Load Distribution System of PIE64

Satoshi MURAKAMI, Hidemoto NAKADA,
Yasuo HIDAKA, Hanpei KOIKE, Hidehiko TANAKA

*Faculty of Engineering, the University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.*

{satoshi,nakada,hidaka,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract

We have been developing Parallel Inference Engine PIE64 and its description language Fleng, aiming at fast execution of large-scale knowledge processing. Fleng is a programming language for fine-grained parallel symbolic processing, and it is suitable for highly-parallel processing of non-uniform problems.

For fine-grained parallel processing, effective load distribution with small overhead is a crucial issue. In this paper, we describe the Fleng system on PIE64, focusing on its load distribution system. The results of preliminary evaluation are also presented.

1 Introduction

Parallel processing of uniform problems, such as scientific computation, have been studied for many years, and many technologies for them have been developed. However, it is difficult to apply these technologies to non-uniform problems, such as knowledge processing.

We have been developing Parallel Inference Engine PIE64 and its description language Fleng, aiming at fast execution of large-scale knowledge processing. Fleng is a programming language for fine-grained parallel symbolic processing, and it can realize highly-parallel processing of non-uniform problems by extracting control concurrency.

The problem of communication and synchronization is known as essential for parallel processing. Besides these issues, parallel management such as load distribution and scheduling is also important. Overhead of parallel management is relatively small for coarse-grained parallel processing. However, for fine-grained parallel processing, parallel management becomes more crucial for efficient execution.

In this paper, we describe the overview of Fleng system on PIE64, focusing on its load distribution system. Load distribution of PIE64 is handled at three stages: 1) static load partitioning by the compiler, 2) dynamic load partitioning by the parallel management kernel, 3) dynamic load assignment by the interconnection network. This combination of static optimizations and dynamic control enables efficient execution. Moreover, since load distribution is performed automatically, a programmer need no longer be concerned with load distribution, and he can concentrate on extracting as much concurrency as possible.

The problem of load distribution has been well-studied[1]. Much work has been done in case where program behavior is known. However, since highly-parallel symbolic processing often shows irregular behavior, it is difficult to apply known techniques to it.

A common method for highly-parallel symbolic processing is that programmer designates a strategy[2][3]. However, a programmer should concentrate on extracting as concurrency much as possible in order to get sufficiently-high parallelism.

Fleng and PIE64 are described briefly in Section 2 and Section 3, respectively. Fleng system on PIE64 is described in Section 4. The details of the load distribution system of PIE64 are described in Section 5. The results of preliminary evaluation are shown in Section 6. Section 7 concludes this paper.

2 Committed-Choice Language Fleng

Fleng is a programming language for fine-grained parallel symbolic processing, and is one of Committed-Choice languages, or parallel logic programming languages. GHC is famous as one of those languages.

A Fleng program is a set of Horn clauses like:

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

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

The unit of execution in Fleng is called a goal. Body part of a clause consists of several body goals, and it can be regarded as invocations of processes.

Figure 1 shows the execution model of Fleng. Execution of a Fleng program begins when the top query goal is put into the goal pool. An arbitrary goal in the goal pool is selected and removed from the pool. Then, the goal is checked if it is unifiable with clauses in the program (unification). If there is a clause whose head is unifiable with the goal, the clause is chosen, and the goal is reduced to the body part of the clause (reduction). The new body goals are added to the goal pool. Execution of Fleng is the repetition of this process. When the goal pool becomes empty, execution ends.

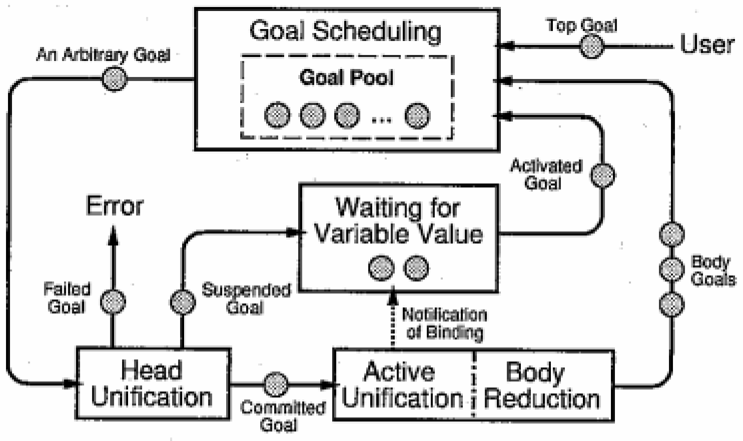


Figure 1: Execution model of Fleng

A variable in Fleng is a single-assignment variable. A variable is unbound when it is created, and it can later be bound with a value only once.

