TR- 908

Distributed Pool Pool for Efficient Data Distribution in KL1

M. Yamauchi & M. Satho & T. Chikayama

February, 1995

ICOT Mita Kokusai Bldg. 21F
4-28 1-Chome
Minato-ku Tokyo 108 JAPAN
Tel(03)3456-3191～5

Distributed Pool for Efficient Data Distribution in KL1

Masahiko YAMAUCHI Masaki SATO Takashi CHIKAYAMA
yamauchi@icot.or.jp masaki@icot.or.jp chikayama@icot.or.jp
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

Pool is a data management utility for users of the concurrent logic programming language KL1, providing an updatable table of data including variables. Logic programming languages have single-assignment variables that make concurrent programming easier. On the other hand, updatable tables have to be represented as processes. For programs with a single large table running in a distributed parallel environment, access concentration and delays become problematic. To solve this, we propose a new feature called *a distributed pool* which enables data caching for increased access locality. A cache coherency mechanism implementing the same functionality as the non-distributed version was designed. The proposed mechanism uses split-transactions for higher parallelism and the number of cache states is reduced by a loose data location management policy. The distributed pool has been implemented on the Parallel Inference Machine (PIM). Performance results for its primitives and speed gain with a MGTP (Model Generation Theorem Prover) simulator are reported. Preliminary results of a further optimized implementation on the KLIC implementation of KL1 are also given, which utilizes the object-oriented foreign language interface framework of KLIC, *generic objects*.

Keywords: a concurrent logic programming, cache coherent protocol, distributed pool, split-transactions, loose location management

1 Introduction

Pool is a data management utility which offers users of a concurrent logic programming language KL1 [Ueda 1990] an updatable table of data including variables.

Unlike in procedural languages, variables in logic programming languages have a single-assignment property. This makes writing concurrent programs easier by removing the fear of destroying the contents of variables.

On the other hand, to realize updates without side-effects, updatable tables have to be represented as processes. Pools are represented in this way.

In KL1, a user can write a program to share data between physical processing nodes without explicitly specifying physical data locations. A program with many processes located on different nodes accessing a single pool can be written easily. However, when many processes access a pool managing a single large table at the same time, we will find the following problems.

- Access congestion
  When many processes access a pool at the same time, access performance declines because a pool handles requests sequentially.
- Access delays when transmitting data between nodes
  When processes using a pool are distributed over different nodes and the pool is not located at the same node, a delay in data access becomes apparent.

- Data concentration at one particular node
  When a large amount of data is stored on one node in which a pool is located, a memory shortage may occur on that node.
  A good example is MGTP (Model Generation Theorem Prover) [Hasegawa 1992], a parallel theorem prover based on model-generation. In its current implementation, all the processing nodes have copies of all the data for rapid access. However such a data management policy restricts the size of the model which can be handled by MGTP according to the size of the memory of each node.

There are two ways to solve the problems. One is distributing the data to some predefined nodes. Data locations are somehow maintained and access requests are forwarded by the data location manager. The other method is to manage data distribution more dynamically. Data are cached on local nodes where accesses are made frequently, and may migrate when access patterns change.

The former method must access a remote process each time data are needed. The latter must guarantee that cached data are up-to-date. Taking the communication load into consideration, we chose the latter method. We extended the concept of pool with the caching method and call it a *distributed pool.* To distinguish a distributed pool from a conventional pool, the latter is called a *centralized pool* here.

The next section proposes a new feature called *distributed pool* which solves the problems, followed by a section giving performance evaluation results of its prototyping. Section 4 presents performance improvement ideas and results. Finally we conclude the paper by showing preliminary results when a pool is implemented by a different implementation scheme, *generic objects.*

2 Distributed Pool

2.1 Basic Principles

To solve the problems described in the previous section, we propose a distributed pool. A distributed pool allocates data among several caches processes located on physical processing nodes to maximize the data access locality. As copies of the same data can be cached where they are needed, access congestion and access delays are alleviated.

However, if data between caches becomes inconsistent, concurrent programming will be very difficult, if not impossible. It is important for the distributed pool to behave in the same way as the centralized pool.

