Implementing Tuple Space With Threads

João Carreira, João Gabriel Silva
Dept. Engenharia Informática,
Universidade de Coimbra, Portugal
{jcar,jgabriel}@dei.uc.pt

Koen Langendoen, Henri Bal
Dept. of Mathematics and Computer Science
VU, Amsterdam, The Netherlands
{koen, bal}@cs.vu.nl

Keywords: Parallel Programming, Open Implementation, Tuple Space, Optimization

Abstract

The development of efficient and portable parallel programming systems can be a complex and troublesome task. Although there are several portable environments that are meant to be used as a support layer for higher level programming systems, they all provide different features and different levels of functionality to the system programmer. In this paper we report on our experience implementing a tuple space library, called OpenTS, on top of two portable environments: the MPI message passing interface and the Panda virtual machine. We discuss the difficulties encountered when implementing OpenTS on both systems, and how the availability/lack of some features affected the flexibility, elegance, and refinement of the overall tuple space implementation. In particular, we focus on the system support that is required to build the sophisticated OpenTS run-time system module that supports different tuple distribution policies based on programmer annotations. Experimental results of OpenTS collected on a cluster of workstations using Panda and MPI are presented. The comparison of these performance numbers shows the practical impact of both environments on the overall performance of a higher level programming system.

1. Introduction

Parallel programming using message passing environments such as MPI [1] and PVM [2] are the de facto standard for programming multicomputers and networks of workstations. The main reasons for the widespread acceptance of this programming model are performance and portability. In fact, they do not require any special hardware, compiler, or operating system support and they are sufficiently low level to guarantee good performance on general applications with a medium level of complexity. There are, however, many algorithms for which the message passing implementation is non-trivial, tedious, and error-prone. For these cases the choice is to use a higher level programming model, so problems can be handled at a higher abstraction level and the programmer is freed from the burden of dealing with the complexity of message passing.

The problem that is usually posed to the developers of high level parallel programming systems, is how to achieve portability without compromising the flexibility and efficiency of the implementation. One popular approach is to build the high-level system as a parallel library on top of the standard message passing environments MPI and PVM. This solution is adequate for systems that support classes of recognized, delimited, and well-known problems such as task farming, domain decomposition, and numerical analysis. In these systems, common structural elements of the problems are encapsulated within the library, so they can be reused in several programs. These libraries range from general numerical packages such as [3], which provide previously parallelized mathematical functions, to more complex libraries that encapsulate parallel programming paradigm specific requirements. In the last case, libraries are said to be based on the skeleton [4] or template principle. Examples of such libraries are the PUL libraries [5], DSM packages like ADSMITH [6] and DSMPI [7], and Linda-like environments such as P4-Linda [8] and MPI-TS [9]. The library-based approach is supported, e.g. in MPI, by incorporating special features specially focused on the development of parallel high level libraries [5] such as message contexts, process groups, and virtual topologies. In fact, MPI was designed to be a common framework for parallel application development, targeted both to the end user using explicit message passing and to the system's programmer as a support layer for building higher level systems.

An important unanswered question, however, is whether or not user-level message-passing environments such as MPI can provide the support required to develop sophisticated high-level parallel programming systems in a flexible and efficient way. For example, systems that manage distributed data such as Distributed Shared Memory, Tuple Space, and matrix solvers need to transparently distribute, replicate, and/or migrate data throughout the system. This can require more support than a user level-message passing-package, such as MPI, can (and should) provide given its efficiency requirements. Most of the research groups currently involved in the development of high-level parallel systems end up building their own enhanced message-passing system to guarantee portability, flexibility, and efficiency of
their implementations. Examples of this approach are the Converse portable parallel programming framework [10] used to implement CHARM++ [11], the PORTS consortium [12] runtime system, Panda [13] used in Orca [14], and the Nexus library [15] used to develop CC++ and Globus [16].