Synchronizations between goals are attained by the single-assignment variable. A variable is bound with a value by active unification during the reduction process. If a particular value of an unbound variable is required during head unification, execution of the goal is suspended and wait for the variable to be bound. When the variable is bound with a value by execution of another goal, the suspended goal is re-activated and returned to the goal pool.

3 Parallel Inference Engine PIE64

PIE64 is a parallel inference machine which is designed to execute Fleng efficiently. It consists of 64 processing elements called Inference Units(IUs) and two interconnection networks.

Inference Unit

Figure 2 shows the rough block diagram of an IU. Each IU has three kinds of processors; UNIRED(Unifier-Reducer) for computation, NIP(Network Interface Processor) for communication and synchronization, and MP(Management Processor) for management.

Heap memory in each IU can be accessed in a single global address space throughout all IUs. In other words, PIE64 has distributed shared memory or NUMA(Non-Uniform Memory Access) architecture.

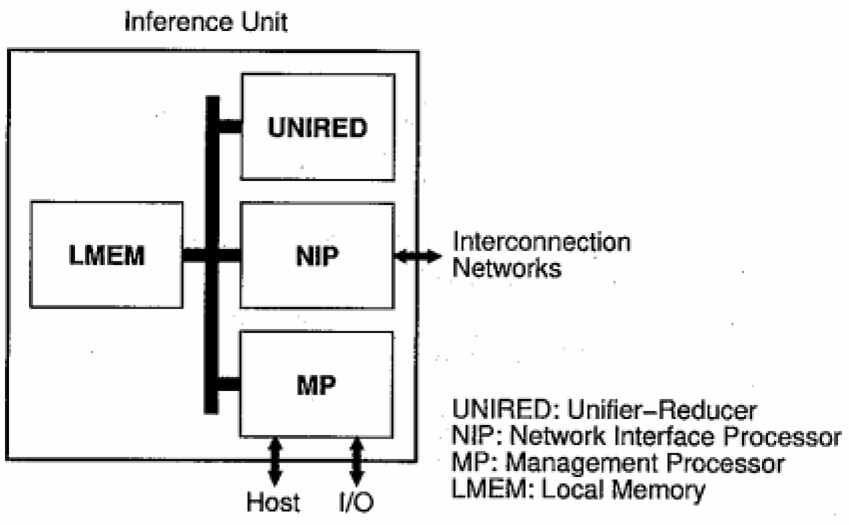


Figure 2: Diagram of the Inference Unit

UNIRED[4] is a dedicated processor for executing Fleng programs. It has a tag architecture and an ordinary RISC instruction set with some dedicated instructions for executing Fleng programs. Fleng programs are compiled into UNIRED instructions, and unification and reduction of each goal are performed as a thread in UNIRED.

NIP[5] is a dedicated processor for communication between IUs and synchronization among Fleng processes.

MP performs parallel management. We use SPARC, a general purpose RISC processor, as MP.

Interconnection Network

The interconnection network[6] is a multi-stage, circuit switching network. The most notable feature of the interconnection network is automatic load balancing facility. Each IU declares its local load level to the interconnection network, and the network can automatically select the least-loaded IU as the destination IU. As the least load level is fed back to each IU, this facility can also be used to observe global load level over the entire machine. As this facility utilizes unused resources, it yields no overhead.

4 Fleng system on PIE64

For deriving good performance from highly-parallel computers, load distribution and scheduling are important. To cope with these issues, we have adopted combination of static optimizations and dynamic control.

The Fleng system consists of a compiler system and a run-time system. In PIE64, compiled codes executed by UNIREDs and the run-time kernel executed by MPs cooperate to execute Fleng programs. The run-time system includes also a Fleng interpreter.

4.1 Fleng compiler system

Figure 3 shows the compiling process of a Fleng program.

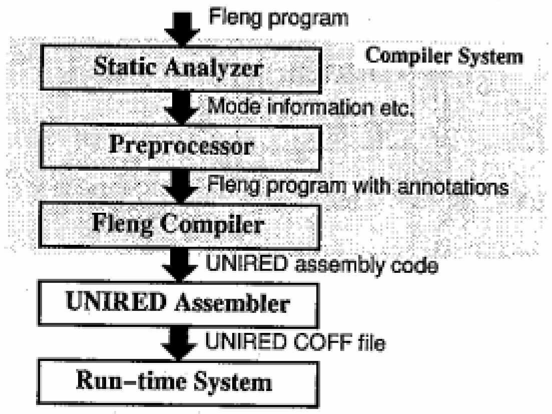


Figure 3: The compiling process of a Fleng program

The preprocessor performs load partitioning and scheduling according to the information obtained by the static analyzer. Decisions made by the preprocessor are added to the Fleng program as annotations, and the compiler generates UNIRED assembly code according to the annotations. Finally, UNIRED assembler generates UNIRED instruction codes, and they are executed by the run-time system.