When data copies are made without any restrictions, memory shortage may become problematic. To avoid this, a limit on the amount of data cached in a physical node has to be set.

The distributed pool extends the centralized pool with the following two new functions.

1. To cache data distributed to nodes requiring them for decreased access latency and to maintain consistency

   Data updates have the same semantics as a centralized pool with the consistency management mechanism. Any data can be accessed with the same overhead once the data is cached in the local node. This function works very effective where the difference in access speed between the local memory and a remote node is large.

2. To distribute data among many nodes for increased data capacity

   In particular, when we use the Parallel Inference Machine (PIM) [Goto 1989] with 256 processing nodes, the available memory quantity increases by two orders of magnitude.

2.2 Protocol Design

2.2.1 Coherency Protocol

Cache processes may have copies of the same data. If one process modifies its copy, other processes run the risk of reading obsolete data.

To avoid this inconsistency, we have to keep the caches coherent. Generally cache coherent protocols can be categorized into two.

One method allows all the caches to modify data directly. On modification, the new data are broadcast to all cache processes. This method is called the *update* protocol.

The other method allows only one cache to modify data, making all the others read-only. When a cache process is to modify data, all copies of the data in other cache processes are invalidated first. This method is called the *invalidation* protocol.

In the former method, updating messages must be broadcast every time a cache process modifies data. It is usually difficult to implement a system with that protocol efficiently on a large distributed system, because small messages for updating data are sent frequently. The latter method is desirable in our case.

2.2.2 Ownership

When a data item not existing in the local cache process is accessed, it must wait for the data to be transferred from one of the remote cache processes with that specific data. The cache process that takes the responsibility for sending the data is the *owner* of the data.

For data transfer, data location information is necessary. A process called the *directory process* maintains the information keeping track of the data movements (called the data sharing information). The directory process forwards data transfer request messages from cache processes to the process with the ownership of the data.

To prevent performance deterioration caused by congestion in access to the directory process, data indices are distributed to directory processes through hashing.

2.2.3 Replacement

Receiving data from the owner cache process may make the amount of data in the cache exceed the predefined capacity limit. In such a case, the cache process has to make room for the new data by selecting and abandoning some copies.

Since a directory process maintains the data sharing information, if replacement occurs, the cache process must inform the directory process about the data thrown away.

There are two possible ways that the directory process can be notified.

- When the cache process abandons the data
  This makes the cost of replacement itself higher. Thus, it is disadvantageous for applications with frequent replacements.
- When the cache process receives an invalidate message
  This lowers the replacement cost, but the directory process must send redundant invalidation messages to cache processes no longer keeping the data.

We selected the latter because many more replacements may take place than invalidations for data items. The pool may be full of data which are swapped frequently because of the capacity limit, but which are almost never updated.

This means the data sharing information is managed loosely; cache processes with no data might be registered in a directory process.

Table 1: The interface of a pool

message	description
carbon_copy(Key, O)	Returns a value of element with the key 'Key' to a variable 'O'. If no such elements exist, the pool returns [].
put(Key, X, Status)	Adds an element 'X' with the key 'Key'. If the such an element already exists, the pool returns 'replaced' to 'Status'; otherwise, the pool returns 'added' to 'Status'.
get_if_any_and_put(Key, X, Y)	Returns the value of an element with the key 'Key' in the form {Value} to 'X' and registers a new value 'Y' in place of it. If no such elements exist, returns {} to 'X' and adds a new element 'Y'.
get_and_put_if_any(Key, X, Y, Y1)	Returns the value of an element with the key 'Key' in the form {Value} to 'X', registers a new value 'Y', and returns {} to 'Y1'. If no such elements exist, returns {} to 'X', and returns {?Y} to 'Y1'.
get_if_any(Key, X)	Returns the value of an element with the key 'Key' in the form {Value} to 'X'. If no such elements exist, returns {} to 'X'.
remove(Key, Status)	Deletes the element with the key 'Key' and returns 'removed' to 'Status'. If no such elements exist, the pool returns 'non_existent' to 'Status'.
empty(Key, YorN)	Queries whether elements with a key 'Key' are in the pool. If any element with the key 'Key' exist, 'YorN' is instantiated to an atom 'yes'. If no such elements exist, 'YorN' is instantiated to an atom 'no'.

