Future Generation Computer Systems 10 (1994) 29–43

Architecture of parallel [illegible]

Yasuo Hidaka *, Hanpei Koike, Hidehiko Tanaka

Department of Electrical Engineering, The University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

We describe the architecture of the parallel management kernel for the parallel inference engine PIE64, focusing on how to treat load distribution and scheduling in highly parallel symbolic processing. Since the kernel manages automatic load distribution and scheduling, the task remaining for the programmer is to employ parallel algorithms with sufficient concurrency. A programmer need no longer be concerned with load distribution, load partitioning, load assignment, parallelism explosion, exhaustion of resources, or execution efficiency. We also describe an evaluation of the load distribution method.

Key words: Parallel operating system; Load distribution; Load partitioning; Scheduling; Parallel inference machine

1. Introduction

The most fundamental issues in parallel processing are said to be communication latency and synchronization cost [2]. Besides these issues, parallel management, such as load distribution and scheduling, is also crucial to highly parallel processing. Although parallel management does not cause so much overhead in large grained parallel processing, it causes inevitable overhead in fine grained parallel processing. In order to absorb such overhead effectively, we have investigated a cooperative processing model. In this model, communication, synchronization and parallel management are performed simultaneously with basic computation. As a result, we have adopted a composite architecture for the processing element of the highly parallel inference engine, PIE64 [5]. This composite architecture consists of three kinds of processors which bear responsibility for 'computation', 'communication and synchronization' and 'management'.

We will describe the parallel management kernel executed by the management processor. The parallel management kernel is a kind of operating system kernel, and performs low-level system management. In particular, we attach much importance to 'parallel management', which is special management necessary for parallel processing, as shown in Fig. 1. Higher-level functions of operating systems, such as programming environment or user management, are not included in the functions of the parallel management kernel. Those functions will be written in Fleng, the principal programming language of PIE64, and executed in the same manner as application programs.

* *Corresponding author.* Email: hidaka@mtl.t.u-tokyo.ac.jp

0167-739X/94/$07.00 © 1994 Elsevier Science B.V. All rights reserved
SSDI 0167-739X(93)E0044-Z

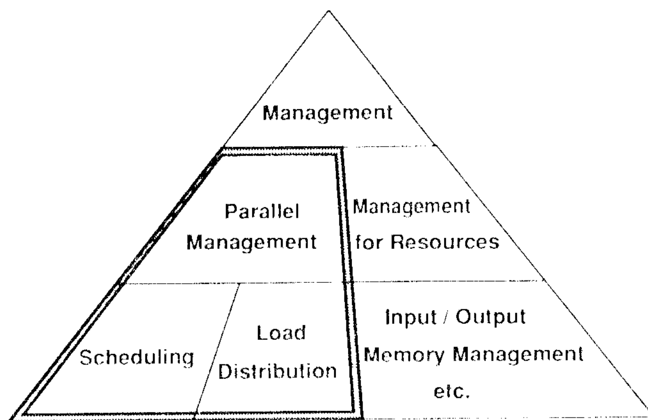


Fig. 1. Parallel management is management needed for each thread in parallel processing, like load distribution and scheduling. The cost of parallel management increases as granularity becomes finer.

First, we will discuss the details of parallel management. Parallel management is that processing needed for each thread in parallel computation, and is distinguished from other ordinary management, such as resource management. Here, a thread is the unit of parallel processing, and is supposed to be executed sequentially. We will enumerate the kinds of parallel management found in operating systems, and discuss characteristics of each kind of parallel management, focusing on requirements for load distribution and scheduling, which greatly affect performance.

Next, we will describe the architecture of the parallel management kernel of PIE64, focusing on strategies for load distribution and scheduling.

In PIE64, load distribution is handled at three stages, namely, static partitioning by the compiler, dynamic partitioning by the parallel management kernel, and dynamic assignment by the interconnection network. All the requirements for load distribution, i.e. sufficiently high parallelism, reduced communication, and balanced loads, will be attained with this three-stage strategy.

The scheduling strategy in the kernel introduces several kinds of dynamic priority. One is based on the room in heap memory, and its purpose is to release programmers from fear of resource exhaustion caused by explosive parallelism. Another kind of dynamic priority is based on run time parallelism; its purpose is to quickly increase insufficient parallelism. In addition, the scheduling strategy also introduces *respite time* in starting execution of a thread. Respite time is determined by the time to prepare required data, in order to prevent suspensions, and reduce context switching.

Finally, we will describe preliminary results of a comparison between static and dynamic partitioning. If parallelism greatly exceeds the number of PEs, simple dynamic partitioning with little overhead is more effective than sophisticated static partitioning. However, if parallelism and the number of PEs are of comparable size, dynamic partitioning becomes dramatically ineffective. We conclude that the most promising method is a composite method which provides the merits of both static and dynamic partitioning.

The problem of load distribution and scheduling has been well-studied [3]. Much work has been done in cases where the program behavior is already known. It has also been done in cases of a scientific numerical program with regular behavior. However, since highly parallel symbolic processing often has very irregular behavior with many processes, it is difficult to apply known techniques to it. A common method for highly parallel symbolic processing is that a programmer designates a strategy [10,15,16]. In such a method, it is assumed that the behavior of the program is well understood by the programmer.

Nevertheless, this assumption is not always true for highly parallel symbolic processing. Even though a programmer supposes that he has extracted moderate concurrency, the actual parallelism is often weak because of some overlooked implicit dependency. Also, in order to achieve sufficiently high parallelism, a programmer should concentrate on extracting as much concurrency as possible, and should not think of eliminating excessive concurrency, which should be done by the system.

In PIE64, load distribution and scheduling by the parallel management kernel ensures that a programmer need no longer fear explosive parallelism, or be forced to designate a strategy. Therefore, he can concentrate on increasing the maximum concurrency as determined by the algorithm.

Fleng and PIE64 are briefly described in Section 2 and 3 respectively. Parallel management is generally discussed in Section 4. The architecture of the parallel management kernel is described in Section 5, which is focused on load distribution and scheduling. The comparison between static partitioning and dynamic partitioning is shown in Section 6.

2. Fleng

Fleng [9] is a programming language for fine-grained parallel symbolic processing, and is one of the committed-choice languages, or concurrent logic programming languages. In this family of languages, there are Concurrent Prolog [11], PARLOG [4], GHC [17], etc. Among them, Fleng is probably the simplest language: there are no mode declarations, no guard goals, or no implicit AND relations among goals.