The static analyzer, the preprocessor, and the compiler are written in Fleng itself.

4.2 Execution of Fleng in PIE64

Here we describe how Fleng programs are executed in PIE64.

In PIE64, three kinds of processors in each IU are connected through a high-speed command bus, and cooperate to execute by exchanging commands, as shown in Figure 4.

Execution of Fleng begins by putting a top query goal into the goal pool. In PIE64, the goal pool is managed by MP. MP selects an arbitrary goal from the pool for execution, and provides it to UNIRED by sending `reduce`. On receiving `reduce` from MP, UNIRED starts unification and reduction as a thread, and on termination of the thread, UNIRED sends `endreduce` to MP.

When memory reference is required,

- if the memory address is local, UNIRED directly reads from the address.
- if the memory address is remote, UNIRED sends `read` to NIP, and NIP in the destination IU reads from the address and returns the data.

During the reduction process, new variables, cons cells, vectors, and goals are generated dynamically.

Generation of new variables, cons cells, and vectors are performed by allocating them in heap memory. Allocation of heap memory in the local IU can be performed directly by an instruction of UNIRED. However, in order to allocate heap memory in the remote IU, UNIRED first creates the data which should be transferred to the remote IU in the temporary area of the local memory, and then transfers it by sending `writelm`[1] to NIP. In the destination IU, NIP automatically allocates heap memory and writes the received data there.

New goals generated by reduction are delivered from UNIRED to MP.

- If the goal is to be executed in the local IU, UNIRED sends `newgoal` to MP, and MP adds the goal to its goal pool.
- To distribute a goal to other IU, UNIRED sends `writelm` to NIP. In the destination IU, after the transfer, NIP sends `newload` to MP in order to notify arrival of a new goal.

On reference to an unbound variable during head unification, UNIRED sends `suspend` to MP and suspends execution of the goal. MP allocates a suspension record, and sends `suspend` to NIP. Then NIP registers the suspension record on the undefined variable.

When the value of the variable is determined, NIP receives `bind` from UNIRED. Then NIP binds the variable with the value and sends `activate` to MP for each of the suspension records registered on the variable. MP returns the goal to the goal pool. Communications required during suspension and activation are performed automatically by NIP.

[1]This command sends data to a designated IU if the destination IU is designated as an operand. If the destination IU is not designated as an operand, it sends to the least-loaded IU, using the automatic load balancing facility of the interconnection network.

