A Static Load Partitioning Method Based on Execution Profile for Committed Choice Languages

Yasuo HIDAKA, Hanpei KOIKE, Jun'ichi TATEMURA and Hidehiko TANAKA
Tanaka Laboratory
Department of Electrical Engineering
Faculty of Engineering
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 JAPAN
{hidaka,koike,tatemura,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract

Load distribution is a key problem for highly parallel computers. Tasks must be so partitioned and be so assigned to processing elements that parallelism is maintained and communication is reduced.

This paper proposes a new load distribution method for Committed Choice Languages. It is flexible enough to make *'automatic' static partitioning* possible. In our approach, the purpose of static partitioning is, "all natural concurrency is preserved, and memory reference locality is enhanced." *Dynamic partitioning* to eliminate excessive concurrency and *dynamic assigning* to balance loads among PEs (processing elements) will be done at run time, and they will be reported in another paper.

We make two proposals, first, that "both goals and data" should have placement tactics and that "the same processing element where an existing related datum is placed" should be a possible tactic. It enables partitioning according to, not only a computation tree, i.e. control dependency, but also according to data dependency.

Second, a new kind of profiler is described. It manages "histories" of an execution and produces enough information to optimize static partitioning.

The method's effectiveness is shown by quantitative evaluation with experimental implementation and qualitative discussion about the contribution in parallel systems.

Keywords: load partitioning, load distribution, committed choice language, execution profile, granularity throttling, non-uniform memory access time.

1 Introduction

Load distribution is a key problem for deriving good performance from highly parallel computers. Load distribution comprises two sub-problems. One is *load partitioning*, how to partition loads such as tasks and data, and the other is *load assigning*, how to assign those divided loads to processing elements (PEs). Bad load distribution causes imbalance of each PE's load and a bottleneck of the inter-PE communication, and fails to get the speedup which is the benefit expected of parallel computers.

The problem of load distribution has been studied by many researchers[2]. There is a graph theoretic approaches in which a program is represented as a task graph and the problem is solved by decomposing the graph and assigning nodes to PEs[4]. There is also a heuristic approach for scientific numerical programs with regular

iterations[1]. However, they are not applicable for Committed Choice Languages (CCLs). The former assumes that the whole task graph has already been obtained, but this is infeasible for CCL since the granularity is very fine and the graph size is enormous. The latter is not applicable because the characteristics of CCL programs are very irregular.

The load distribution method mainly used for CCL is to annotate those points in a program, where load distribution should be performed, with their placement tactics. Placement tactics which have been proposed so far are direct PE numbers, indirect designation on the model of turtle graphics[8], and partitioning fractions on a hypothetical processing power plane[11]. Furthermore, only goals could be annotated with placement tactics. However, these tactics are not so flexible, and a program must be partitioned according to the computation tree or the control dependency between goals. Therefore, unless the original program is specialized for distributed execution, load distribution often fails by excessive concentration or clutter of loads. For moderate load distribution, the programmer must be conscious the number of PEs and rewrite the original program[12]. Furthermore, it is difficult to annotate programs automatically because structural modifications of programs are necessary.

Our approach consists of three stages, i.e. *static load partitioning*, *dynamic load partitioning* and *dynamic load assigning*. Static and dynamic load partitionings make clusters of goals and data which should be placed together in the same PE. While goals and data of the same cluster will be always assigned to the same PE, those of different clusters may be assigned to different PE or eventually to the same PE. Static partitioning is done at compile time and dynamic partitioning is performed during execution. The final assignments of those clusters to PEs are left indefinite until dynamic load assigning decides them during the execution. Load partitioning can be also regarded as granularity throttling.

The purpose of static partitioning is, "all natural concurrency is preserved, and communication is reduced." Dynamic partitioning eliminates excessive concurrency by further clustering and dynamic assigning balances loads among PEs. The dynamic assigning method was reported in [7, 10], and the dynamic partitioning methods will be reported in another paper. This paper concentrates on the static partitioning method.

The principal communications in the CCL on distributed memory machines are remote memory references and goal transfers, but the latency in goal transfer is not so crucial, because the recipient PE doesn't wait for a transferred goal as long as it processes other goals. However, a remote memory reference causes suspension of the execution and affects execution time. Thus we intend static partitioning to enhance memory reference locality as far as the maximum parallelism is maintained.

In this paper, we make two proposals, first, that *both goals and data* should have placement tactics and that *the same PE where an existing related datum is placed* should be a possible tactic. The flexibility of this strategy is such that partitioning according to *data dependency* becomes possible. Therefore, partitioning such as goal and data are sent to the most appropriate PEs can be briefly described.

Second, a method to optimize static partitioning using a new kind of profiler is described. This is a novel attempt to annotate CCL programs with a load partitioning strategy *automatically*. The peculiar point of this method is the profiler which collects not only statistical information but also enough information for later optimization by managing background *histories* of an execution. The histories represent dependence between goals, data and load partitioning tactics. The output from the profiler shows what sort of strategy has much effect on local memory references and parallelism degradation. As a result, the optimizing algorithm after profiling becomes very simple and precise without any heuristics.