A Fleng program is a set of Horn clauses of the form:

$$\text{Head} :- \text{Body}_1, \text{Body}_2, \ldots, \text{Body}_n.$$

The left side of $:-$ is called a head, and the right side is a body.

Basic data types in Fleng are atoms (which include symbols and numbers), structural data (which include list cons cells and vectors), and logic variables. A list cons cell has a pair of data as its car and cdr. A vector has any fixed number of data as its arguments. A logic variable initially has an undefined value; the value will be determined later.

A computation process is represented by a goal. Syntactically, a goal is a predicate symbol, and any fixed number of data as its arguments.

Fig. 2 shows the pure execution model of Fleng. Execution begins when a top query goal is put in the goal pool. An arbitrary goal in the pool is chosen for execution, removed from the pool, and checked for unifiability with a clause head. If a unifiable clause head is found, the goal is committed to that clause, and the goal is reduced to the body part of the clause. The new body goals are then added to the goal pool. This unification and reduction process is repeated again and again

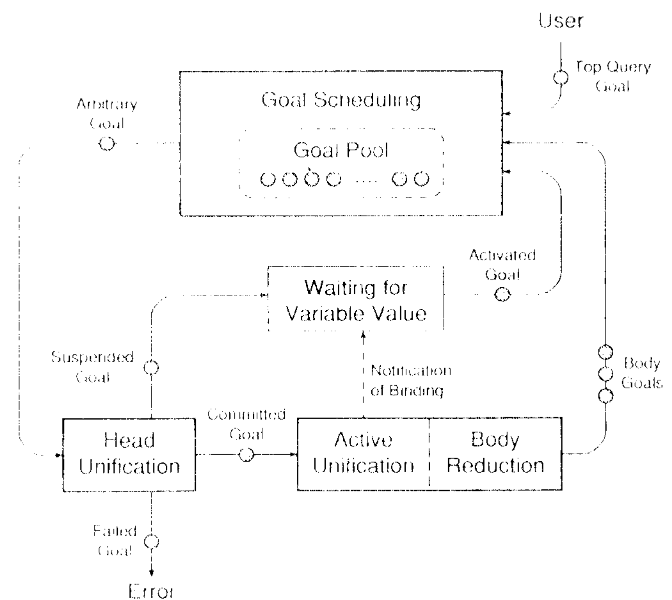


Fig. 2. Pure execution model of Fleng.

without any backtracking. If no clause head is unifiable with a goal, an error message is printed out, and the goal is just thrown away; failure of one goal does not affect the other goals, because there are no implicit AND relations among goals in Fleng. Since there are some clauses with an empty body part, the goal pool eventually becomes empty, and the execution ends.

A goal may contain a logic variable, but it is a single-assignment variable in Fleng; the variable is bound with a value just once. For this reason, a variable is bound only after commitment by active unification during body reduction. If a particular value of an undefined variable is necessary for head unification, execution of the goal is suspended. When another goal subsequently binds the variable with a value, the suspended goal is re-activated and returned to the goal pool.

In this pure execution model, goals in the pool can be executed in arbitrary order, or even in parallel. Provided the program is correct, a correct result will be guaranteed in any scheduling order by synchronization using single-assignment variables. In an extreme case, if there are an infinite number of processors, all goals in the pool can be executed simultaneously. Hence, neglecting the difference of execution time between

goals, we can regard the number of goals in the pool as the maximum parallelism at the time.

2.1. Implementation in PIE64

In the implementation in PIE64, one goal corresponds to one thread; a thread is created when a new goal is created, and the thread is terminated when the goal is reduced to other goals. In other words, a thread is the incarnation of a goal. In the rest of the paper, we will use the term *thread* to emphasize an implementation point of view, but it is sometimes used in the almost same meaning as a *goal*.

During the reduction process, new variables, new structural data, and new goals are dynamically allocated in heap memory, and the heap memory is distributed among PEs. When one PE has consumed most of its heap memory, global garbage collection is triggered. Therefore, we have to balance heap memory consumption among PEs as well as the number of threads to be executed.

While there is no specific scheduling in the pure execution model, we introduce some scheduling strategy in the actual implementation. In this strategy, a goal is classified and executed by an appropriate type of thread. There are the following types of thread:

- A *producer* thread executes a goal which will produce a lot of data in heap memory.
- A *fork* thread executes a goal which will be reduced to a lot of goals.
- A *regular* thread executes other goals.

Since the classification of goals is done by the compiler, and used only for efficiency and to avoid failure due to resource exhaustion, semantics of programs are not changed from those in the pure execution model. So, programmers need not be concerned.

Still, we introduce a user defined goal priority for convenience. It is used as the priority of the thread which executes the goal.

The general concept of our scheduling strategy will be described in detail in Section 5.2. However, there are still some possibilities, e.g. thread types are completely distinct, or they are compatible and expressed in degree. Since such a decision needs quantitative consideration, they are left as open questions. The final decision will be made after experiments on actual implementations.

When we go into more detail, the compiler may create a single thread from a parent thread and a single child thread. However, such an optimization is done after thread classification, and it is applied only if
(1) the child thread has an appropriate type and priority,
(2) it is assigned to the local PE, and
(3) it is guaranteed to be executed without suspension.
Since most discussion here is before such an optimization, one-to-one mapping of goals onto threads is our main concern.

3. Parallel inference engine PIE64

PIE64 [7] consists of 64 processing elements and two interconnection networks. A processing element is called an Inference Unit, or 'IU'. Fig. 3 is a rough block diagram of IU, which consists of UNIRED (Unifier-Reducer), NIP (Network Interface Processor), MP (Management Processor) and local memory.

The interconnection networks of PIE64 have an automatic load balancing facility, by which the least loaded IU can be automatically selected as a target of communication. With this facility, load

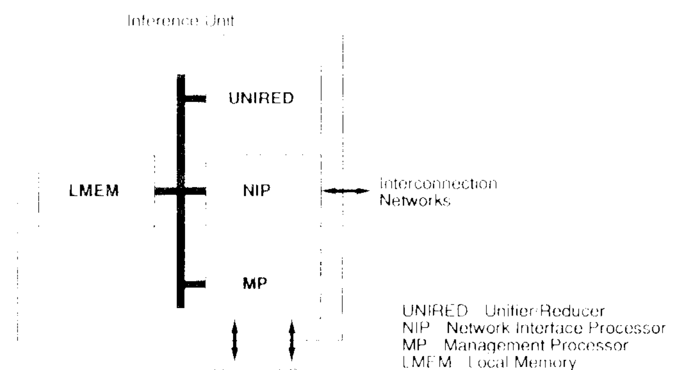


Fig. 3. A processing element of PIE64 is called an Inference Unit, which consists of three kinds of processors and local memory. UNIRED, NIP, and MP perform computation, communication and synchronization, and parallel management, respectively.

levels declared by IUs are compared and propagated in the network, and each switch in the network knows the direction in which the least loaded IU exists. The details of this facility and the implementation of the network are described in [14].