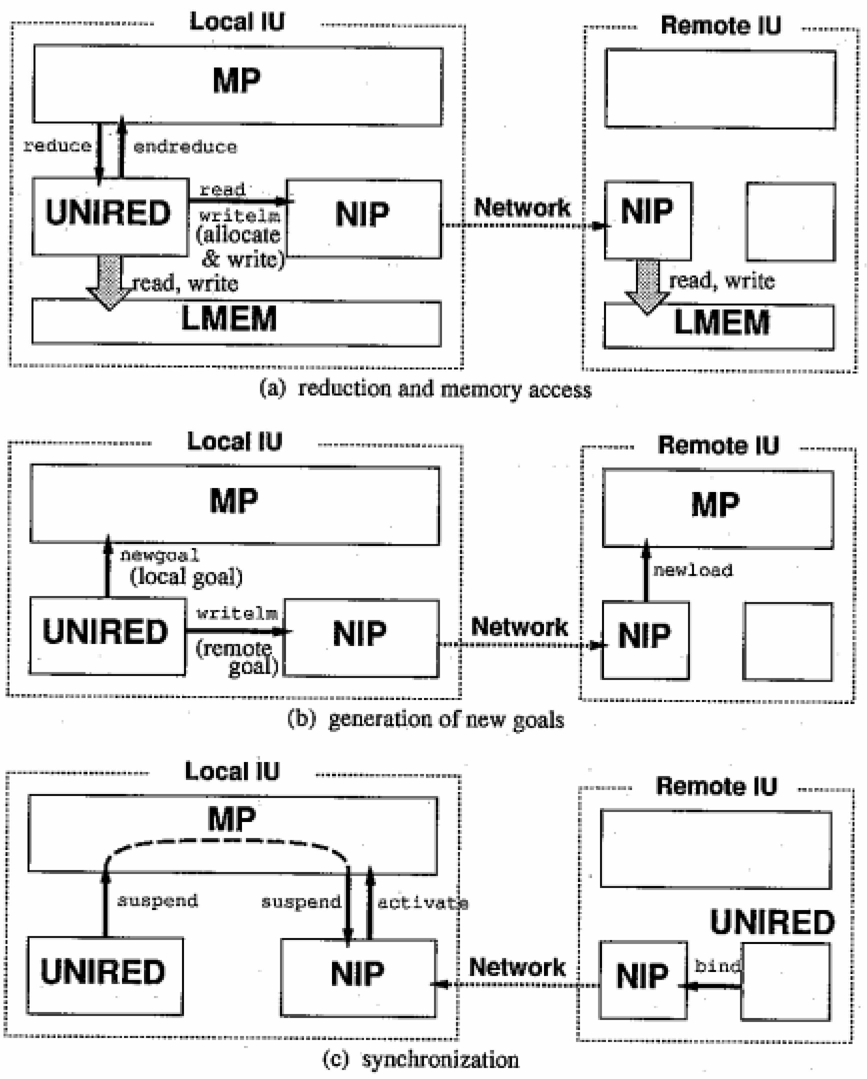


Figure 4: Cooperative Processing Model in PIE64

5 Load distribution system of PIE64

5.1 Three-stage automatic load distribution

In PIE64, load distribution is automatically handled at three stages as follows:

1. Static Load Partitioning by the compiler.
2. Dynamic Load Partitioning by the parallel management kernel.
3. Dynamic Load Assignment by the interconnection network.

In the first stage, the compiler partitions loads according to the data-dependency. This optimization is done so as to enhance memory reference locality as far as the maximum concurrency is maintained.

In the second stage, the parallel management kernel makes an easy decision whether partitioning should be done or not, according to the global load level at the time. Such global information can be collected by the interconnection network without any overhead. This optimization eliminates excessive concurrency and reduces communication further.

And finally, in the third stage, when partitioning is decided to be done by the parallel management kernel, the load is assigned to the least-loaded IU in order to balance load among IUs. This assignment is done by the automatic load balancing facility of the interconnection network. If partitioning is not taken, the load is held in the local IU.

In PIE64, load distribution is performed automatically. Thus, a programmer need no longer be concerned with load distribution, and he can concentrate on extracting as much concurrency as possible.

5.2 Static load partitioning

In the first stage of three-stage load distribution in PIE64, the compiler statically partitions loads in order to reduce communication and synchronization without loss of concurrency.

5.2.1 load partitioning tactics

A point where load partitioning is possible in the program is called a load partitioning point. We consider two kinds of load partitioning points:

1. Points where heap memory is allocated. (The IU where heap memory is allocated must be designated.)
2. Points where a goal is generated. (The IU which performs the goal must be designated.)

For each load partitioning point in the program, one of the following load partitioning tactics is selected:

Tactic A: Select the least-loaded IU.

Tactic B: Select the local IU.

Tactic C: Select the IU pointed to by a pointer obtained as an argument of the parent goal or as an element of a structure in the arguments.

Tactic D: Select the same IU as is selected by an invocation of tactic A at a different load partitioning point of heap memory allocation within the same clause.

In our compiler system, automatic designation of these load partitioning tactics has been realized. Load partitioning tactics are expressed by annotations in the form of '...`@`*annotation*', and the preprocessor designates load partitioning tactics by adding the annotations to the Fleng program.

- `any(`*label*`)`
  This is an invocation of tactic A.
- `local`
  This is an invocation of tactic B.
- `on(`*label*`)`
  This is for a datum referred to by invocations of tactic C.

- `to(`*label*`)`
  This is an invocation of tactic C or D which refers to `on(`*label*`)` or `any(`*label*`)` with the same label in the same clause.

Note that these tactics use only relative designation; absolute designation with the explicit IU number is not used. Thus, the same code can be executed with any number of IUs, and scalability is guaranteed.

5.2.2 Data-flow analysis

Static load partitioning is performed automatically by the preprocessor. To partition loads effectively, the preprocessor uses data-dependency information. First, mode analysis is performed for each clause in the program. And then, the data-flow graph is built according to the mode information.

The data-flow graph is represented by a directed graph, the nodes of which represent goals or variables. An Edge from a variable node to a goal node represents that the value of the variable is required to execute the goal. In other words, the goal has to suspend until the variable is bound. An Edge from a goal node to a variable node represents that the goal binds the variable with a value.

Figure 5 shows an example of a Fleng program. It is a part of an n-queens program. The data-flow graph of the program is shown in Figure 6.

```
check(P, D, L, [Q|Lp0], Lp, A0, A) :-
    add(Q, D, Sum),
    equal(Sum, P, R1),
    sub(Q, D, Dif),
    equal(Dif, P, R2),
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A).
```

Figure 5: Sample program

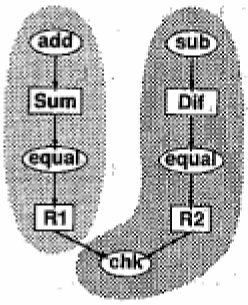


Figure 6: Data-flow graph

The problem of load partitioning can be resolved into partitioning of the data-flow graph. For any walks in the data-flow graph, any two goals on the walk cannot be executed simultaneously. For example, `add(Q,D,Sum)` and `equal(Sum,P,R1)` in Figure 6 cannot be executed simultaneously. These goals can be assigned to the same IU without any loss of concurrency.