2.2.4 Split Transactions

When the requested data do not exist in the local cache process, it must wait for a response from a directory process. There may be many succeeding requests that can be processed without interaction with directory processes. If the cache process is suspended, waiting for the response to the request for the first message, parallel execution will be unnecessarily restricted.

Hence, when a cache process has to wait for a response from a directory process, we let the process continue transactions after marking the data as *waiting for response*, and suspending only the transaction on that data. Higher parallelism can be obtained by adopting this split transaction policy.

2.3 The Interface of a Pool

The interface that a pool provides for users is summarized in Table 1. In this table, [] means the atom NIL and {} means a null vector.

2.4 State Transition Diagram

We have designed the cache coherent protocol according to guidelines described above.

Figure 1 shows the state transition of a cache process for operations 'carbon_copy' (for data retrieval) and 'get_if_any_and_put' (for updates). Figure 2 shows the corresponding transition of the directory process. A brief descriptions of all the states of cache processes and directory processes are given in Tables 2 and 3.

3 Performance Evaluation of the Distributed pool

We implemented the prototyping system of a distributed pool described in the previous section, and measured the performance on the Parallel Inference Machine (PIM).

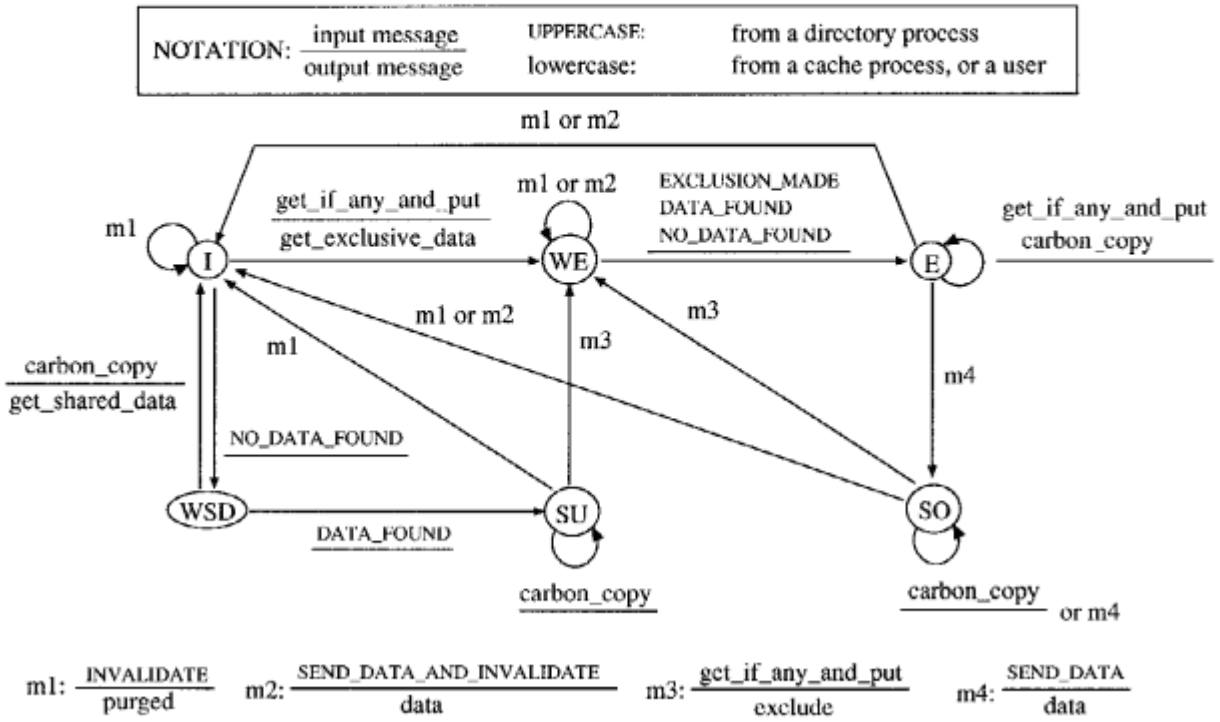

Figure 1: The state transition diagram of a cache process

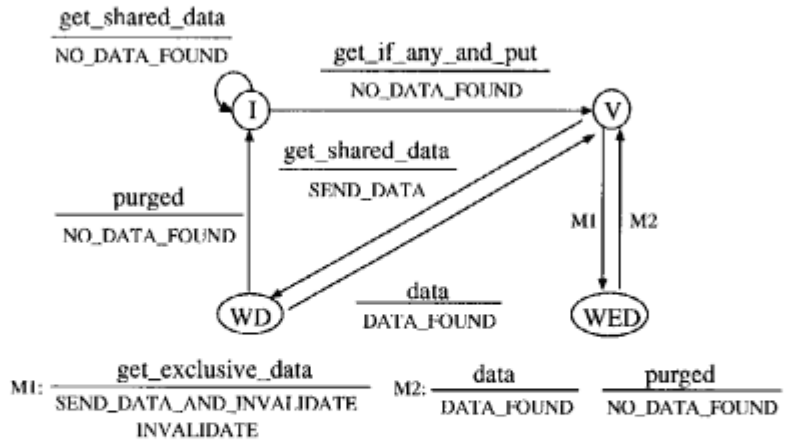

Figure 2: The state transition diagram of a directory process

Table 2: The states of a cache process and their meaning

permanent state	description