This facility also makes it possible to judge whether the entire parallelism is sufficient or insufficient, using the least load level which is finally passed to each IU from the network; if the least load level is high, all IUs are busy, and the entire parallelism is sufficient.

The two networks are called PAN (Process Allocation Network) and DAN (Data Access Network), and they are physically independent of each other, in order to separate different kinds of communication; PAN is mainly used for load distribution, where high throughput is important, and DAN is mainly used for remote memory access, where low latency is important. While the ideal architecture for each network should be different, we simply adopted the same architecture in order to reduce design cost. Accordingly, both networks have the automatic load balancing facility.

In addition, the load level is fully controllable by the parallel management kernel. Therefore, each network can be used for different kind of load distribution. In fact, PAN does ordinary load distribution (using load information about the number of threads); DAN does special load distribution and remote heap allocation (using load information about heap memory consumption) as well as remote memory access, which is still the principal role of DAN.

UNIRED [12] is a dedicated processor for executing Fleng programs. Fleng programs are compiled into UNIRED instructions, and unification and reduction of each goal is performed by a thread in UNIRED. UNIRED accesses heap memory in a single address space throughout all the IUs, i.e. PIE64 has distributed shared mem-

Table 1
Principal commands to/from management processor

Command	Description
MP → UNIRED	
reduce	Start a thread to perform unification and reduction of a goal.
UNIRED → MP	
newgoal	Deliver a new goal, created by reduction, to MP.
suspend	Inform MP of suspension of unification with the undefined variables which have caused the suspension.
endreduce	Inform MP of termination of a thread, because no recursive goal is created by reduction.
fail	Inform MP of failure of unification.
MP → NIP	
writem	Make NIP transfer a goal when a destination IU is determinate.
writel	Make NIP transfer a goal when a destination IU is indeterminate. The destination IU is determined by the automatic load balancing facility of the interconnection network.
suspend	Register a suspension record to an undefined variable.
NIP → MP	
newmsg, newload	Deliver a goal, transferred from another IU, to MP.
activate	Inform MP of a suspension record when a value is assigned to the undefined variable to which the suspension record has been registered.

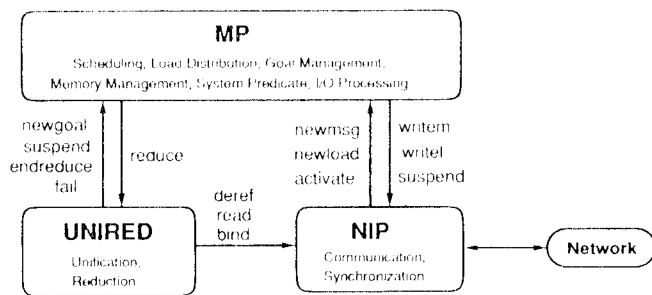


Fig. 4. Cooperative Processing Model in Inference Unit. Execution of Fleng programs is performed by exchanging commands between three kinds of processors.

ory or NUMA (Non-Uniform Memory Access time) architecture. Another feature of UNIRED is a multi-threaded architecture; the pipeline of UNIRED is shared by four threads in order to fill it during remote memory references.

NIP [13] is a dedicated processor for communication and synchronization. NIP accepts commands from UNIRED or MP, and performs them by specialized hardware sequencer.

MP executes the parallel management kernel. We use a general purpose high-speed RISC processor, SPARC, as the controller of MP. This is because generality is important in order to investigate various algorithms of parallel management, and to obtain basic knowledge for future implementation of MP in dedicated hardware.

3.1. Cooperative processing model

These three kinds of processors are connected through a high-speed command bus, and cooperate by exchanging commands, as shown in Fig. 4. Table 1 shows the principal commands to and from MP. The following is an outline of this cooperative process:

On receiving `reduce` with a goal from MP, UNIRED begins execution of a thread to perform unification and reduction of the goal. On a remote memory reference during execution, UNIRED sends `deref` or `read` to NIP. In order to assign a value to a variable, UNIRED sends `bind` to NIP. UNIRED delivers a new goal, created by reduction, to MP by `newgoal`. UNIRED also sends `suspend`, `endreduce`, and `fail` to MP, on suspension of unification, on termination of a thread, and on failure of unification, respectively.

In order to transfer a goal for load distribution, MP sends `writem` or `writel` to NIP. In the destination IU, after the transfer, NIP sends `newmsg` or `newload` to MP to notify arrival of a goal.

The following is a synchronization sequence based on a single-assignment variable of Fleng: First, on a reference to an undefined variable, UNIRED sends `suspend` to MP, and suspends execution. MP allocates a suspension record, and sends `suspend` to NIP. Then NIP registers the suspension record on the undefined variable. The reason for suspending through MP is to let MP manage suspended goals and to allow a goal to suspend on multiple variables. When the value of the variable is subsequently determined, NIP receives `bind`. Then NIP binds the variable with the value, and sends `activate` to MP for each of all the suspension records which have been registered on the variable. MP marks the suspension record as 'once activated' to avoid multiple activations, and re-inserts the goal into the scheduling queue. Communication required during suspension and activation among IUs is performed automatically by NIP.

4. Discussion on parallel management

Here, we present a general discussion on parallel management and its requirements, not limited to the case of PIE64. In the subsequent section, we show how these general requirements are treated within PIE64.

In a conventional operating system, ordinary management provides application services which are usually performed in response to explicit requests. The processing cost for such management is determined by characteristics of a requesting program and by the size of the original problem, but is independent of the granularity of parallel processing.

In our case, the purpose of parallel management is not to offer services in response to a request, but to reduce execution time of a program. Therefore, even if there is no explicit re-

quest, parallel management must implicitly support efficient parallel execution.

In order to clearly distinguish parallel management from ordinary services such as disk resource management, we define it as *management which is needed for each thread in parallel processing.* As the granularity of parallel processing becomes finer, the processing cost for parallel management grows rapidly.

In a sequential computer or a coarse-grained parallel computer, parallel management is not expensive, and is easily performed in the same processor as basic computation. However, in a highly parallel computer, finer granularity means parallel management becomes costly. If parallel management is performed in the same processor as basic computation, the overhead grows intolerably. Efficient execution can be achieved only if the parallel management is overlapped and performed in a different unit from the basic processor.

