system. Our goal is to minimize that overhead. Therefore, new concepts for extremely efficient distributed operating systems, highly optimizing compilers, and adequate support by the hardware had to be developed. While for the virtual shared memory paradigm efficient solutions are already evolving, the virtual processor approach is entirely new. Some first implementation ideas are presented in the paper.

2 VIRTUALLY SHARED MEMORY

Largely parallel computer architectures are by nature *distributed memory architectures*. In such a system each *processing node* has access only to its local memory, while communication among the nodes takes place through *message passing*. This leads to the programming model of *cooperating parallel processes*. The processes communicate on the basis of an appropriate inter process communication protocol. Data objects are encapsulated into the process that owns them.

If the algorithmic structures and solution domains are very regular, the communication and synchronization conditions of the parallel program can be satisfied by preprogrammed communication constructs loaded from a library [1]. Eventually, compilers may even become smart enough to generate the communication constructs and put them into the right places [2].

If the algorithmic structures and solution domains are highly irregular, these methods are prone to fail. Consequently, it is left to the programmer to explicitly perform the program partitioning into co-operating processes, the workload distribution over the nodes, and the communication and synchronization through the insertion of *send* and *receive* constructs. In massively parallel systems this is not practically feasible [3].

These problems are largely avoided in shared memory systems. In this case, a conventional programming style can be applied as reflected by such commonly used programming languages as Fortran, Lisp, Ada, or C. Data objects exist in a global address space; thus, different threads of control can readily share them. Of course, this still requires *data access synchronization* provided by some locking mechanism, e.g., semaphores, to ensure the correct order of data access.

The global address space can be established in a distributed memory environment by mechanisms that provide the application software with the view of a *virtually shared memory* [4]. Such a scheme introduces additional overhead. To keep the overhead within tolerable limits, a very efficient mechanization is mandated, based on specific architectural support by the system. Such a mechanism will be outlined below.

The main challenge of a *virtually shared memory architecture* (VSMA) is to ensure the coherence of the distributed data. Fortunately, data consistency need not be enforced at all points in time. Copies of the shared data entities that are distributed over the system may be written at their owner's discretion as long as nobody else reads them, and copies may be read in various nodes as long as nobody else writes them. Therefore, data consistency need be enforced only at certain synchronization points. This scheme is called *weak coherence*. Weak coherence means that there are critical regions -- we call them *weak blocks* -- within which multiple writes to copies of the same data object are permitted. On leaving the block, the system must merge the diverging copies into one consistent object, so that strong coherence is reestablished without changing the program semantics. Weak consistency can be mechanized efficiently in a distributed environment [5], thus enhancing the efficiency of the VSMA significantly.

While most data entities in a VSMA may be owned by only one of the *thread of controls* (TOC), there exist also a number of shared entities. In this case the sharing TOCs must know where in the system they are residing, and access rights must be issued for every such entity. Performing this at the granularity level of single memory words would be inefficient. Therefore, it has been proposed to do the sharing on the basis of the pages of the demand-paging node memories [4]. This scheme is therefore called *virtually shared memory*.

The *virtually shared memory* solution offers the advantage that its management can be supported by the local virtual memory managers of the nodes [4]. Copies of an entity that must be destroyed according to the *write-invalidate* method can simply be tagged *invalid* in their page descriptor. Any attempt to access an invalid page leads to a page fault in the same manner as if the page were not existent in the local memory, thus triggering the fetch of the missing page.

In principle, the *virtually shared memory* approach offers the unique advantage that data migrate dynamically to the site where they are needed. However, it depends very much on the initial distribution of services and data over the system how long it will take until the *local working sets* will have built-up in that manner. The better the original workload distribution, the lesser will be the message traffic needed to have the data objects migrate to the nodes where they are required.

This adds an additional dimension to the complexity of the compiler. In addition to the common compiler optimizations, the compiler should automatically perform some a priori coarse grain parallelization and load distribution such that the run-time migration of shared data entities between the nodes is minimized, not because the VSMA principle would require it but for better efficiency. In connection with the MANNA project, we are working on the implementation of our distributed *virtually shared memory* scheme [5], including the operating system [6], the compiler, and the supporting hardware [7],

3 AN IMPLEMENTATION OF STRONG AND WEAK COHERENCE

Our implementation of *virtually shared memory* [5] is based on a capability mechanism. A *capability* is a short message of fixed size containing: the page identifier, the access right specification (read/write), the owner identifier, and a pointer to the copy set list. In the strong coherence scheme each page has exactly one owner-TOC: only the owner TOC may have the write capability. The owner has a copy set list, listing all the TOCs that have a read capability to that page. The granting of access capabilities is governed by the following two rules.

Write capability granting rule:
A write capability to a shared page can be held only by one TOC at a time; i.e., there is only one owner. On occurrence of a write fault in a TOC, that TOC will request the write capability from the current owner. As long as the owner is in a critical region, it ignores the request, else it must honor it. Before the new owner can exercise the write capability granted, it must invalidate all other copies of that page.