I(Invalid)	State in which the cache process doesn't have data
E(Exclusive)	Exclusive state in which no cache processes except for oneself have data
SO(Shared-Owned)	Shared state in which several processes are sharing data. This cache process has to be responsible for supplying the data as requested by other cache processes.
SU(Shared-Unowned)	Shared state in which several processes are sharing data. As replacement occurs in this cache process, the cache process may abandon data.
temporary state	**description**
WE(Waiting-Exclusion)	Waiting to move the E or I state.
WSD (Waiting-Shared-Data)	Waiting to move the SU or I state.
WP(Waiting-Purge)	Waiting to move the I state.

Table 3: The states of a directory process and their meaning

permanent state	description
I(Invalid)	State in which the cache process doesn't have data
V(Valid)	State in which some cache processes might have data. At least the cache process having ownership does contain data.
temporary state	**description**
WD(Waiting-Data)	State in which the directory process waits for data
WED(Waiting-Exclusive-Data)	State in which the directory process waits for all cache processes keeping the data to be invalidated, and then receives the data if it exists.
WEDI (Waiting-Exclusive-Data-If-any)	State in which the directory process waits for all cache processes keeping the data to be invalidated, and then might receive the data.
WE(Waiting-Exclusion)	State in which the directory process waits for a response to the invalidation message, and then moves the V state.
WRD(Waiting-Removed-Data)	State in which the directory process waits for a response to the invalidation message and receives the data if it exists.
WP(Waiting-Purge)	State in which the directory process waits for a response to the invalidation message, and then moves the I state.