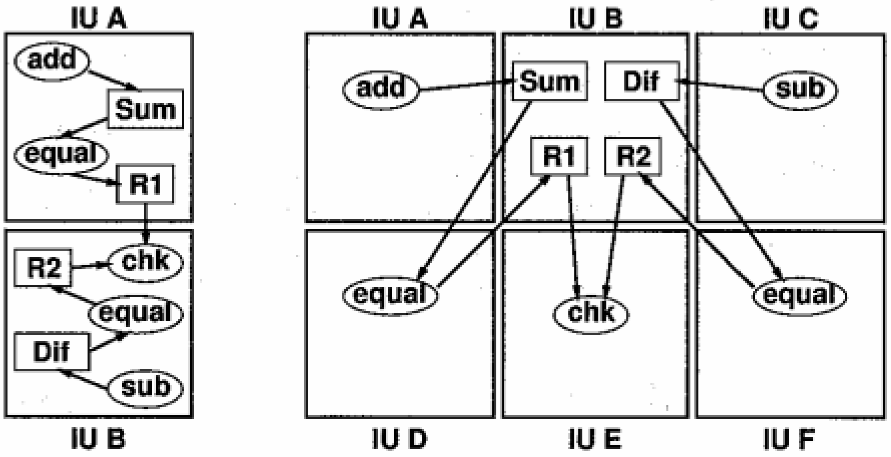
In this way, the graph in Figure 6 can be divided into two parts and assigned to different IUs. Figure 7 shows the program after static load partitioning.

Figure 8(a) shows the result of static partitioning, and Figure 8(b) shows the worst result of naive partitioning; new goals are all distributed to different IUs, while all data are allocated in the local IU. A large square represents an IU, and a directed edge between goals and variables represents a memory access.

Static load partitioning reduces the number of remote references to only one. Note that concurrency is not reduced by static load partitioning, even though only two IUs are used.

```
check(P, D, L, [Q|Lp0], Lp, A0, A) :-
    add(Q, D, Sum @ any(1)) @ to(1),
    equal(Sum, P, R1 @ to(1)) @ to(1),
    sub(Q, D, Dif @ any(2)) @ to(2),
    equal(Dif, P, R2 @ to(2)) @ to(2),
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A) to(2).
```

Figure 7: Statically partitioned program



(a) Optimized partitioning (b) Naive partitioning

Figure 8: Results of load partitioning.

5.3 Dynamic load partitioning

Dynamic load partitioning is performed in order to eliminate excessive concurrency and reduce communication.

Static load partitioning designates load partitioning tactics for each load partitioning point. However, whether partitioning is actually taken or not depends on the run-time situation. This decision is made dynamically by the parallel management kernel.

Parallel management kernel

The parallel management kernel is executed by MP in each IU, and it performs low-level system managements. The most significant function of the kernel is load distribution and scheduling. The kernel uses global load level over the entire machine obtained from the interconnection network, in order to determine the best load distribution and scheduling.

When the global load level is low, parallelism is insufficient, and loads should be distributed in order to increase parallelism immediately. So the loads are distributed according to the static tactics.

However, when the load level is high, parallelism is sufficient, and there is no need to worry about the loss of concurrency. Therefore, load distribution is not taken to reduce communication.

Implementation of dynamic load partitioning

Dynamic load partitioning is implemented as follows.

When UNIRED receives `reduce` form MP, it starts execution of a thread by dispatching to the corresponding entry-point of the compiled code. Dynamic load partitioning is implemented by changing the dispatch target of `reduce`.

First, we prepare two kinds of codes for UNIRED; 1) code which distribute loads (statically load-partitioned code), 2) code which never distribute loads.

The number of active goals in the goal pool can be regarded as the load level of the IU. Each IU declares this load level to the interconnection network. Then, as we described in Section 3, each IU can know the global load level as a feed-back from the network. MP compares the global load level with a threshold that is specified beforehand, and sets a flag which indicates whether load level is high or low. When UNIRED receives `reduce` form MP, UNIRED checks the flag and changes the dispatch target according to it. When global load level is low, UNIRED dispatches to the entry-point in the code which distributes loads. On the other hand, when global load level is high, UNIRED dispatches to the entry-point in the code which never distribute loads.

A rough diagram of this mechanism is shown in Figure 9.