This load partitioning strategy and this optimizing method are developed for the

Parallel Inference Engine PIE64[5] and its description language Fleng[6]. While PIE64 is equipped with hardware supporting this method efficiently, similar methods are well applicable to other CCLs and parallel computers.

Fleng and PIE64 are described briefly in section 2 and 3 respectively. The load partitioning strategy is proposed in section 4. The mechanisms of the profiler are described in section 5. The evaluation of the method is described in section 6. The discussion about the importance of the method in parallel systems is described in section 7. Section 8 concludes the paper.

2 Fleng

Fleng[6] is a programming language for fine-grained parallel symbolic processing, and is one of the Committed Choice Languages, or parallel logic programming languages. Guarded Horn Clauses[13] is famous as one of those languages. The principal differences of Fleng from the other CCLs are that there are no guard goals and no implicit AND relations among goals, i.e. failure of a goal doesn't stop the execution of the other goals.

A Fleng program is a set of declarations of horn clauses like:

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

The left side of :- is head part, and the right side is body part. Execution begins when top query goals are given. For each goal, one of the clauses whose head is unifiable with the goal, is selected, and then the goal is reduced to the body part of the clause. Execution is the repetition of this process and it is the unit of parallel processing.

In the implementation of PIE64, the execution of a goal begins with extracting it from a goal queue and dispatching to the compiled code of the clauses by the predicate and the arity of the goal. Then the goal represented as a vector is examined and when it succeeds, new goals are created and inserted to goal queues. During the reduction process, new variables, cons cells, vectors are dynamically allocated on the heap memory. This is one of common implementations of CCLs.

3 Parallel Inference Engine PIE64

PIE64[5] consists of 64 processing elements and two interconnection networks. A processing element of PIE64 is called an Inference Unit, or "IU". This section describes the architectural feature of PIE64 to support the method efficiently.

A notable feature of the interconnection networks[10] is the automatic load balancing facility. This facility is that the network automatically selects the lowest load IU as the destination IU. It performs dynamic assigning, the last part of our three load distribution stages, to balance loads among IUs. Therefore, during the load partitioning stages, a load assignment to IU can be left with an incomplete designation, just the lowest load IU. Where the lowest load IU is designated as the placement tactic, the IU assignment is determined by the network without any overhead at run time.

IU consists of one UNIRED (Unifier-Reducer), four NIPs (Network Interface Processors), one management processor, memory and so on. UNIRED is a dedicated processor for executing Fleng programs. Though PIE64 is a distributed memory machine, it has a single address space, i.e. local and remote memory references are performed by same instruction at different cost. In other words, it is NUMA (non-uniform memory access time) architecture. NIP is a dedicated processor for communication among IUs, and synchronization of Fleng processes.

In the next section, the data placement tactics in other IU's heap memory will be introduced. NIP[9] supports efficiently this remote heap allocation. The remote heap allocation is performed immediately by the NIP on the allocating IU without any handling of the management processor. Furthermore, it doesn't disturb a process of the UNIRED.

4 Load partitioning strategy

In this section, a new load partitioning strategy is proposed. This strategy is simple, flexible and has low overhead at run time.

4.1 Load partitioning points and Load partitioning tactics

A point where load partitioning is possible in the program is called a *load partitioning point.* A load partitioning point is also where an IU must be designated. 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.)

There are several load partitioning points in a clause.

For each load partitioning point, one of the *load partitioning tactics* is selected. The selection of a load partitioning tactic is determined statically, so that the same tactic is invoked for every commitment of the clause. Thus the number of load partitioning points is not proportional to the order of the size of the computation tree, but proportional to the order of the program code size.

We consider the following four kinds of load partitioning tactics:

Tactic A. Select the lowest load IU. (Determined by the network with the automatic load balancing facilities in PIE64.)

Tactic B. Select the local IU. (The IU where the parent goal resides.)

Tactic C. Select the IU pointed to by a pointer (see note 1) obtained as an argument of the parent goal or as an element of a structure in the arguments. (The IU where resides a variable or a structure obtained during the unification.)

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.

The number of load partitioning tactics is proportional to the clause size and it is constant within the same clause. An NUMA machine like PIE64 can implement tactics C and D with little overhead, since the IU number is a portion of address bits and it is easily extracted.

The noteworthy point is that tactics C and D are data oriented tactics, and goals and data are treated symmetrically, both in the load partitioning points and tactics. On the other hand, this kind of load partitioning strategy can be viewed as relative PE addressing.