In this paper we report on our experience implementing a tuple space library, called OpenTS, on top of two portability environments with different philosophies: the MPI message passing interface and the Panda virtual machine. The paper is organised as follows: In Section 2 we briefly present our prototype tuple space implementation. Section 3 describes the two support layers used to implement OpenTS: Panda and MPI. In Section 4 we discuss the implementation strategies, advantages, and drawbacks of each approach and Section 5 presents performance results. Finally, Section 6 concludes the paper.

2. Tuple Space

The Linda parallel programming paradigm [17] is a simple and elegant model for parallel programming. It is based on the notion of the Tuple Space (TS), a logically shared memory with the logical tuple as the basic data unit. We will not describe Linda in detail here, the interested reader is referred to [18]. We will focus instead on the OpenTS novel optimisation techniques.

2.1 Open Implementation

OpenTS is based in the concept of Open Implementation [19] where critical system implementation decisions are exposed to the user in a clear and elegant way without compromising the modularity underlying the black-box concept. An open implementation presents two interfaces: the primary interface that provides the basic functionality and the meta-interface that allows the user to adjust the implementation strategy decisions that underlay the primary interface (see Figure 1). OpenTS provides a primary interface with the basic tuple space access functionality and a separate meta-interface that allows the user to select execution models, to adjust tuple distribution/access policies, to control process mapping, control actions to be taken in the case of distributed deadlock, and to select adequate termination modes. The primary interface of OpenTS provides the five well-known primitives (in out, rd, inp, rdp), which can be used to access tuples of the form (tag, data) in the Tuple Space. OpenTS also provides the eval primitive, which is used to launch a process or thread on a remote processor.

FIGURE 1. OPEN IMPLEMENTATION

This interface per se is enough to write user applications. The meta-interface can be used by advanced programmers to tune the implementation policies according to their needs. Basically, the meta-interface provides means to transfer the programmer’s knowledge of the application to the run-time system improving performance without compromising portability. Optimisations can be then applied incrementally, using simple mechanisms provided by the system itself, which do not make the code dependent on a particular architecture, significantly alter the program structure, or make the code unreadable.

OpenTS was built as a library of C functions on top of Panda and MPI. No compiler or pre-processor is involved. Therefore explicit initialisation and termination functions are needed. Currently, OpenTS can be regarded as a system with a similar design philosophy as the CRL DSM package [20] or DSMPI [7] where shared regions are addressed by global tags and shared among all processes. Of course, the shared memory is accessed in a fundamentally different way in both systems and synchronisation is intrinsic to the OpenTS primitives.

2.2 System Architecture

An OpenTS application consists of a single program image, which is launched simultaneously in a set of predefined processors. At startup the application invokes InitTS() to initialise the OpenTS environment and to decide the role of each process in the system.

The organisation of processes in OpenTS follows the client-server model. Processes capable of storing tuples are called TS servers and handle requests from TS clients. Additionally, each client is provided a local tuple cache to use in special cases as will be shown later. One of the servers in the system plays a special role, and is called the central server. The additional task of this server, besides storing tuples, is to keep a consistent global state of the system and forward eval requests to the most appropriate processors according to configurable rules.

In Tuple Space applications one of the client processes is called the master process. Of all the processes launched during the startup phase, this is the only one in which control is returned to the user.
application. The other processes remain idle waiting to execute user process(es)/thread(s) upon request from the master or other processes. This is the most natural model for programming the Tuple Space: with a primary process that dynamically creates all the others using \textit{eval}. However, some programmers (e.g. those familiar with MPI) are not used to create processes dynamically and prefer the SPMD (Single Program Multiple Data) approach. In this model, all the processes initially launched return control to the user application and the separation of tasks is made explicitly by the programmer. OpenTS provides a way to select the most appropriate model through the SetParadigm() meta-interface primitive. The OpenTS environment is terminated either explicitly by the user or automatically by the runtime system using a distributed program termination algorithm discussed in detail in Section 4.

2.3 Programmer Hints

The OpenTS meta-interface can be used to adjust the implementation policies of several aspects of the system. The \texttt{Hint()} meta-interface primitive is of primary importance for optimization and will be discussed in detail. Hints provide a way for the programmer to transfer knowledge to the run-time system about expected patterns of tuple usage within the application. Typical patterns supported by OpenTS are: read many, producer-consumer, and write many tuples.