There are several kinds of parallel management, as follows:

(1) *Load distribution.*
To determine partitioning of a program into threads, and to determine assignment of threads to processing elements (PEs).

(2) *Scheduling.*
To determine execution order of threads assigned to the PE.

(3) *Synchronization.*
Mutual exclusion or rendez-vous between execution of threads.

(4) *Execution control.*
To interrupt or quit execution of threads in response to user's request, and to trace execution with a debugger.

(5) *Address space management.*
If each thread has its own address space, they require parallel management. If address space is shared between different threads, separate management for each thread is not needed.

(6) *Context management.*
Management of execution context for each thread, e.g. register values and stacks. If there is no stack, register values are the only context to be managed.

Scheduling, synchronization and execution control are related to each other, in the sense that all of them handle the execution state of a thread, e.g. waiting, executable and executing. However, they are distinguished as follows: the case where synchronization has to be strictly obeyed to get a correct result. Alternatively, scheduling is not strict but more flexible, and should be done as well as possible for efficient execution. Finally, execution control is distinguished from the others, because it is exceptional and its purpose differs from the original intent of the program.

If a synchronization primitive exists in the programming language, the synchronization mechanism is often determined in detail. Hence, synchronization is rather easy to implement in hardware to obtain good performance.

The synchronization primitive offered in Fleng is a single-assignment variable, and as described in Section 3, we have implemented the main part of its process in hardware as the NIP commands, `suspend`, `bind`, and `activate`.

Address space management is in general resource intensive. For this reason, even in a coarse-grained parallel computer, address space is often shared by different threads in order to reduce the cost of thread creation and termination.

Besides an address space for each thread, one must consider the management of address space for each PE and garbage collection of heap memory. If each PE has its own address space, an imported- and exported-address table must be used [1]. Its merits include ease of local garbage collection, but routine management of the table is still expensive. On the other hand, if the address space is shared by all the PEs, the routine management cost can be reduced, but local garbage collection may be difficult.

In PIE64, heap memory has a single address space for all the IUs, and it is shared by all the threads. Hence, it is not necessary to manage address space for each thread or for each IU. Garbage collection of the heap memory is performed in all the IUs at the same time.

Context management must manage stacks, if threads have them. However, in highly parallel symbolic processing, the number of threads is

unknown even at the beginning of execution, and it is also difficult to predict the required depth of a stack. Therefore, stack management must allow a dynamically growing stack. If threads do not have their own stack, management costs can be reduced.

In PIE64, threads do not need stacks. Hence, stack management is unnecessary. The context for each thread is held in a goal frame which is allocated in heap memory. Since UNIRED performs loading and storing context between the goal frame and the registers by itself, MP manages just the top address of the goal frame as the context of the thread.

In the following, we summarize requirements for load distribution and scheduling, which have a great effect on execution efficiency.

4.1. Requirements for load distribution

(1) Extraction of concurrency,
(2) Load balancing,
(3) Reduction of communication.
First, sufficiently high concurrency has to be extracted to make all the PEs operate simultaneously. Second, the load must be balanced among PEs in order to avoid overload on particular PEs. The load measure should incorporate not only

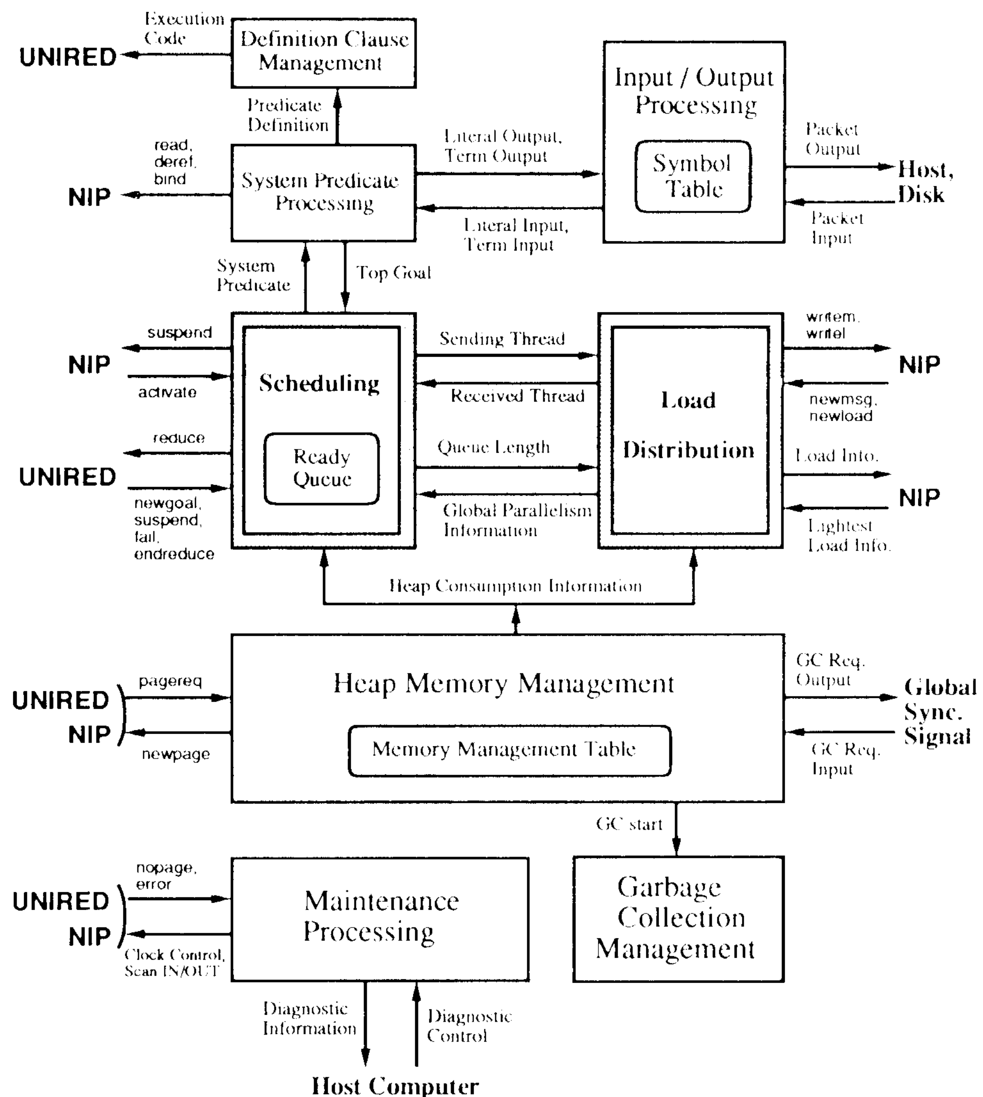


Fig. 5. Organization of parallel management kernel.

the number of threads, but also the consumption of heap memory. Third, communication between PEs has to be reduced to avoid a bottleneck.