The following is an example of load partitioning points and tactics in a clause of naive reverse:

```
nreverse([X | L], R) :-
        nreverse(L, R1), append(R1, [X], R).
```

There are four load partitioning points:

1. Selecting an IU where the variable `R1` is allocated.
2. Selecting an IU where the cons cell `[X]` is allocated.
3. Selecting an IU which performs the goal `nreverse(L, R1)`.
4. Selecting an IU which performs the goal `append(R1, [X], R)`.

And there are eight load partitioning tactics:

1. The lowest load IU. (Tactic A.)
2. The local IU. (Tactic B.)
3. The IU where the cons cell `[X|L]` resides. (Tactic C.)
4. The IU where the variable `X` resides. (Tactic C.)
5. The IU where the variable `L` resides. (Tactic C.)
6. The IU where the variable `R` resides. (Tactic C.)
7. The IU where the variable `R1` is allocated by an invocation of tactic A. (Tactic D.)
8. The IU where the cons cell `[X]` is allocated by an invocation of tactic A. (Tactic D.)

A load partitioning strategy is expressed by program annotations in the form '...`@`*something*.' Annotations are placed on the right side of the following: body goals, and those variables and structures which require heap memory allocation, corresponding to load partitioning points; and those variables and structures in a clause head referred to in invocations of load partitioning tactic C. Their meanings are:

- `@any`, `@any(`*id*`)` are invocations of tactic A. The latter is used when its result will be referred to by other invocations of tactic D. A symbol or a number is used for *id* as an identifier.
- `@local` is an invocation of tactic B.
- `@on(`*id*`)` is for a datum referred to by an invocation of tactic C.
- `@to(`*id*`)` is an invocation of tactic C or D which refers to `@on(`*id*`)` or `@any(`*id*`)` with the same *id* in the same clause.

For an example, figure 1 shows a naive reverse program with load partitioning annotations.

```
append([A@on(iu1) | X@on(iu2)], Y, Z@on(iu3)) :-
                Z = [A | Z1@to(iu1)]@to(iu3), append(X, Y, Z1)@to(iu2).
append([], Y, Z) :- Z = Y.

nreverse([X | L@on(iu1)], R) :-
    nreverse(L, R1)@to(iu1), append(R1@local, [X]@local, R)@local.
nreverse([], R) :- R = [].
```

Figure 1: A Naive Reverse program with load partitioning annotations.

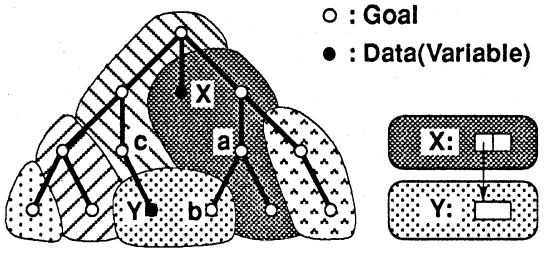

Figure 2: A computation tree partitioned by the data oriented tactics C and D.

4.2 The relation between load partitioning tactics and local memory references

Tactics C and D make it possible to assign those goals and data to the same IU which are created at distant branches in the computation tree. Figure 2 shows an example of such a case. The tree in the figure is the computation tree, i.e. the goal reduction tree, combined with data and variables on the heap memory. Those goals, data and variables on the same background pattern are assigned to the same IU. In this example, goal `b` and datum `Y` are at distant branches. Nevertheless, they are on the same IU. This is derived from the following steps:

1. Goals `a` and `c` have a shared variable `X`.
2. Goal `c` binds `X` with `[Y|Z]`.
   `c( X@on(iu1) ) :- X = [Y | Z]@to(iu1).`
3. Goal `a` waits for `X` to be bound, then spawns goal `b` to the IU pointed to by the address of `Y`, the car part of `X`.
   `a( [Y@on(iu2) | Z] ) :- b(Y)@to(iu2).`
4. Goal `b` reads the contents of `Y` locally.

The noteworthy point is that goal `a` does not require the contents of `Y`, but just the address of `Y`, i.e. `Y` is not dereferenced. The contents of `Y` is accessed at the execution of the spawned goal `b`.

On the other hand, if a variable is bound with a structure, it is necessary for local reference of not only the variable but also the bound structure that the other tactic at the binding side allocates the structure on the same IU as the variable. The tactic in step 2 is this case, and it helps local reference of `[Y|Z]` in step 3.

Here is another example in the previous figure 1. In the first clause , a new goal `append(X,Y,Z1)` is spawned to the `iu2` where its first argument `X` resides so that it can read locally the contents of `X` in the next unification. While `append(X,Y,Z1)` is thrown at the variable `X`, the cons cell bound to it is not guaranteed to be accessed locally. The other binding side tactic is in the same clause but at a different reduction, allocating `[A|Z1]` on the `iu3` where `Z`, previously named as `X`, resides.

5 Profiler for Fleng

The profiler outputs what sort of strategy causes, and how often, local memory references and parallelism degradation. In order to examine these conditions for local memory references and parallelism degradation as precisely as possible, tactics for each load partitioning point are *not fixed* in the profiler, and various possibilities of load partitioning strategies are examined concurrently.