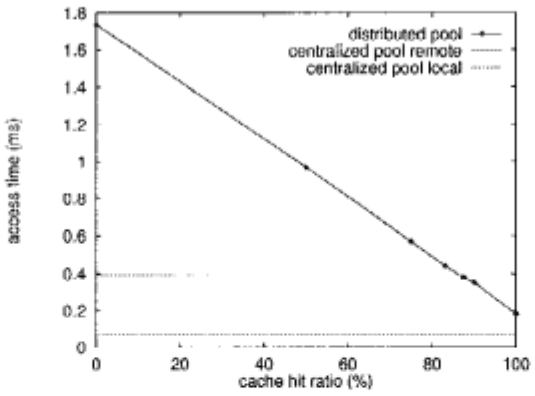

Figure 3: Access time when referring to integer data

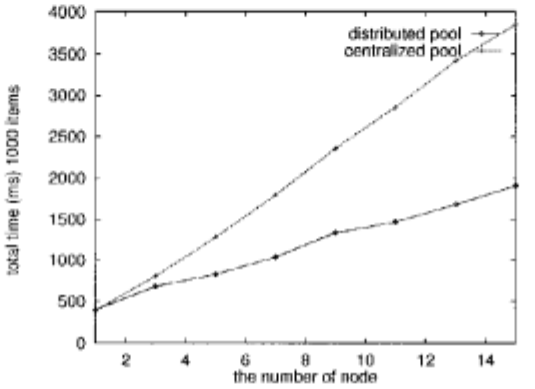

Figure 4: Execution time when access congestion occurs

Figure 3 shows access time for the distributed pool. The access time for referring to integer data was measured. The vertical axis represents the access time, and the horizontal axis represents the hit ratio when accessing data. Local and remote access time for a centralized pool is also shown. Local access means that both a pool and a user process are located on the same node. This figure shows that the performance of a distributed pool is equal to that of remote access of the centralized pool, when the hit ratio is about 85 %.

Figure 4 shows how the computing load is alleviated by a distributed pool. The vertical axis represents the execution time referring to data, and the horizontal axis represents the number of consumer processes. According to this figure, the execution time of a centralized pool increases in direct proportion to the number of consumer processes. On the other hand, the execution time of a distributed pool increases by the ratio of half the centralized pool. One reason for lowering the performance of a distributed pool, is that the processing of sending data is concentrated in the cache process having ownership, when a miss-hit occurs on the data.

Speed up of the *mgtp_bench* program, which is an MGTP simulator, is shown in Figure 5. MGTP is the parallel theorem prover based on model-generation. The task of model generation is to construct a model for a given set of clauses, starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found. The model consists of many atoms which are managed by using a distributed pool in the simulator program.

Many atoms of a model candidate are distributed to many worker processes on each node. Worker

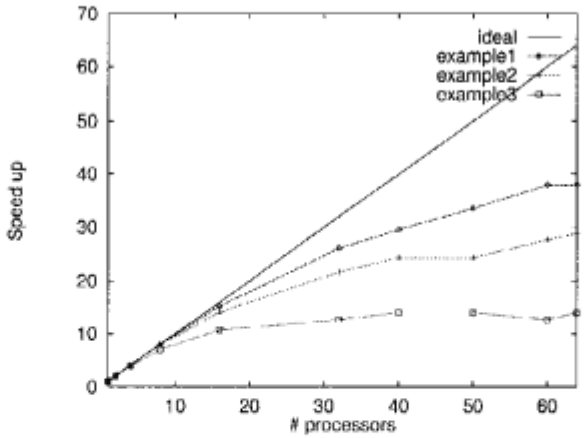

Figure 5: Speed up of the mgtp_bench program

Processes are connected with each others in a ring topology. Referring to all atoms of the model needs to go around the ring.

The sample problems *example1*, *example2*, *example3* have the following model size and execution time by processing in one node.

- example1
  No. of atoms: about 20000, execution time: 1700(sec)
- example2
  No. of atoms: about 10000, execution time: 600(sec)
- example3
  No. of atoms: about 5000, execution time: 140(sec)

4 Performance Improvement of the Distributed Pool

In this section, we will describe some approaches to performance improvement for a distributed pool.

4.1 Simplifying the Cache Coherent Protocol

It is possible to reduce the number of states of the cache process and the directory process by adopting the following policy:

In the prototyping protocol, a cache process needn't inform the data abandonment to the directory process managing the data sharing information, when replacement occurs. Therefore the data sharing information might contain some cache processes no longer keeping data. However we assumed the cache process having ownership should have data at least.

If we allow even cache process keeping ownership not to have data, we can combine the states of WED, WEDI, WE, WRD, and WP into one state.

That is, we can essentially categorize requests to a directory process into only two. These two requests correspond to sharing data and having exclusive data. On receiving a request having exclusive data, a directory process moves to the combined state.

In the protocol of the prototyping system, a directory process moves to the I (Invalid) state, after the WRD or WP states. But in the case of the simplified protocol, it doesn't move to the I (Invalid) state. It continues to incorrectly assume that a cache process has data even if data actually disappears by a retracting operation like get_if_any.

Table 6 summarizes how the simplified protocol decreases the number of states. Also, there was little performance deterioration by adopting this protocol.