Let us assume that load distribution comprises two sub-problems, *load partitioning* and *load assignment*. In general, the requirement for load partitioning is concurrency extraction and communication reduction; and for load assignment, is load balancing and communication reduction. However, if the distance between PEs is nearly equal, the requirements for load assignment are reduced to only load balancing. Since concurrency extraction and communication reduction are incompatible, a trade-off is required.

4.2. Requirements for scheduling

(1) Minimize synchronization cost,
(2) Avoid exhaustion of heap memory,
(3) Exploit parallelism but avoid explosive parallelism,
(4) Observe priority specified by user.

First, even if synchronization is supported by dedicated hardware, premature scheduling of a thread suffers increased context switching, because the thread has to be suspended due to incomplete data. In order to reduce the synchronization cost further, a thread should be scheduled after all the required data are obtained.

Second, in a parallel program utilizing stream parallelism, flow control must be included in the stream between a producer and a consumer. Without flow control, execution of the program may fail intermediately because of exhaustion of the heap memory. There are several programming techniques for flow control, e.g. demand driven computation and bounded buffer. However, they impose a burden on a programmer and increase synchronization costs. Furthermore, if, in the consumer, each thread for a datum in the stream can run in parallel, that kind of program has potential to provide sufficiently high parallelism. Nevertheless, the programming techniques above seriously restrict the concurrency which can be extracted. Hence, scheduling should support the flow control of a stream by incorporating dynamic priority control according to the remaining quantity of the heap memory.

Third, when parallelism is insufficient, higher priority should be given to the threads which will increase the parallelism. On the contrary, when parallelism is too high, those threads should have lower priority in order to avoid explosive parallelism and resource exhaustion. This is related to the previous requirement.

Last, the scheduling strategy must respect priority specified by a user.

5. Parallel management kernel

The parallel management kernel is executed by MP in each IU, and performs low-level system management. Its organization is shown in Fig. 5. The most significant function of the kernel is load distribution and scheduling. The kernel uses heap consumption information obtained from the heap memory management, and load information of other IUs from NIP, to determine the best load distribution and scheduling.

5.1. Load distribution

5.1.1. Three-stage load distribution

In PIE64, load distribution is not handled at the parallel management kernel alone, but handled at three stages as shown in Fig. 6.

(1) *Static load partitioning by the compiler.*
The compiler carefully optimizes the tactic at each point where a load distribution can take place in the program, i.e. where a thread is created and where a datum in heap memory

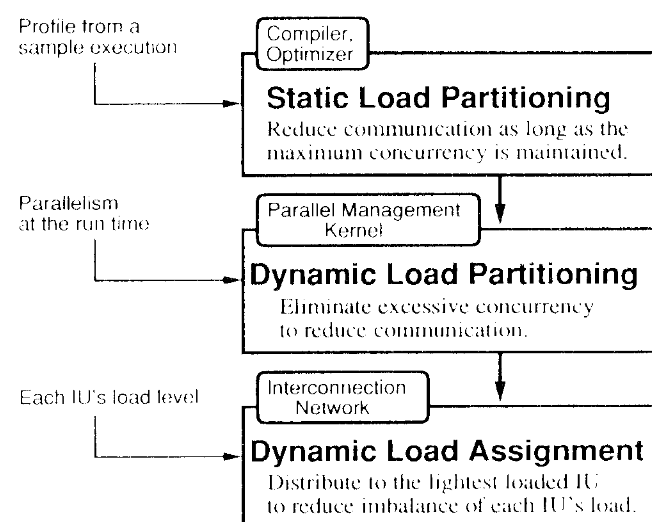


Fig. 6. The three-stage load distribution in PIE64.

is allocated. The optimization is done so as to enhance memory reference locality as long as the maximum concurrency is maintained. This corresponds to partitioning according to data dependency. The details are described in [6].

(2) *Dynamic load partitioning by the parallel management kernel.*
The parallel management kernel makes an easy decision regarding whether partitioning should be done or not, according to the parallelism at the time and the remaining quantity of heap memory. It eliminates excessive concurrency and further reduces communication. This corresponds to partitioning according to a computation tree, or control dependency. The details are described later.

(3) *Dynamic load assignment by the interconnection network.*
When partitioning is done, the load is assigned to the lowest load IU in order to balance the load among IUs. This assignment is done by the automatic load balancing facility of the interconnection network, and momentary concentration on the lowest load IU is completely avoided [14]. If partitioning is not done, the load is held in the local IU.

The basic idea of this method is that the first stage reduces communication, and the second stage further reduces communication by eliminating excessive concurrency, and finally, the third stage balances load.

5.1.1.1. Fusion of load partitioning and assignment stages. Since the fusion of these stages is realized by available tactics in static load partitioning, we summarize them here.

In the static partitioning, a tactic is determined for each of the following kinds of points in a program:

- where a structural datum or a variable is created in heap memory: heap memory is allocated from the IU specified by a tactic;
- where a thread is created: the thread is executed by the IU specified by a tactic.

The following kinds of tactics are available for each point:

(1) *Any* IU: the datum or the thread can be assigned to any IU. Partitioning may take place here. Actual assignment will be done in a later stage.

(2) *Local* IU: The datum or the thread should be assigned to local IU, i.e. the same IU as the parent thread.

(3) *Same* IU: the datum or the thread should be assigned to the same IU as a structural datum (or a variable) which has already been allocated. (Data available for reference are limited to those within the same clause.)

In contrast to the tactic of *any* IU, the latter two tactics can be regarded as tactics of a *definite* IU.

It is important to note that these tactics use only relative designation, not absolute designation with explicit IU number. This is also true in the stage of dynamic partitioning. The fusion is realized because the tactic of *any* IU is available only at the last stage, and the assignment of IU is performed only in the last stage. Scalability is also guaranteed, because the same code can be executed with any number of IUs.

Note further the opportunity to apply tactics for load distribution at memory-allocation time. One example is to allocate a datum in the *same* IU as the variable which will be bound with a pointer to the datum. Another example is to allocate a datum in *any* IU in advance of sending a thread to the *same* IU, so that the thread can access the datum locally. Such tactics in a writer thread are helpful to enhance a reader's locality, based on the assumption of multiple reads. In addition, while implementing these tactics requires remote heap allocation, NIP has such a capability.