5.1 Labels and Histories of Objects

First, we define several terms about data handled in the profiler. Since a part of this research originated in taking note of the algorithm of ATMS[3], some terms came from ATMS.

- **Object**
  A goal, a structure or a variable, which is generated at a load partitioning point, is called an *object*. In other words, a thing which will be assigned to an IU is an object.
- **Virtual IU**
  For the sake of simplicity, assume that there are an infinite number of IUs and a new IU is always allocated for every invocation of tactic A. The IU allocated by an invocation of tactic A is called a *virtual IU*. Since every load partitioning point has a possibility of invoking tactic A, a new virtual IU is allocated at every execution of load partitioning points.
- **Tactical Assumption**
  A pair of a load partitioning point and a tactic there, is called a *tactical assumption*. It means the assumption that the tactic is taken at the point.
- **Tactical Environment**
  A set of tactical assumptions is called a *tactical environment*. It means a conjunction of tactical assumptions, i.e. assuming all of those tactical assumptions together. A tactical environment never includes two tactical assumptions concerning the same load partitioning point. This is because a load partitioning point never has two different tactics in a regular execution, and if their tactics are same, they can be reduced to one tactical assumption. These situations often occur in recursive execution.
- **History**
  A pair of a tactical environment and a virtual IU is called a *history*. Some histories make up a label (see next) of an object. If the tactical environment of a history is actually the fact, the history shows the virtual IU where the object is.
- **Label**
  A set of histories is called a *label*. It is attached to an object and shows possible virtual IUs where the object may reside and their conditions. Tactical environments of histories of an object are exclusive. Therefore, if tactics of all points are fixed, only one of them becomes the fact and the virtual IU where the object resides is uniquely determined.

Figure 3 is relation between objects, load partitioning points, load partitioning tactics and virtual IUs. Using this figure, we will explain how load partitioning strategies affect the IU where an object resides. O1 - O4 are objects; P1 - P4 are load partitioning points; T1A - T4A are tactics A; T1B, T1C and so on are tactics other than A; IU1 - IU4 are virtual IUs.

Let us consider the virtual IU where O1 resides. If tactics at P1 and P2 are T1B and T2A respectively, O1 is on IU2. If tactics at P1, P3 and P4 are T1C, T3B and T4A respectively, O1 is on IU4. Thus, in order to assign an object to a virtual IU, load partitioning points which are executed *since the allocation of the virtual IU by an invocation of tactic A until the creation of the object*, must have such invocations of tactics B, C or D *as to inherit the virtual IU*. If any of those points invoke other tactics, this object may be assigned to another virtual IU.

Labels and histories of the objects in figure 3 are shown in figure 4. If all invocations of load partitioning tactics are fixed, the virtual IU where the object resides

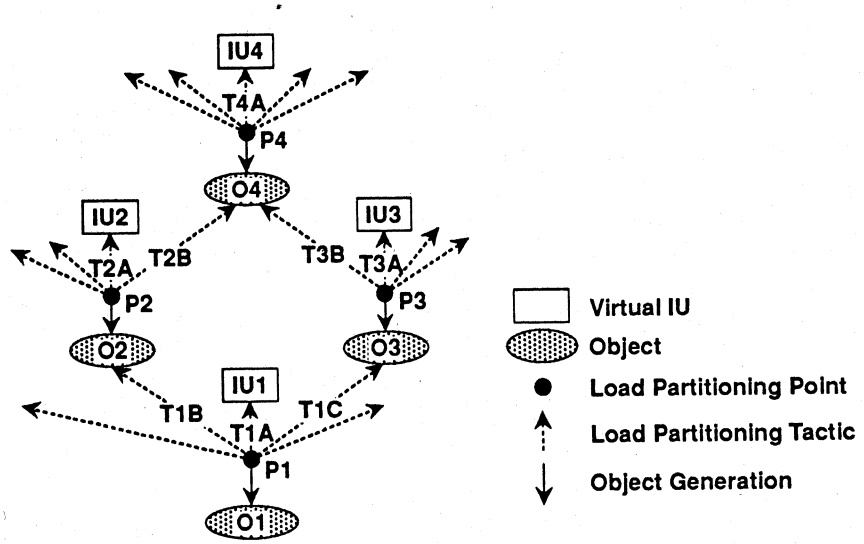

Figure 3: Relation between objects, load partitioning points, load partitioning tactics and virtual IUs.

can be easily discovered from the object's label. On the contrary, the condition that an object reside in a virtual IU can be also derived from the object's label.

Labels of objects are managed by the profiler. At the time of a reduction, labels of all the objects which are generated in the committed clause are created. When an object is collected as garbage, its label can be also collected. Therefore, if the quantity of memory used for a label is limited, the profiler can manage labels with the amount of memory proportional to the amount of alive objects at the same time.