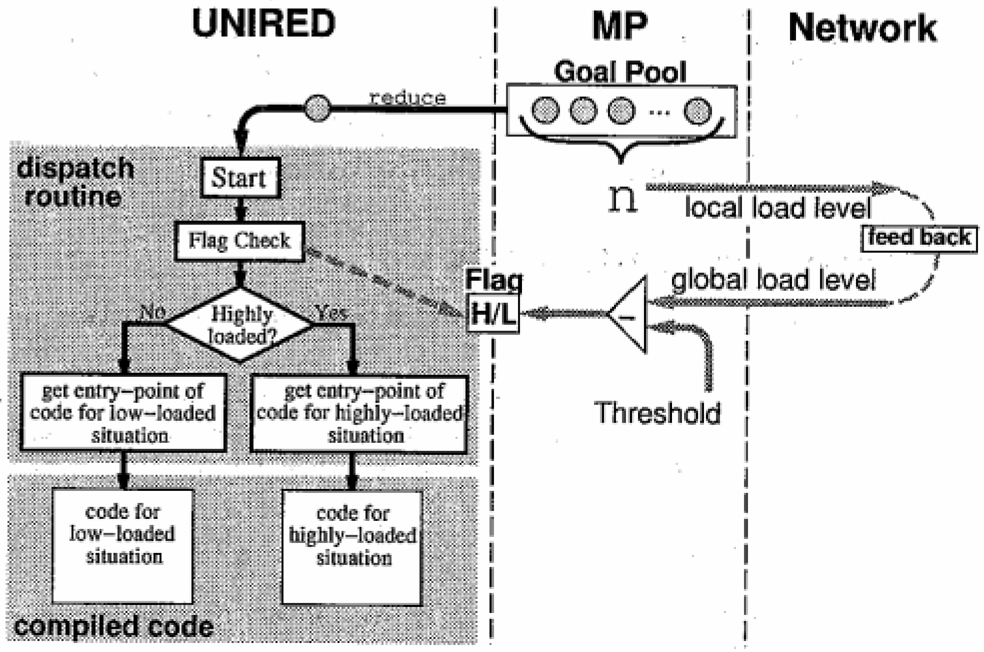

UNIRED
MP
Network
Goal Pool
reduce
dispatch routine
Start
Flag Check
Highly loaded?
No
Yes
get entry-point of code for low-loaded situation
get entry-point of code for highly-loaded situation
code for low-loaded situation
code for highly-loaded situation
compiled code
n
local load level
feed back
global load level
Flag
H/L
Threshold

Figure 9: Mechanism of dynamic load partitioning

6 Preliminary evaluation

We have done preliminary evaluation on PIE64 to know the effectiveness of our load distribution system.

The program we used for evaluation is "primes 3k", which finds all the prime numbers less than 3000. The program was compiled into UNIRED instruction code and executed by UNIRED. Since the compiler available now is a relatively-naive one, we used hand-optimized codes for evaluation. Conditions for measurements and the kind of measured data are listed below. The results of measurements are shown in Figure 10 through 12.

- Conditions for measurements:

  number of IUs used: 4, 16, 64

  whether to perform static load partitioning or not: ON, OFF

  dynamic-partitioning threshold level: 1, 2, 3, 4, 6, 8, 10, 12, 16, 32, 64, 128

- Measured data:

  UNIRED running time: total time that UNIRED was running without stall.

  UNIRED stall time: total time that UNIRED was waiting for a reply from NIP.

  NIP running time: total time that NIP was running.

  Suspension rate: rate of suspension count against reduction count

Each figure consists of two graphs. The graph on the left side shows the results with static partitioning off, while the graph on the right side shows the results with static partitioning on. Thus, the effects of static load partitioning can be known by comparing the two graphs in each figure.

Each graph in the figure shows the results measured by changing the threshold level of dynamic load partitioning, i.e. each graph shows the effects of dynamic load partitioning. If the global load level is greater than or equal to the threshold level, dynamic load distribution is not performed, and new goals and data are held in the local IU. As the threshold level gets lower, dynamic load partitioning is performed aggressively. Thus, setting the threshold level high can be regarded as disabling dynamic load partitioning.

Figure 10 shows the results using 4 IUs.