Read capability granting rule:
A read capability may be simultaneously requested from the owner by any number of TOCs. As long as the current owner is in a critical region, it ignores the request, else it must honor it. Before granting the read capability, the current owner must change its own capability from *write* to *read*. Subsequently, it sends the requested read capability together with a copy of the page to the requestor.

For efficiency reasons, this strong coherence scheme is supplemented by a weak coherence scheme. *Weak coherence* means that inside a *weak block* several TOCs may have the right to write into the same shared page *but not into the same location* (we call this a *multiple-write page*). Thus, inside a weak block a shared page may have many owners (each one owning a write capability to its specific copy). On leaving the weak block, the differing copies of the same page must be merged into a new unique entity that will reflect all the changes made by the various writers. Thus, the strong coherence is reestablished.

Weak coherence is mechanized efficiently in the following manner. Let $P_o = \sum p_{io}$ be the initial page. Let $P = \sum p_i$ be the value of the page after a multiple write (in different locations p_i). The new value P can be expressed by the identity

$$P = P - P_o + P_o = [\sum_i (p_i - p_{io})] + P_o$$

That is, multiple writers form the differences $(p_i - p_{io},)$ between the elements of p_{io} of the original page and the modified elements p_i. Each such difference can be viewed as an *update mask*. The updated page P is obtained by adding all the update masks to the original page P_o. Since different TOCs must not write into the same location, the different update masks are mutually exclusive.

This scheme allows the unification of all the changes to be performed in a distributed, pipelined fashion, i.e., without any need for a global system manager. It also allows weak and strong data coherence to coexist in the VSMA.

However, the VSMA approach inevitably leads to a large amount of traffic in the system. Critical regions must be locked and unlocked, write and read capabilities and invalidation notices must be exchanged between the nodes, and copies of data objects must be sent to the node where they are referenced. Therefore, the most effective hardware support fc r the VSMA is a very fast message-passing mechanism for short, fixed size messages, including broadcasting capabilities. To minimize the communication start-up time, each node must be equipped with a dedicated communication processor [8].

4 THE VIRTUAL PROCESSOR MODEL

As was pointed out above, by *massively parallel computing* we understand a programming model where the user can assign an individual processor to every basic computational activity in the program. For example, if the user simulates a neural network, every neuron in the network -- no matter how many -- will have assigned its individual, dedicated processor to compute the firing behavior of that neuron. In a lattice gauge simulation each molecule has its 'own' processor to compute its movement and its energy, etc. If one has a billion molecules, one therefore needs a billion processors.

Synchronization of and communication among the cooperating processors is greatly facilitates by two major simplifications:

(1) Synchronization is performed in a *lock-step* fashion. That is, there are alternating computation and communication phases. In the computation phase all computational steps of an iteration are performed by all processors. In the subsequent communication phase the results of the computation step are communicated throughout the system. The lock step-step mechanisms ensures that the receiving processors are ready to receive a message when the communication phase is entered. Therefore, the overhead of a rendezvous protocol can be avoided.

(2) Communication is not secured. This assumes that the applications for massively parallel systems are such that sporadically occurring message faults do not affect the result significantly. This is certainly the case in all Monte Carlo simulations and in neural networks, but also in numerical applications where a faulty message causes not more than a local perturbation that "dies off" after a few steps.

The virtual processor model in connection with the lock-step synchronization is appropriate in all cases that allow for a fine-grain partitioning of the solution algorithm into a very large number of primitive operations that all have approximately the same execution time. In the realm of Monte Carlo simulations, large neural networks, etc. such a behavior is evident. However, the model can be applied also in "classical' numerical applications such as PDE solvers, by assigning to each grid point a dedicated processor.

This model is maintained regardless of the physical size of the machine, i.e., the actual number of processors. Consequently, the processors of the programming model are *virtual processors* that are dedicated by the compiler to the computational tasks of the program and mapped by the operating system onto the set of available physical processors. That is, a virtual processor is in reality a thread of control (a process) that is executed by one of the physical processors. In order to make this scheme as efficient as possible, we introduce in the following the notion of *featherweight processes*.

5 FEATHERWEIGHT PROCESSES

5.1 What Are Featherweight Processes ?

Unlike *heavyweight processes*, *lightweight processes* do not constitute a domain of protection but just some means of structuring a program [9]. Protected address spaces are usually provided only for entire groups of lightweight processes. This makes the context switches among lightweight processes of the same group much faster, for they all have the same environment.

In order to implement our virtual processor model efficiently, we need a mechanism that is yet simpler and more efficient than lightweight processes. Therefore, we introduce the notion of *featherweight processes*. The foundation for the implementation of featherweight processes is provided by the PEACE family of operating system kernels [6].

Featherweight processes are the mechanization of the *virtual processor programming model*. They provide no protection for process execution and inter process communication, leaving the physical separation of the node memories as the only means of protection in the system. Naturally, this does not protect the system against faults in inter process communication. Therefore, featherweight processes are suitable only in *a single-user, single-task mode of operation* that can tolerate sporadic communication faults. The thus reduced functionality of featherweight processes reduces their run-time overhead to a bare minimum.