Table 4: The states of a cache process and their meaning

permanent state	description
I(Invalid)	State in which the cache process doesn't have data
E(Exclusive)	Exclusive state in which no cache processes except for oneself have data
SO(Shared-Owned)	Shared state in which several processes are sharing data. This cache process has to be responsible for supplying the data being requested by other cache processes.
SU(Shared-Unowned)	Shared state in which several processes are sharing data. As replacement occurs in this cache process, the cache process may abandon data.
temporary state	**description**
W(Waiting)	Waiting for response from a directory process

Table 5: The states of a directory process and their meaning

permanent state	description
I(Invalid)	State in which the cache process doesn't have data
V(Valid)	State in which some cache processes might have data.
temporary	**description**
WD(Waiting-Data)	State in which the directory process waits for data, but there might be no data.
WED(Waiting-Exclusive-Data)	State in which the directory process waits for all cache processes keeping the data to be invalidated, and then might receive the data.

4.2 Caching to the Directory Process

To improve performance in Figure 4, it is effective to divide the task of cache processes keeping ownership. Fortunately for KL1, data movements between nodes is done by a demand-driven policy. So the contents of data aren't transmitted to the node in a directory process, even if we decide to cache the pointer of data in the directory process, because the directory process doesn't refer to the content of the data.

A pointer to data is cached in the directory process only if requesting to refer to data and no pointer is cached. When a cache process requests exclusive data, the cached pointer is discarded. Once a pointer is cached, requests referring to data are treated without interaction with the cache process keeping ownership. Figure 6 shows the improvement in access concentration performance without any degradation of access time.

4.3 Integrating the Centralized Pool

In the prototyping system, a cache process and a directory process use a centralized pool as their data management utility. Yet in this implementation, access time using the centralized pool via a stream (a logical variable) is rather slow. Hence, we integrate the centralized pool within the cache process. Both processes can use the centralized pool via function calls.

With this improvement, the access performance improves as shown in Figure 7. This figure shows that the performance of the distributed pool is equal to that of remote access to the centralized pool, when the hit ratio is about 80 %.

Table 6: Comparison between the two protocols

No. of states	prototype's protocol	simplified protocol
Cache:		
permanent state	4	4
temporary state	3	1
Directory:		
permanent state	2	2
temporary state	6	2

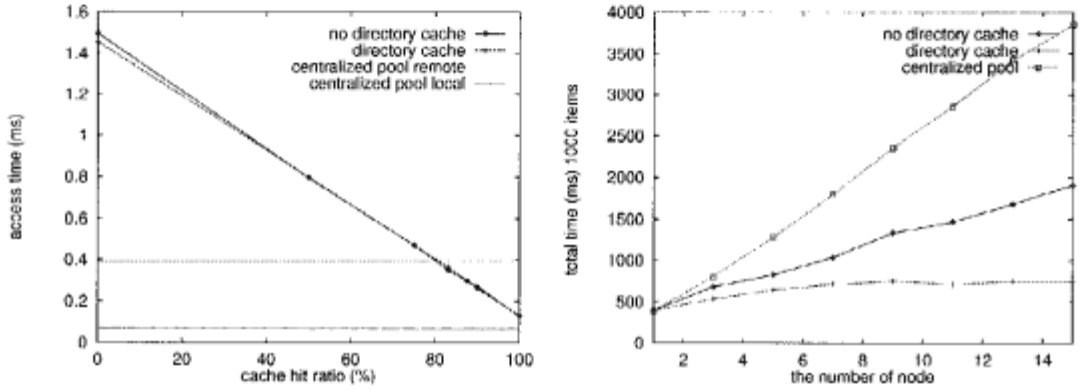

Figure 6: The caching effects in a directory process

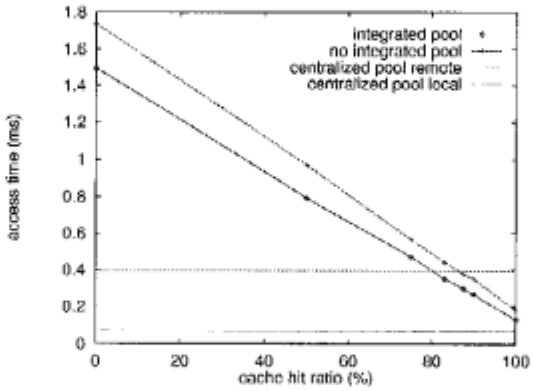

Figure 7: The performance of an integrated pool

Table 7: The performance of a centralized pool using consumer objects

	ratio	execution time (msec)
consumer objects (no suspension)	1	26
consumer objects	1.4	37
KL1 (no suspension)	3.8	99
KL1	4.9	126

5 Preliminary Evaluation of Different Implementation Scheme

KLIC [Chikayama 1994] is a portable implementation of the concurrent logic programming language KL1 by compiling into C. KLIC has a feature called *generic objects* which allows easy modification and extension of the system without changing the core implementation.