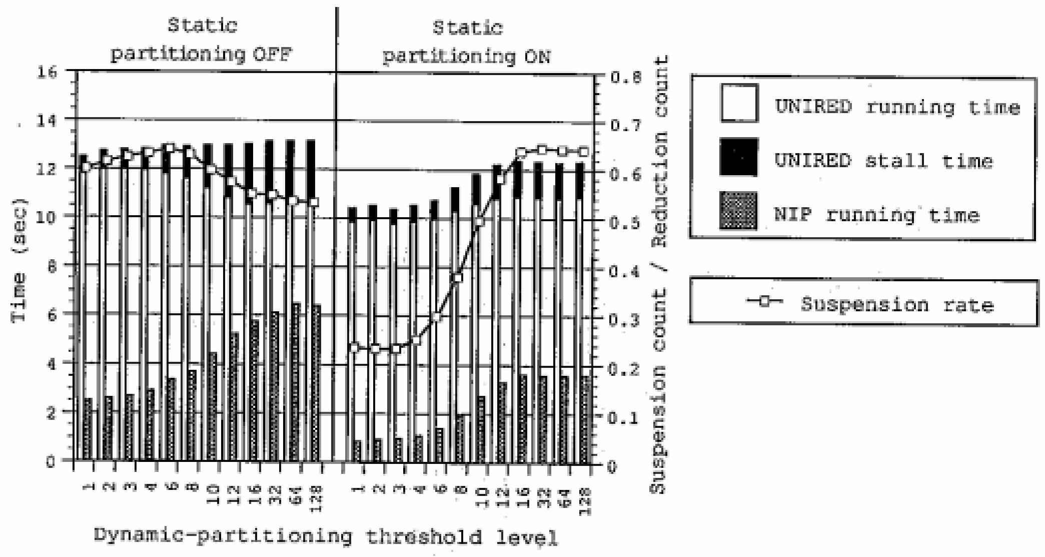


Figure 10: Effects of load partitioning (4 IUs are used)

Comparing the two graphs in the figure, we can see that UNIRED stall time has been reduced by the static partitioning, which means that memory reference locality has been enhanced. The two graphs also show that the running time of NIP has been reduced by static partitioning, which means that communications between IUs have been reduced.

Each of the two graphs in Figure 10 shows that NIP running time and UNIRED stall time gets shorter as the dynamic-partitioning threshold level gets lower. Namely, dynamic partitioning also reduces communications and enhances memory reference locality.

The graph on the left(results without static load partitioning) shows that the running time of UNIRED becomes longer as the dynamic-partitioning threshold level becomes lower. The reason is that the suspension rate gets greater and handling of suspension makes the running time of UNIRED longer.

Figure 11 and 12 show the results using 16 and 64 IUs, respectively.

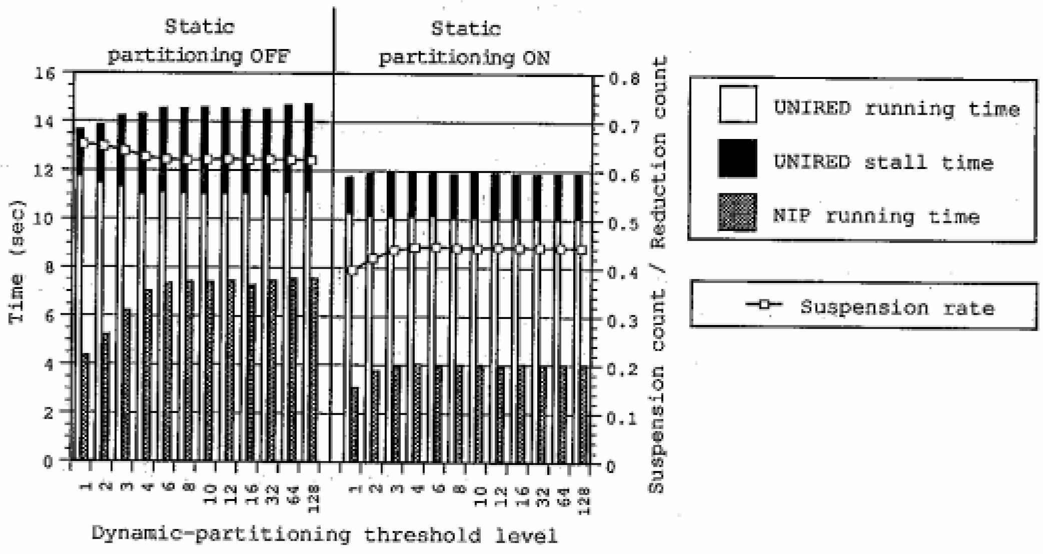


Figure 11: Effects of load partitioning (16 IUs are used)