5.2 Semantics of Inter Process Communication

Since no protected communication domains for the featherweight processes are provided, the communication among them is totally unrestricted. That is, each process can communicate with every other process. Unlike conventional systems, where it is the task of the operating system kernel to provide such domains of protection, the correctness of the program execution is ensured by the execution model of a single-user, single-task operation in a lock-step fashion. The global lock-step synchronization scheme excludes the possibility that messages may be sent at a time when the recipient is not in the receiving state.

Hence, featherweight processes are combined with *featherweight communication*, in which data transfers are secured only by the hardware but not by the communication protocols carried out by the operating system kernel. Unsecured communication means that the sender must not first seek a rendezvous with the receiver to find out whether the latter is ready for the communication, before a message can be sent. However, even in that scenario some minimum of supervisory functions must be exercised by the operating system kernel, because it must still be guaranteed that the physical communication channel does exist; e.g., that the interconnect is not blocked and buffer space is available in the receiving node.

5.3 Featherweight Process Implementation

The implementation of featherweight processes can be based upon a *coroutine mechanism*, executed under supervision by the operating system. This reduces context switches to the simple task of saving some registers. Therefore, the operating system kernel executes a function **toc_resume***()* (toc: thread of control). The implementation of **toc_resume**() and the number of registers to be saved depends in the first place on the node processor architecture. In our implementation of the PEACE kernel, which is written in C++, one must make sure that all the registers used by C++ are saved too. Therefore, the number of registers to be saved depends also on the implementation. The overhead caused by register saving is minimized by the following mechanism.

The compiler partitions the register set of the processor into two banks: *volatile* and *non-volatile* registers. The volatile registers are used only for holding local parameters and intermediate results. Therefore, they need not be kept consistent between procedure calls (assuming an interrupt-free operation). Only the non-volatile register bank is used for storing global variables and, thus, need be exchanged on a coroutine switch.

The functionality of the operating system kernel that supports featherweight processes is simple: all that is required is the handling of TOCs and the (unprotected) communication among them. The availability of the communication hardware is checked by an appropriate driver. This operating system kernel is the simplest member of the PEACE family of operating system kernels [6]. It can be bound to the application program as a library routine.

5.4 Performance

In the existing implementation of **toc_resume**() on the MANNA system [7], which operates with the Intel i860 processor, the cost of a coroutine switch amounts to 2 instructions for the call, 1 instruction for the return, and 2n instructions for saving the non-volatile registers. Hence, a coroutine switch takes a total of

$$3 + 2n$$

cycles.

Our C or C++ compilers assign 15 integer and 6 floating-point registers to the non-volatile bank, thus requiring the saving of 21 registers. In contrast, the Fortran compiler utilizes the full set of registers for optimizations and thus declare all 32 floating-point registers of the i860 as non-volatile. Hence, the coroutine switch overhead on the next generation of RISC processors or superscalar processors will be in the order of magnitude of 1 microsecond. Assuming for such a processor a sustained performance of 30 MFLOPS, this is the equivalent of 30 numerical operations. One can assume that the primitive operation performed by the virtual processor in each computational step encompasses approximately that many instructions, which means that we will have a fairly balanced system.

The estimate above scales with technology; i.e., as the MFLOPS rate goes up, so will the MIPS rate. This reconfirms our belief that the virtual processor paradigm will become a viable, highly appropriate programming model for massively parallel computer systems.

References

[1] Rattner J.: *Grand Challenge Computing Systems*, plenary paper presented at the 24th Hawaii Internat. Conf. on System Sciences, Jan. 1991

[2] Zima H.P., Bast H.-J., Gerndt M.: *SUPERB -- A Tool for Semi-Automatic MIMD/SIMD Parallelization*, Parallel Comp. 6 (1988), 1-18

[3] Joint Workshop of INRIA and the US Office of Naval Research, Roquencourt, France, Dec. 2-5, 1990

[4] Li K.: *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D.thesis, Yale University 1986

[5] Giloi W.K. et al.: *A Capability-Based, Distributed Implementation of Virtual Shared Memory Architectures*, Proc. 2nd European Distributed Memory Conf., (1991)

[6] Schroeder-Preikschat W.: *Overcoming the Startup Time Problem in Distributed Memory Architectures*, Proc. 24th Hawaii Internat. Conf. on System Sciences (1991), 551-559

[7] Giloi W.K.: *GENESIS and MANNA: Goals, Concepts, and Achievements*, submitted for publication to the Journal for Distributed and Parallel Computing (1991)

[8] Giloi W.K., Schroeder W.: *Very High-Speed Communication in Large MIMD Supercomputers*, Proc. ICS '89, ACM order no. 415891, 313-321

[9] Cheriton D.R.: *Multi-Process Structuring and the Thoth Operating System*, Ph.D.thesis, University of Waterloo, UBC Tech. Report 79-5, 1979