5.1.2. Dynamic load partitioning by parallel management kernel

When one of the following conditions is satisfied, the parallel management kernel obeys the tactic designated statically by the compiler, and partitioning is performed at the points where the static tactic is *any* IU:

- the overall parallelism is low,
- the IU is too heavily loaded, compared with the overall parallelism,
- the remaining heap memory is low, and a thread of *producer* type will be created at the point.

When none of the above are satisfied, the paral-

lel management kernel does not perform partitioning, and selects the local IU at the points where the static tactic is *any* IU. At the points where the static tactic is a *definite* IU, some other methods are possible. One method is to obey the static tactic, and another is to select the local IU independent of the static tactic. A third method is a composite of the two. The difference between effect of these methods will be shown in the section on preliminary evaluation.

As described in Section 3, two networks are used with different kinds of load information; PAN is used to balance the number of threads, and DAN is used to balance heap memory consumption. These networks are used as follows:

- In load assignment of a thread,
  -when the remaining quantity of the heap memory is sufficient in all the IUs, PAN is always used,
  -when there is an IU lacking heap memory, DAN is used for a thread of *producer* type, and PAN is used for other threads,
- In load assignment of a datum in heap memory, DAN is always used.

In the actual implementation, allocation of heap memory is performed by a thread on UNIRED. Therefore, two kinds of codes for UNIRED are prepared, i.e. with or without partitioning on allocation of heap memory. Then MP controls whether partitioning is done or not, by changing the dispatch target of `reduce`.

5.2. Scheduling

Scheduling takes into account the following factors, to select the next thread to be executed:
(1) static priority,
(2) dynamic priority,
(3) respite time in starting execution.

Static priority is designated by a user, and enables a preceded process and speculative computation as described above.

Dynamic priority is variable priority determined according to parallelism or consumption degree of heap memory at run time. Depending on characteristics of the thread, there are several kinds of dynamic priority as follows:

- If there is little room in heap memory after garbage collection, lower priority is given to the threads of *producer* type, which will likely produce a lot of data in heap memory.
- If the entire parallelism is insufficient, higher priority is given to the threads of *fork* type, which will create a lot of threads.

A respite time in starting execution is assigned to each thread. It is defined as the minimum time from when the thread is created until it begins execution. The respite time is introduced to start a thread after all the relevant variables become definite. Providing respite time according to data dependency statically, the number of suspensions, and hence the synchronization cost, can be reduced.

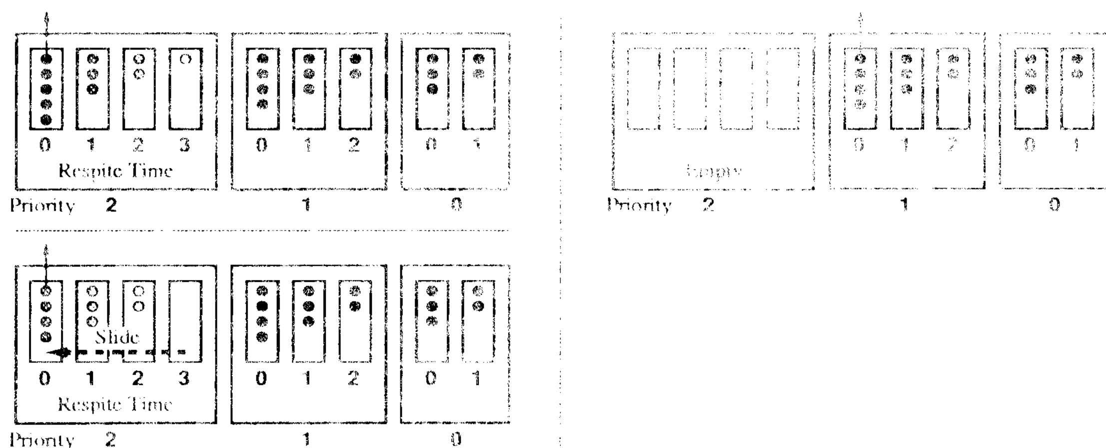


Fig. 7. Sliding ready queue for each priority.

In some situations, adequate respite time cannot be determined, since data dependency may change dynamically at run time. In this case, a variable which is always referred to by the thread, is examined statically. After the thread is created, the registration of suspension is done immediately, instead of making it executable. The registration of suspension before execution does not involve context switching of UNIRED, and consequently, it has very little overhead compared with a normal suspension and the synchronization cost can be reduced.

The ready queue which holds runnable threads, forms sliding queues for each priority as shown in Fig. 7. The respite time is measured by thread generation, instead of by the exact time. When there are no more threads of zero respite time, the thread generation is changed, and the queues in the same priority are slid. Lower priority threads are scheduled only if the higher priority queues are empty. A thread activated by variable binding is put into the queue of zero respite time with an appropriate priority.

Respite time is determined by global control- and data-dependency analysis using the profile data that indicate how frequently each clause is committed to. In short, n respite time is assigned to those threads which require data calculated by threads in $n-1$ respite time, and zero respite time is assigned to those threads which can be executed immediately. The optimization mentioned in Section 2 is applied only to threads with zero respite time. Since the ready queue in each IU is asynchronously slid, some heuristics may be worthwhile in the actual implementation in PIE64.

5.3. Management for resources

In addition to load distribution and scheduling, the parallel management kernel performs heap memory management, input and output processing, clause management, system predicate processing, maintenance processing, and so on.

The address space of the heap memory is a global physical address space shared among all the threads, and there is no address translation table. The heap memory is distributed among IUs, and it is logically divided into pages for management. The kernel allocates heap memory to UNIRED and NIP in units of pages, and allocation in units of words is done within UNIRED or NIP.

The garbage collection of the heap memory is a global garbage collection, which synchronizes all the IUs [8]. Synchronizations at beginning and phase transitions of garbage collection are taken by barrier synchronization signals. Actual processing, such as marking, is done by UNIRED, and an indirect reference table for remote pointers is temporarily made by NIP during garbage collection. The kernel performs management of phase transitions and UNIRED's contexts, reception of marking requests from other IUs, and so on.

In input and output processing, the kernel translates packets from the host computer or peripherals into a stream of literals or terms, and vice versa. In clause management, the kernel manages compiled codes of UNIRED. In system predicate processing, the kernel interprets embedded predicates of Fleng, to invoke various procedures in the kernel and to perform floating point calculations. In maintenance processing, the kernel supports diagnostic analysis with maintenance programs in the host computer.

