Distributed Pool and its Implementation

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

Abstract

PIMOS, an operating system for the parallel inference machine PIM, provides a utility, called *pool* , that manages tables of arbitrary KL1 data. This pool stores any KL1 data and enables it to be extracted and/or referenced when needed according to a search key. This pool is used with many application programs, reducing the cost of program development.

However, its performance is not satisfactory for highly parallel applications, because stored data is located in the memory of the node where the pool was initially created. When its data is accessed from another node, the ensuing data transfer increases network traffic, and simultaneous access causes centralization of the computing load. To overcome this problem, we have developed a distributed pool. The pool is distributed to many nodes and, if a process accesses data existing in another node, the pool caches data in its node. It provides the same functionality as the original pool by maintaining cache coherence. This paper describes the mechanism of this distributed pool.

1 Introduction

PIMOS [Chikayama 1988], the operating system of the parallel inference machine PIM [Goto 1989], provides a comfortable software environment for programming in concurrent logic programming language KL1 [Ueda 1990].

PIMOS provides a utility named *pool*. Pool provides basic in-memory database features, where arbitrary KL1 data are temporarily stored when a program is run, with the necessary data being extracted and referenced according to a search key. The pool feature has been extensively used by most experimental parallel application software systems running on PIM. It has been quite effective in reducing the programming effort required of application programmers.

However, for applications that process a large amounts of data, managing all the data as a unit, pool has performed poorly. Irrespective of the processing node accessing data, they are located in the memory of the node where the pool was initially created. They are then copied between nodes, thus increasing the amount of network traffic. As the PIM has a distributed memory architecture, such centralization of data results in longer latency for data access and a concentration of communication load on specific processing nodes.

There are two other problems. The first is in the nature of the distributed implementation [Rokusawa 1994] of KLIC [Chikayama 1994], for many computers connected through a network. In the current implementation, since interprocessor communication is handled by the operating system, communication cost are very high, making the situation more serious.

The second is that when a program using a large amount of data and centralized data management is executed, the problem size is restricted by the memory limitations of one node. This problem was encountered when running MGTP [Hasegawa 1992] where each node contains the same data.

Given this situation, we have developed a new implementation of the feature where data are distributed to many processing nodes.

Two methods of data distribution may be used.

1. Management of Distributed Data Using Hashing

   In this method, data is assigned a key such that the data is distributed according to the hash value of its search key. While the load concentration can be relieved, the communication frequency and quantity can not be decreased.

2. Management of Distributed Data Using Caches

   In this method, data is managed by keeping a copy of the data made wherever that data is to be used. The load concentration can be relieved and the communication frequency and quantity can also be minimized. However, delicate management of consistency between the copies is required.

We used the latter method with software caching for an efficient distributed pool. The contents of a pool are cached on each node to reduce the cost of access on each node. This distributes memory consumption to many processing nodes, making the maximum capacity of the pool much larger. Section 2 gives an overview of the distributed pool. Section 3 describes the implementation in detail. Section 4 presents an evaluation of our cache protocol.

2 Overview of Distributed Pool

2.1 Cache Protocol

Our caching mechanism is similar to that of the coherent cache memory systems of shared-memory multiprocessor systems [Archibald 1986]. However, some subtle differences exist between the systems. The main features of our cache protocol are as follows.

1. Asynchronous Communication

   A shared-memory multiprocessor system uses a single serialization point as its shared bus, allowing memory transactions to be kept in order[1]. On the other hand, in general, communication between KL1 processes is asynchronous. The implementation of a shared bus as a process can keep messages in order, but it performs poorly from the viewpoint of throughput and prevents natural concurrency. Therefore, we define temporary states between sending a message and receiving a reply. We have to design a state transition scheme containing temporary states. In this point, our system is similar to the DASH [Lenoski 1990] directory-based cache coherence protocol.

2. Nonexistent Back Storage

   In a distributed pool, a cache is not used for small and fast local buffers that mediate fast processors from slow main memory as does hardware cache memory. It is used for caching data, which may be created in the memory of another node, on its own node according to demand for quicker data access, not through the slow network, on the second access. Therefore, the storage structure does not have back storage like main memory; instead a distributed cache on each node constitutes whole storage like in cache only memory architectures(COMA) [Hagersten 1991]. In the event of a cache miss, it is necessary for another cache to supply data instead of back storage. If some caches have the same data, the data in only one of these is set as an *owner* and a cache that has owned data responds to a requirement for data from another cache. When a user tries to store an amount of data exceeding user-defined cache capacity, data has to be removed from the cache to reclaim memory space. Data, except for owned data, has to be selected for replacement because at least one of the data with the same key must be left in some cache.

3. User Message Interfaces

   Besides messages enabling easy handling or storing data, we have a variety of ways to access data stored in the pool. For example, some messages replace data only when data with the same key exists. Others place new data regardless of whether or not data with the same key exists. This complicates our cache coherent mechanism further.