The following gives the outline of the algorithm of generating labels at a goal reduction:

1. Allocate new virtual IUs for each load partitioning point on the committed clause.
2. Retrieve the label of the parent goal and the labels of data obtained from the arguments during the unification.
3. Calculate each label of new objects by synthesizing the above virtual IUs and the labels with the tactical assumptions of the load partitioning point which generates the object.

In step 3, histories with many tactical assumptions are omitted to reduce the required memory and the computation cost. (The validity of this principle is described later.)

5.2 The conditions for local memory references

In order to localize a memory reference during the execution of a goal, it is sufficient that the virtual IU where the referred datum resides is the same virtual IU as that where the executing goal resides. For every time of memory references, the profiler calculates this condition from the labels of the goal and the referred datum, and records and counts it. After the end of execution, the profiler outputs conditions together with the frequency of each. The following is the outline of the algorithm:

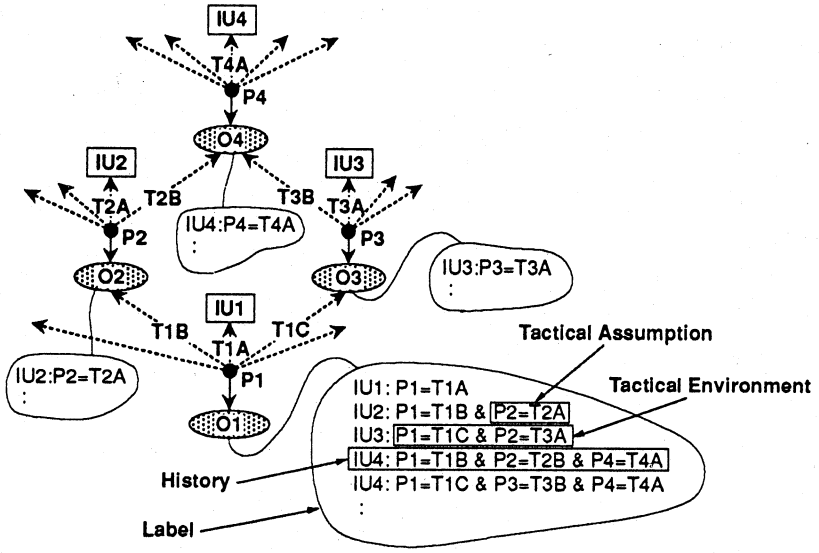

Figure 4: Labels and histories of objects.

1. Find every pair of those histories, one from the label of the goal and the other from the label of the datum, of which the virtual IUs are the same. Virtual IUs won't be used any longer in the algorithm.
2. Merge their tactical environments for each pair of the histories. If they are contradictory, i.e. including same load partitioning point with different tactics, discard the pair.
3. Since tactic A is unnecessary for assigning two objects to same virtual IU, delete the tactical assumption of tactic A from each of tactical environments.
4. Check implications between all the tactical environments. Since the tactical environment of the premise is unnecessary, delete such tactical environments (see note 2). After this step, all the tactical environments are exclusive. (This step won't be performed for the conditions for parallelism degradation.)
5. The obtained set of the tactical environments is the AND-OR condition for the local memory reference. A tactical environment shows conjunction, and a set of tactical environments represents disjunction.
6. Increase counters associated with each of the tactical environments by one (see note 3).
7. When execution is completed, output the recorded tactical environments and their associated counters.

5.3 The conditions for parallelism degradation

Parallelism degradation occurs when some executable goals reside on the same IU.

In order to examine parallelism degradation, goal scheduling is performed in a breadth-first generation-based manner. The goal queue is split into two queues, one for enqueuing and the other for dequeuing. When the dequeuing queue becomes empty, two queues are exchanged. A period between two successive swaps of queues is called a generation.

On the generation transition, the conditions for every pair of active goals to reside on the same virtual IU are examined in a manner similar to that for local memory references, excepting step 4. For the conditions for parallelism degradation, the premise can not be omitted.

The above method treats all parallelism degradation as equal. If even near goals in the computation tree cannot reside in the same IU, it is sometimes impossible to achieve a high rate of local memory references. Those cases occur when many light goals which perform trivial things and terminate soon in a reduction or two, are generated as sibling goals and have shared variables. The above method also separates those sibling goals, and many remote references to shared variables are necessary.

Here, we introduce the *respite time* of parallelism degradation. The conditions on the young virtual IU whose respite time since the allocation has not yet elapsed, are excluded from the conditions of parallelism degradation. Thus light sibling goals can be assigned to the same IU, and many local references to shared variables become possible. Since serious parallelism degradation is mainly caused by heavy goals, which do not terminate in few reductions, most of the parallelism is maintained in spite of introducing the respite time.

5.4 The experimental system

In order to examine the effectiveness of the method, an experimental profiler has been implemented on a unix workstation in advance of the PIE64, which is currently under development. While an existing Fleng interpreter is used as the processing system of Fleng from which the profiler collects data, the profiler itself has the presented algorithm.