6. Preliminary evaluation

In the three-stage load distribution in PIE64, both static load partitioning and dynamic load partitioning have an effect on reducing communication. Hence, we have done preliminary evaluation for comparison between the effects of these two kinds of load partitioning.

We used a Fleng interpreter in a Unix workstation for the evaluation. For each goal and each datum in heap memory, a history is traced to examine the IU where it resides. A memory reference is judged local or remote by examining whether or not the IUs of the executing goal and the referred-to datum are the same. In order to make the scheduling characteristics similar to those in a parallel system, the scheduling is per-

formed in a breadth-first manner using two goal queues. One queue is dedicated to enqueuing, and the other to dequeuing. When the queue for dequeuing becomes empty, two queues are alternated. An alternation of the queues is called a generation transition, and a period between two successive alternations is called a generation. Therefore, assuming that every goal takes the same execution time, we can obtain maximum parallelism in each generation from the queue length at the generation transition immediately before, as described in Section 2. (Note that this goal generation is different from the thread generation in Section 5.2, because a thread with zero respite time is in the same thread generation as its parent, but not in the same goal generation. The thread generation is used to reduce synchronization cost; the goal generation is used to measure parallelism.) While the sample program used is a program of 6 Queens, it is satisfactory for this preliminary evaluation, because we don't expect precise results for making a trade-off. The purpose of this evaluation is to characterize the static and dynamic load partitioning.

Measurement is done for the following six strategies:

(1) *Simple strategy:* Allocate all the data in the local heap memory. Perform one recursive goal in the local IU; distribute each of the other goals by automatic load balancing.

(2) *Static strategy:* Obey the static strategy determined before execution from profile data. Where the tactic is *any* IU, perform automatic load balancing.

(3) *Dynamic strategy:* At each generation transition, examine the number of active IUs. Then, in order to avoid excessive load partitioning, let the number of inactive IUs be the maximum number of times of automatic load balancing in the following generation. (These are used also in the following composite strategies.) Until maximum automatic load balancing takes place, obey the simple strategy. After that, always select the local IU, because all IUs are now active. (A similar situation can actually be realized in PIE64, using the least load level passed from the network.)

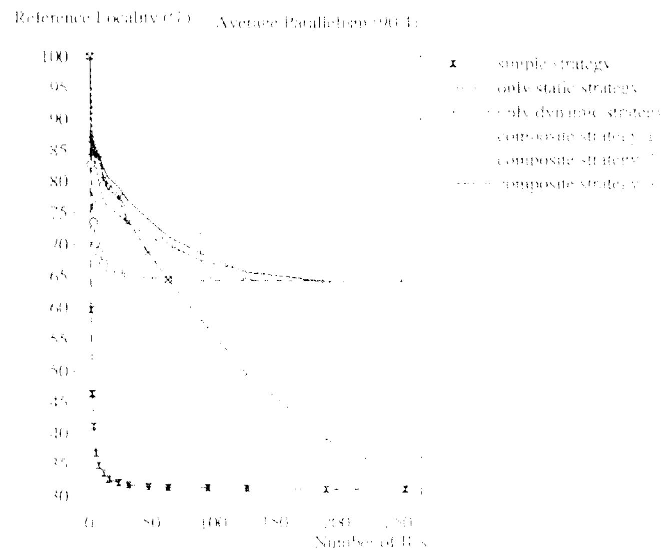


Fig. 8. Variation in memory reference locality versus the number of IUs (6 Queens).

(4) *Composite strategy-1:* Until maximum automatic load balancing takes place, obey the static strategy. After that, where the static tactic is *any* IU, select the local IU. In the other places, obey the static strategy.

(5) *Composite strategy-2:* Until maximum automatic load balancing takes place, obey the static strategy. After that, perform all the goals in the local IU. For heap memory allocations, if the static tactic is *any* IU, select the local IU; otherwise, obey the static strategy.

(6) *Composite strategy-3:* Until maximum automatic load balancing takes place, obey the static strategy. After that, select always the local IU in spite of the static strategy.

Fig. 8 shows the variation in memory reference locality against various numbers of IUs for each strategy. The average maximum parallelism among generations was 90.4 goals. The static strategy ensures a regular level of memory reference locality in spite of the relation between parallelism and the number of IUs. The dynamic strategy is more effective than the static strategy when there are a small number of IUs. However, as the number of IUs increases, partitioning takes place more frequently, and the effectiveness of the dynamic strategy dramatically decreases finally to that of the simple strategy. All composite

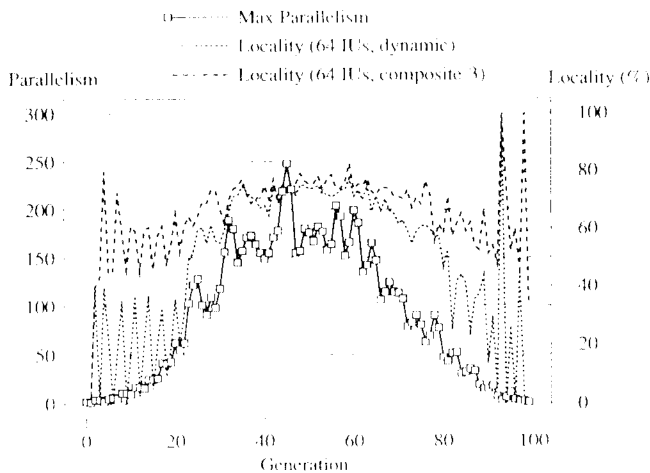


Fig. 9. Variation of parallelism versus variation of memory reference locality.

strategies show good effectiveness. However, the composite strategy-1 and -2 are not as good as the composite strategy-3. This is because remote pointers are reproduced, and persist due to specific load distribution and remote heap allocation after all the IUs become active in the composite strategy-1 and -2. In addition, the composite strategy-1 has a little less locality than the dynamic strategy at a small number of IUs. This is because control dependency does not affect partitioning in the composite strategy-1 as much as in the dynamic strategy. Furthermore, the best effectiveness is always observed when using the composite strategy-3, which has the biggest weight of partitioning according to control dependency in the composite strategies.

Fig. 9 shows the variation in the maximum parallelism in each generation, and the variation in memory reference locality between the dynamic strategy and the composite strategy-3. The number of IUs was assumed to be 64. This figure also shows that the dynamic strategy is effective only when the parallelism exceeds the number of IUs. Note that the composite strategy-3 has no reduced effectiveness, and therefore, transition between the static strategy and the dynamic strategy is smooth.