We plan to implement a distributed pool using this feature. Before this implementation, the preliminary evaluation of a centralized pool using the feature was investigated.

5.1 Generic Objects

Generic objects provide an object-oriented foreign language interface framework for KLIC. A programmer defines a data structure and operating methods to allow use of generic objects.

KLIC has three kinds of generic objects. These consist of *Data objects,Consumer Objects* and *Generator Objects*. Pool utilizes *Consumer Objects* which are objects with time-dependent states, and are invoked by instantiation of logical variables. They look like a goal process waiting for instantiation.

5.2 Performance of a Centralized Pool

A centralized pool using consumer objects was implemented and its performance was measured on a SparcStation 10. Table 7 shows the results of a task that repeats registration and reference of 1000 integer data. In the table there are items on a centralized pool using consumer object and KL1, and items indicating whether suspension occurs or not.

The table shows that a pool using consumer objects is about 3 times as fast as a pool using KL1. From these results, when a distributed pool using consumer objects is implemented, we can get a gain in performance speed of hit access of 3 times. However, it is difficult to confirm that a distributed pool using consumer objects runs 3 times as fast as one using KL1 in a miss-hit access. Because in the miss-hit access there are several complicated conditions including communication latency between physical nodes.

6 Conclusion

The distributed pool was proposed in order to solve problems of a centralized pool when the application manages a large amount of data using a *pool* and processes them in parallel in a distributed environment.

Using a distributed pool, access contention can be eased, access latency between remote nodes is shortened, and all of the memory can be used efficiently in distributed computing systems, since data are redistributed dynamically according to access patterns.

The cache coherent protocol was carefully designed. It has the following features.

- Adoption of split transactions which allow higher parallelism

- Reduction in the number of states of the protocol by loose management of the data sharing information
- Caching a pointer to data in the directory processes

The performance of a distributed pool implemented by this protocol was evaluated on the Parallel Inference Machine (PIM). We confirmed that the performance of the distributed pool equals that of remote access to a centralized pool when the hit ratio is about 80 % in spite of various overheads for data coherence management. We also found that, with the increase in processes accessing the pool, the execution time with the distributed pool remains almost constants, while that of the centralized pool increases in proportion to the number of accessing processes. An experiment that applies a distributed pool to the MGTP simulator also showed its advantages.

We plan to implement a distributed pool in KLIC, which is a portable version of the concurrent logic programming language KL1. Moreover, we plan to apply the distributed pool to more practical application programs, and evaluate it from the viewpoints of both programming ease and performance.

References

[Chikayama 1994] T. Chikayama, T. Fujise and D. Sekita. A Portable and Efficient Implementation of KL1, In Manuel Hermenegildo and Jaan Penjam (Eds.), Programming Language Implementation and Logic Programming (Proceedings of 6th International Symposium, PLILP'94), pp. 25–39, LNCS 844, Springer-Verlag, Berlin, 1994.

[Goto 1989] A. Goto. Research and Development of the Parallel Inference Machine in FGCS Project. In M. Reeve and S. E. Zenith (Eds.), *Parallel Processing and Artificial Intelligence*, Wiley, Chichester, 1989, pp. 65–96.

[Hagersten 1992] E. Hagersten, A. Landin and S. Haridi. DDM — A Cache-only Memory Architecture. Computer, pp. 44–54, IEEE, September 1992.

[Hasegawa 1992] R. Hasegawa and M. Fujita. Parallel Theorem Provers and Their Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1992, pp. 132–154.

[Nitzberg 1991] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer*, August 1991, pp. 52–60.

[Ueda 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494 500.