4. Cache Line

   In hardware cache memory systems, selection of the cache line size is one of the central design issues. Cache memories are accessed using data addresses as keys and data with adjacent addresses are likely to be accessed in a short period of time. Thus, we can create a prefetching effect by making the cache line size greater than one. On the other hand, data in pools are accessed by keys arbitrarily chosen by the user: integer, atom, string or structured data. In general, we cannot expect any access localities associated with similarities of keys. Thus, we made the cache line size to be one; that is, we abandoned the idea of grouping data with similar keys.

5. Write-Invalidate Type Cache Consistency

   There are two ways to keep the cache consistency: write-invalidate and write-update. Our coherent protocol introduces the write-invalidate type to decrease the amount of data sharing as in most coherent cache memory systems.

2.2 Components of Distributed Pool

Figure 1 shows two principal kinds of processes that constitute a distributed pool.

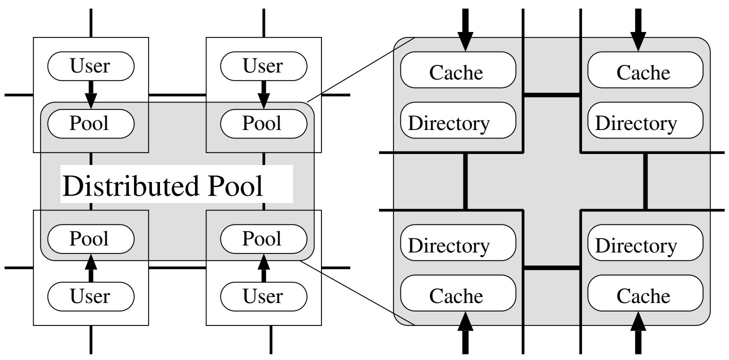

Figure 1: Process Constitution

[1]The weak consistency model used on split bus implementations is not acceptable for ease of programming in our case.

- Cache Processes

  Cache processes keep data in a hash table and manage their four permanent states and seven temporary states. The permanent states are classified according to exclusivity and the responsibility for data supply. The temporary states take care of crossed messages.

- Directory Processes

  Directory processes keep no data but manage cached data locations with lists of caching nodes. Such a list is called a *cache list* . The head element of a cache list shows the owner. The owner is responsible for providing data for cache misses, and is not selected for replacement. According to the cache list, directory processes transfer messages between caches. The directory processes have two permanent states classified according to the state of data validity, and six temporary states. To avoid access centralization, directory processes are located in each node unless otherwise designated. The managed set of data is decided according to search keys.

2.3 Information Located in a Cache

The currently available KL1 implementations employ lazy data transfer policies. The actual data is not moved only by moving pointers. Thus, we cannot be sure where the data exists. In fact, a cache may not keep real data, only a pointer. Nevertheless, managing the pointer locations can be effective because a user process that obtains a pointer from a pool usually reads this data that causes data migration.

3 Implementation

3.1 User Interfaces

Principal messages to pools for users are as follows.

carbon_copy(Key, O) : copies an element.
put(Key, X, Status) : adds an element.
get_if_any_and_put(Key, X, Y) : extracts an element, if any, and then stores a new element with the same key.
get_and_put_if_any(Key, X, Y, Y1) : extracts an element, if any, and then stores a new element with the same key. If there is no corresponding element, the new data is not stored in the pool either.
get_if_any(Key, X) : extracts an element, if any, with the given key.
get_all(Key, O) : extracts all the elements with the key.
remove(Key, Status) : deletes an element.

3.2 Cache States

Cached data can take one of the following states.

Invalid(I) : the cache does not contain this data.
Exclusive(E) : no other cache contains this data.
Shared-Owned(SO) : other caches possibly contain this data, but this cache is responsible for supplying the data and continues to keep it.
Shared-Unowned(SU) : at least one other cache contains this data and this cache does not take any responsibility for keeping it.
Waiting-Exclusion(WE) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state.
Waiting-Exclusion-If-any(WEI) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state. If the data is invalidated, because of crossed messsages, and no directory sends back the data, later it enters the I state.
Waiting-Shared-Data(WSD) : the cache does not contain this data and is waiting for the data to be transferred. If other caches contain this data, it will be shared after the data arrives.
Waiting-Purge(WP) : the cache contains this data and is waiting for invalidation of the same data on all other caches, after which it enters the I state.
Waiting-Exclusion-without-Data(WED) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state.
Waiting-Exclusion-If-any-without-Data(WEID) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the E state. If no directory sends back the data, later it enters the I state.
Waiting-Purge-without-Data(WPD) : the cache does not contain this data and is waiting for invalidation of the same data on all other caches, after which it enters the I state.

3.3 Directory States

The directory manages two permanent and six temporary data states.

Invalid(I) : no cache contains this data.
Valid(V) : some caches contain this data.
Waiting-Data(WD) : waiting for the data to be sent from a cache.
Waiting-Exclusive-Data(WED) : waiting for the data for exclusive access. Later it enters the V state.
Waiting-Exclusive-Data-If-any(WEDI) : waiting for the data for exclusive access. If no cache sends back the data, because of crossed messages, later it enters the