From the results described above, we conclude that the dynamic strategy is important if the parallelism exceeds the number of IUs to a high degree; otherwise, the static strategy is important; and the composite strategy, which combines both of their merits, is the best.

7. Conclusion

We have discussed problems in parallel management, and described the architecture of the parallel management kernel of PIE64, focusing on load distribution and scheduling. In PIE64, load distribution is handled at three stages, namely, static load partitioning, dynamic load partitioning, and dynamic load assignment. The parallel management kernel bears the responsibility for the dynamic load partitioning, eliminating excessive concurrency and reducing communication. The scheduling of the kernel introduces dynamic priority to avoid heap memory exhaustion and to extract parallelism. It also introduces the concept of respite time in starting execution, to reduce the cost of synchronization. We have also shown the difference between dynamic and static partitioning, and concluded that a composite method is most desirable.

Naturally, the following points are important for future work:

- static analysis at compile time,
- implementation, evaluation and improvement of the parallel management kernel for PIE64,
- implementation of the management processor in dedicated hardware.

Furthermore, in the same way as the Memory Management Unit was required for Virtual Memory instead of painful overlaying methods, some dedicated hardware will be required for completely automatic load distribution and scheduling. The final goal of our research is to clarify the function and the architecture of such a 'Parallel Management Unit'.

8. Acknowledgements

We are thankful to Professor Randy Goebel and Mrs. Gita Mithal for polishing up this paper. This work has been supported by Grant-in-Aid for Scientific Research (No. 62065002, No. 03555071, No. 03003891) from the Ministry of

Education, Science and Culture. One of the authors, Hidaka, is supported by JSPS Fellowships for Japanese Junior Scientists.

[1] S. Abdullahi, E. Miranda and G. Ringwood, Collection schemes for distributed garbage, *Proc. 1992 Int. Workshop on Memory Management, LNCS 637* (Springer, Berlin, 1992), 43–81.

[2] Arvind and R.A. Iannucci, Two fundamental issues in multiprocessing, Computation Structures Group Memo 226-6, Laboratory for Computer Science, MIT, 1987.

[3] T.L. Casavant and J.G. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Trans. Software Eng.* (2) (1988) 141–154.

[4] S. Gregory, *Parallel Logic Programming in Parlog: The Language and its implementation*, (Addison-Wesley, Deading, MA, 1987).

[5] Y. Hidaka, H. Koike and H. Tanaka, The architecture of the inference unit of Parallel Inference Engine PIE64 (in Japanese), IEICE Technical Report CPSY90-44, The Institute of Electronics, Information and Communication Engineers, Japan, 1990, 37–42.

[6] Y. Hidaka, H. Koike, J. Tatemura and H. Tanaka, A static load partitioning method based on execution profile for committed choice languages, *Proc. 1991 Int. Symp. on Logic Programming* (1991) 470–484.

[7] H. Koike and H. Tanaka, Overview of the Parallel Inference Engine: PIE64, Annual Report of Engineering Research Institute, Faculty of Eng., Univ. of Tokyo, Vol. 48 (1990) 63–68.

[8] H. Koike and H. Tanaka, Generation scavenging GC on distributed-memory parallel computers, *Proc. High Performance and Parallel Lisp Workshop*, London (1990).

[9] M. Nilsson and H. Tanaka, Massively parallel implementation of Flat GHC on the Connection Machine, *Proc. Int. Conf. on Fifth Generation Computer Systems* (1988) 1031–1090.

[10] E. Shapiro, Systolic programming: A paradigm of parallel processing, *Proc. Int. Conf. on Fifth Generation Computer Systems* (1984) 458–470.

[11] E. Shapiro, ed., *Concurrent Prolog: Collected Papers*, Vols. 1, 2 (MIT Press, Cambridge, MA, 1987).

[12] K. Shimada, H. Koike and H. Tanaka, UNIRED II: The high performance inference processor for the Parallel Inference Machine PIE64, *Proc. Int. Conf. on Fifth Generation Computer Systems* (1992) 715–722.

[13] T. Shimizu, H. Koike and H. Tanaka, Details of the network interface processor for PIE64 (in Japanese), *SIG Reports on Computer Architecture* (Info. Processing Society of Japan) 87-5 (1991).

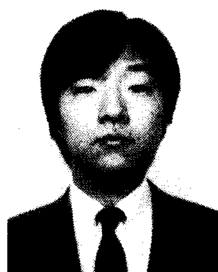
[14] E. Takahashi, H. Koike and H. Tanaka, A study of a high bandwidth and low latency interconnection network in PIE64, *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing* (1991) 5–8.

[15] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki, A load balancing mechanism for large scale multiprocessor systems and its implementation, *Proc. Int. Conf. on Fifth Generation Computer Systems* (1988) 978–986.

[16] K. Taki, Parallel Inference Machine PIM, ICOT Technical Report TR-0775, Institute for New Generation Computer Technology, Tokyo, 1992.

[17] K. Ueda, Guarded Horn Clauses, Doctoral Thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo, 1986.

was born in Tokyo, Japan in 1963. He received his B.E. degree in Precision Machinery in 1989, and his M.E. degree in Information Engineering in 1991 from the University of Tokyo, Tokyo, Japan. He established SOUM Corporation in Tokyo, Japan, in 1984, and worked as a member of the board for several years. He is currently a D.E. candidate in Information Engineering from the University of Tokyo. His research interests include computer architecture, parallel computer systems and operating systems. He is a member of IEEE Computer Society, Information Processing Society of Japan, and the Institute of Electronics, Information and Communication Engineers in Japan.



was born in Kanagawa, Japan in 1961. He received the B.S., M.E. and D.E. from the University of Tokyo in 1984, 1986, 1990, respectively. In 1990, he joined the faculty of Engineering, the University of Tokyo, where he is now a lecturer. His current research interests include parallel computer architecture and parallel computer software.

was born in Hyogo, Japan on January 15, 1943. He received the degrees of Bachelor of Electronics Engineering, Master of Electrical Engineering and Doctor of Electrical Engineering from the University of Tokyo in 1965, 1967 and 1970, respectively.

In 1970, he joined the faculty of Engineering, University of Tokyo, where he is now a processor. From 1978 to 1979, he was a visiting professor of City College of New York, NY. He published several books titled Non-Neuman Computer, Computer Architecture, VLSI Computer, Software Oriented Computer Architecture (in Japanese), etc.

His current research interests are in computer architecture, distributed systems, CAD system for LSI design, and artificial intelligence. He is a member of IEEE, ACM, Information Processing Society of Japan and IECE of Japan.