The interpreter and the profiler run as separated programs. The trace information which is sent to the profiler through a socket is sufficiently abstracted, e.g. for a memory reference, just two addresses of the goal and the referred datum, for a goal reduction, just the addresses of the committed clause, the parent goal, the given arguments, the new goals and the new data. It is the same trace information as sent in parallel systems. Furthermore, since the goal scheduling of the interpreter is generation-based scheduling, the characteristics of scheduling are similar to parallel systems.

On the other hand, since the profiler itself doesn't assume a particular load partitioning strategy, the profiler's output doesn't depend on the load partitioning strategy at the sample execution in parallel systems. Therefore, even if the strategy is too bad and the sample execution is performed by one processor, the profiler's output is valid. That is to say, the output of the experimental system is valid.

The limitation of the experimental system is that it is based on the interpreter and the characteristics of heap memory references are a little different from those in the compiler based system. Hence, while the optimized strategies for the experimental system are effective for parallel systems with similar characteristics of memory references, the profiling must be done again for systems with different characteristics.

From the above consideration, while the experimental system doesn't always produce valid information for parallel systems, the effectiveness of the method is confirmed by the experimental system.

5.5 Optimization of a load partitioning strategy

The outputs from the profiler are a set of tactical environments and their frequencies which are the conditions for local memory references, and another set of tactical environments which are the conditions for parallelism degradation. As they include enough and precise information, a simple optimization without any heuris-

```
qu(N, A) :- gen(N, L@local)@any, qu(L, [], [], A, [])@any.

gen(N, L) :- gen(true, N, L)@any.

gen(true, N, L) :-
        L = [N | L1@local]@local, sub(N, 1, N1@local)@any,
        gt(N1, 0, R@any(1))@any,  gen(R, N1, L1)@to(1).
gen(false, N, L) :- L = [].

qu([P | Lu@on(1)], Ls, Lp@on(2), A0, A) :-
        append(Lu, Ls, Lr@to(3))@to(3),
        check(P, 1, Lr, Lp, Lp, A0, A1@any(3))@to(2),
        qu(Lu, [P | Ls]@to(1), Lp, A1, A)@to(1).
qu([], [P | L], Lp, A0, A) :- A0 = A.
qu([], [],      Lp, A0, A) :- A0 = [Lp | A]@local.

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

chk(true, _, P, D, L, Lp0, Lp, A0, A) :- A0 = A.
chk(_, true, P, D, L, Lp0, Lp, A0, A) :- A0 = A.
chk(false, false, P, D, L@on(1), Lp0, Lp, A0, A@on(2)) :-
        add(D, 1, D1@to(2))@to(1),
        check(P, D1, L, Lp0, Lp, A0, A)@any.

append([], Y, Z) :- Z = Y.
append([A | X], Y, Z) :- append(X, Y, Z1@any(1))@to(1), Z = [A | Z1]@local.
```

Figure 5: An n-Queens program with an optimized load partitioning strategy.

tics becomes possible. The optimization of a load partitioning strategy is performed in the following manner:

1. Select a load partitioning point to be decided next, from the conditions of the local memory references. Select one likely to affect many local memory references.
2. Select the tactic with the most local memory references, and the fewest other undecided tactics. (This is the reason that histories with many tactical assumptions can be omitted in step 3 of the label generation algorithm described in 5.1 above.) But if it decides any parallelism degradation, do not select it.
3. If there are no such tactics because of causing parallelism degradation, select tactic A for the point.
4. Repeat 1 through 3 until there are no more undecided conditions of local memory references.
5. For the remaining load partitioning points, select tactic B unless it decides any parallelism degradation, or select tactic A otherwise.

Figure 5 is an n-Queens program optimized in this manner.

6 Evaluation

The profiler has another mode for evaluating an optimized load partitioning strategy. In the evaluation mode, the tactics for each load partitioning point are *fixed*

differently from the profiling mode, and the profiler traces just the virtual IU where each object resides.

When a memory reference occurs, the virtual IU of the currently executing goal and that of the referred datum are compared, and if they are identical, this reference must be done locally. Thus frequencies of local memory references and remote memory references are counted. On generation transition, the number of different virtual IUs where active goals reside, is examined as the parallelism of that generation. The average parallelism among generations is calculated for two cases, infinite IUs or limited to 64 IUs. In the latter case, the generation when the degree of parallelism ($= n$) exceeds 64 is replaced by $n/64$ generations of 64 degree parallelism. The limitation of the number of IUs may cause different virtual IUs to be eventually assigned to the same IU by the dynamic load assigning, but those cases are ignored here.

The optimized strategy is compared with the following simple strategy:

- All structures and variables are allocated on the local IU (tactic B).
- One recursive goal is performed on the local IU (tactic B). The other goals are distributed to the lowest load IU (tactic A).