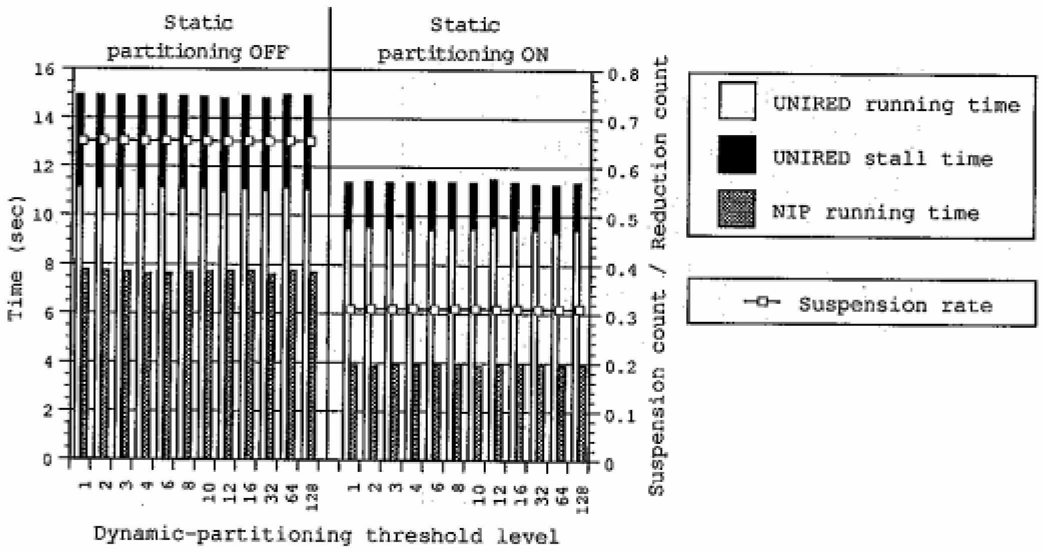


Figure 12: Effects of load partitioning (64 IUs are used)

These results also show that static load partitioning reduces communication and enhances memory reference locality. However, as the number of used IUs increases, the effectiveness of dynamic load partitioning gets smaller. Namely, if the concurrency of the program is insufficient against the number of processors, dynamic load partitioning shows little effect and static load partitioning is important.

While our load distribution system does reduce the running time of UNIRED and NIP, we evaluated only the running time of UNIRED and NIP, and overall execution time was not evaluated. The reason is that, in the current run-time system on PIE64, overhead of managements performed by MP is considerably large, and the overall execution time is dominated mainly by the running time of MP. Thus the improvement in the running time of UNIRED and NIP have little effect on the overall execution time.

However, a new improved run-time system which decreases the overhead will be available before long. With the new run-time system, improvement in the running time of UNIRED and NIP must contribute to the improvement of the overall execution time.

7 Conclusion

This paper presented the overview of the Fleng system on PIE64, focusing on its load distribution system.

Load distribution in PIE64 is performed by combination of static partitioning, dynamic partitioning, and dynamic assignment. The results of preliminary evaluation show that:

- when there is sufficiently-high concurrency against the number of IUs,
  - static partitioning succeeds in reducing communication,
  - dynamic partitioning reduces communication further.
- when concurrency is insufficient against the number of IUs,
  - static partitioning succeeds in reducing communication without losing concurrency,
  - dynamic partitioning shows little effect.

From the results, we can conclude that static partitioning is effective in any situations, especially in low-concurrency situations. On the other hand, dynamic load partitioning is effective in high-concurrency situations, and the combination of static and dynamic load partitioning exhibits the best effects in any situations.

For future work, the following points are important:

- Optimizations to reduce suspension,
- Evaluation of the overall execution time on the new run-time system.

As the result of preliminary evaluation shows, increase in the number of suspension may cancel the effectiveness of load partitioning. Thus, scheduling that causes less suspension becomes important.

Acknowledgements

This work was supported by Grant-in-Aid for Specially Promoted Research of the Ministry of Education, Science and Culture (No.62065002).

References

[1] T. L. Casavant and J. G. Kuhl: "A taxonomy of scheduling in general-purpose distributed computing systems", IEEE Transactions on Software Engineering, **14**, 2, pp. 141–151 (1988).

[2] E. Shapiro: "Systolic Programming: A Paradigm of Parallel Processing", Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 458–470 (1984).

[3] K. Taki: "Parallel Inference Machine PIM", Technical Report ICOT Technical Report TR-0775, Institute for New Generation Computer Technology (1992).

[4] K. Shimada, H. Koike and H. Tanaka: "UNIREDII : The High Performance Inference Processor for the Parallel Inference Machine PIE64", New Generation Computing, Special Issue for FGCS'92, **11**, 3,4, pp. 251–269 (1993).

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

[6] E. Takahashi, H. Koike and H. Tanaka: "A study of a high bandwidth and low latency interconnection network in PIE64", Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 5–8 (1991).

[7] Y. Hidaka, H. Koike and H. Tanaka: "Architecture of parallel management kernel for PIE64", Future Generation Computer Systems, **10**, 1, pp. 29–43 (1994).

[8] Y. Hidaka, H. Koike, J. Tatemura and H. Tanaka: "A Static Load Partitioning Method Based on Execution Profile for Committed-Choice Languages", Proceedings of the 1991 International Symposium on Logic Programming, MIT Press, pp. 470–484 (1991).

[9] H. Nakada, T. Araki, H. Koike and H. Tanaka: "A Fleng Compiler for PIE64", Proceeding of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques '94, pp. 257–266 (1994).