The simple strategy can easily exploit the maximum parallelism.

The profile data are obtained from the execution of Primes-100 and 6-Queens. The parameters of the profiler are as follows: The maximum number of tactical assumptions in a history is three, and the respite time of parallelism degradation is two.

The execution time of the profiler on a SUN4/260 is 235.4 seconds for Primes-100, and 3875.9 seconds for 6-Queens. The optimization is performed manually but routinely using a filtering and sorting tool. The optimized strategy is evaluated not only for Primes-100 and 6-Queens, but also Primes-1000 and 8-Queens.

Table 1 shows the optimization effect on the rate of local memory references. Figure 6 and 7 show the optimization effect on the parallelism of 6-Queens and 8-Queens respectively.

The parallelism degradation may be caused by the respite of parallelism degradation or the omission of histories in the label generation. The goals generated by the eighth clause in figure 5 are sibling goals in the respite time, and they are assigned to two IUs and reduce parallelism in three degree per a commitment of the clause. In the case of 8-Queens, the product of three and the number of commitments of the clause, which is 17204, is 51612. Since the degree of the parallelism degradation in figure 7 is 51897, most of the parallelism degradation is caused by the respite of parallelism degradation. Considering that the simple strategy exploits maximum parallelism and even very light goals are executed in parallel, the parallelism degradation in such degree is acceptable.

Table 1: The optimization effect on the rate of local memory references.

Program	Memory reference count total	Local memory reference count (rate)	
		Simple strategy	Optimal strategy
6-Queens	18278	5650(30.9%)	11696(64.0%)
8-Queens	390916	122305(31.3%)	249520(63.8%)
Primes-100	6471	2555(39.5%)	4674(72.2%)
Primes-1000	225041	82772(36.8%)	161222(71.6%)

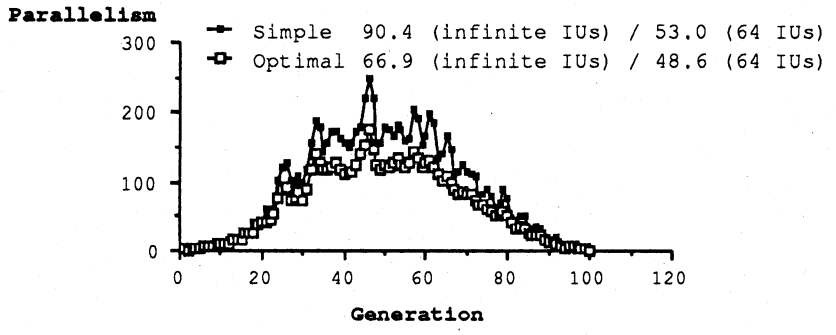

Figure 6: The optimization effect on the parallelism of 6-Queens.

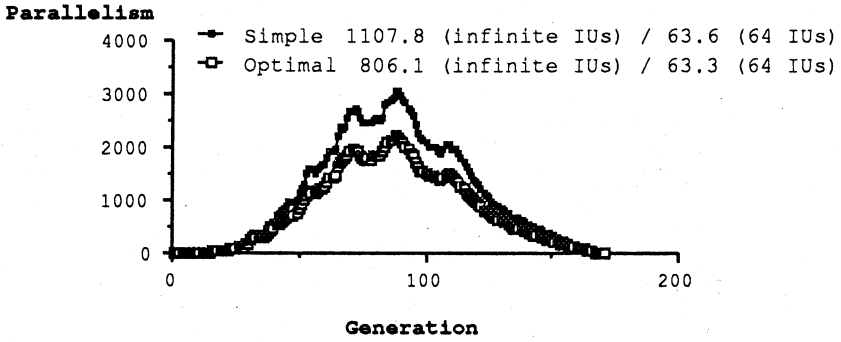

Figure 7: The optimization effect on the parallelism of 8-Queens.

The optimization is based on profile data for 6-Queens and Primes-100, but the optimized strategies are also effective for 8-Queens and Primes-1000. This is because commitment tendency of each clause is similar for varying problem scale.

The results may be summarized as follows: The presented method can increase local memory references without serious parallelism degradation by a factor of two as compared to the simple strategy. Therefore, the method is effective in optimizing a static partitioning strategy.

7 Discussion

While the method improves static partitioning strategies, the purpose of parallel systems is to reduce execution time. Therefore, it is important to discuss relations to other technologies in parallel systems.

The method cooperates fairly well with other technologies. The most related technologies to static partitioning are dynamic partitioning and dynamic assigning. Since static partitioning preserves all natural parallelism and leaves load assignments incomplete, it does not restrict application of dynamic partitioning to eliminate excessive concurrency and dynamic assigning to balance loads. If these methods are adopted, static partitioning can reduce execution time further.

While static partitioning and dynamic partitioning are compatible, they both work in reducing communication. Therefore, the degree of the contribution of static partitioning to reducing execution time depends on whether dynamic partitioning is effective or not.

Although dynamic partitioning works well when the inherent parallelism exceeds the number of PEs to a high degree, it becomes ineffective when the parallelism and the number of PEs are of comparable degree. In contrast with this defect, static partitioning is a technique to discover an effective strategy even if the parallelism and the number of PEs are equal. That is to say, in the field of highly parallel processing where the dynamic partitioning is often ineffective, the static partitioning becomes very significant.

8 Conclusion

In this paper, first, we proposed a very flexible load partitioning strategy, "tactic referring to a datum" and "placement tactic of a datum." Second, we introduced a new kind of "profiler" which manages "histories" of an execution and produces enough information. Then a method to optimize a load partitioning strategy of a Fleng program without heuristics was presented. An experimental system has been implemented on a unix workstation, and the validity of this method has been confirmed. Similar methods are applicable for other CCLs.

The following points are important for future development of the method:

- Development of an efficient profiler using static analysis.
- Consideration about the communication cost of goal transfer and the life length of objects.
- Application of profiling technics to the goal scheduling.
- Evaluation of the method on the PIE64.

Acknowledgements

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

Notes

1: If a variable is used at the corresponding position in the clause head, the unreferenced value should be used. Since this value might not be a pointer, it is necessary to examine whether it is a pointer or not, and to select the local IU if it is not a pointer. On the other hand, if a structure is used at the corresponding position in the clause head, the dereferenced value should be used since dereferencing must be done during unification.

2: In figure 4, a condition to assign both O1 and O2 to IU2 is P1=T1B and P2=T2A, and a condition to assign them to IU4 is P1=T1B, P2=T2B and P4=T4A, and they have no implication. However, after deleting the tactical assumptions of tactic A, they have an implication. Since the consequence (P1=T1B) is sufficient to assign them to same virtual IU, the premise (P1=T1B and P2=T2B) is unnecessary. This is permitted because the condition for local memory references is that of being satisfied. However, the condition for parallelism degradation is that of being avoided, therefore, both the premise and the consequence can not be omitted.

3: The disjunctive relation does not need to be recorded. This is because all the tactical environments are exclusive, and in later optimization, when one of them becomes true, the others certainly becomes false. Therefore, the number of local memory references guaranteed by each tactical environment can be precisely examined, without disjunctive relation. On the other hand, in the conditions for parallelism degradation, the tactical environments are not exclusive. However, the disjunctive relation is unnecessary also in this case, because the conditions will be used to be entirely avoided and it is unnecessary to examine the quantity of the parallelism degradation.

References

[1] Berger, M.J. and Bokhari, S.H.: A Partitioning Strategy for Nonuniform Problems on Multiprocessors, *IEEE Trans. Comput.*, Vol.C-36, No.5, pp.570-580 (1987).

[2] Casavant, T.L. and Kuhl, J.G.: A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems, *IEEE Trans. Software Eng.*, Vol.14, No.2, pp.141-154 (1988).

[3] de Kleer, J.: An Assumption-Based TMS, *Artif. Intell.*, Vol. 28, pp. 127-162 (1986).

[4] Kasahara, H. and Narita, S.: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Trans. Comput.*, Vol.C-33, No.11, pp.1023-1029 (1984).

[5] Koike, H. and Tanaka, H.: Parallel Inference Engine PIE64 (in Japanese), *bit extra number - Parallel Computer Architecture*, Vol.21, No.4, pp.488-497 (1989).

[6] Nilsson, M. and Tanaka, H.: Massively Parallel Implementation of Flat GHC on the Connection Machine, *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pp.1031-1040 (1988).

[7] Sakai, S., Koike, H., Tanaka, H. and Motooka, T.: Interconnection Network with Dynamic Load Balancing Facility (in Japanese), *J. of Information Processing Society of Japan*, Vol.27, No.5, pp.518-524 (1986).

[8] Shapiro, E.: Systolic Programming: A Paradigm of Parallel Processing, *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pp.458-470 (1984).

[9] Shimizu, T., Koike, H. and Tanaka, H.: Inter-PE Communication of the Parallel Inference Machine PIE64 (in Japanese), *SIG Reports on Computer Architecture*, Info. Processing Society of Japan, 79-4 (1989).

[10] Takahashi, E., Koike, H. and Tanaka, H.: A Study of a High Bandwidth and Low Latency Interconnection Network in PIE64, *Proc. of IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp.5-8 (1991).

[11] Takeda, Y., Nakashima, H., Masuda, K., Chikayama, T. and Taki, K.: A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation, *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pp.978-986 (1988).

[12] The Fourth Research Laboratory at ICOT: KL1 Programming introductory course / beginner's course / middle course (in Japanese), Institute for New Generation Computer Technology, Tokyo, p.177 (1989).

[13] Ueda, K.: Guarded Horn Clauses, *ICOT Technical Report TR-003*, Institute for New Generation Computer Technology, Tokyo (1985).