Read-many tuples refer to tuples that are read far more often than written. This is the case for general counters or reference values. Read-many tuples can be identified as those tuples that are most accessed by \texttt{rd} requests. These tuples are distributed through the servers using an hash function and a copy is kept in the local cache of clients that declared themselves as readers using \texttt{Hint(tag, READ_MANY)}. The consistency of the copies in the client’s cache is maintained using an invalidate and update protocol: when the tuple is retrieved from the server (using \texttt{in}) an \texttt{invalidate} is sent to the clients, when it is deposited in the server (using \texttt{out}) an \texttt{update} message is sent instead. If the user only needs weak consistency, \texttt{Hint(tag, READ_MANY \& WEAK)} can be used and the run-time system will only send update messages.

Producer and Consumer tuples are by far one of the most common in Tuple Space applications. They are generated by producers and withdrawn by consumers in a fixed pattern. This is, for instance, the case for job tuples in the master-worker paradigm. They are handled in OpenTS by performing eager tuple movement, i.e. migrating tuples in advance to the clients where they will be consumed. The clients (workers) willing to consume a specific kind of tuple should invoke - \texttt{Hint(jobtag, CONSUMER)} - while the clients (possibly the master) producing tuples should invoke - \texttt{Hint(jobtag, PRODUCER)}. Note that this hint can also be used to classify other, perhaps less evident, tuple access patterns such as the result tuples in master worker applications and the intermediate results of pipelines. In the first case, the master should declare itself as a consumer and the workers as producers. In the second case, each client in a stage of a pipeline declares itself as a consumer from the client in the previous stage and as a producer to the client in the next stage.

Write-many tuples are frequently modified by multiple clients. This kind of objects is problematic in any parallel system and the best solution is to keep them centralised in a certain server. Clients address servers to access tuples in this class by using a simple hashing function named \texttt{\sigma} that computes the server \textit{id} from the number of servers and the tuple \textit{tag}. Note that this policy would be very inefficient \textit{per se} due to possible collisions in the hashed values. Furthermore, this simple policy would not solve the problem of single servers overloaded with requests for a specific tuple. These problems are not serious in OpenTS because this distribution policy is just used for the subset of tuples for which there is no specific access pattern to exploit.

3. Panda & MPI

In this section we will show the differences between MPI and Panda in terms of what kind of support those environments provide for implementing run-time systems. MPI is the accepted standard for message passing and is available on a wide range of systems such as UNIX workstation clusters, Microsoft Windows NT micros, and the Cray-T3D. It provides several features especially oriented to support the development of parallel libraries. Contexts isolate different message-passing modules, possibly from different sources to avoid interferences and side-effects resulting from module integration. Virtual topologies provide means for algorithms that rely on special processor topologies to be hardware independent. Collective communications and process groups enable the execution of operations involving specific groups of processes. On the other hand MPI does not provide integrated communication and multithreading, and broadcast operations. To implement OpenTS, we used the MPIch public domain package from the Argonne National Laboratory.

Panda is a portability layer that offers a common interface to threads and communication on many different hardware/software platforms. Although Panda was originally designed as an efficient virtual machine for the Orca runtime system, it has since been used to implement other parallel programming systems (e.g., PVM and SR) on top several operating
systems (including Amoeba, Solaris, and AIX) and machines (including clusters of workstations, the IBM SP2, and CM-5). The Panda interface includes primitives for threads, synchronization, and communication and is designed to be portable and efficient. An important aspect of Panda is that communication and multithreading are cleanly integrated. All incoming messages are handled in the context of a separate thread. In contrast to MPI, messages need not be received explicitly since Panda will automatically invoke a user function through an upcall to process the incoming message.

4. Implementation

In this section we describe and compare the OpenTS prototype implementation strategies on MPI and Panda. This comparison will focus on design issues, and points out the relative advantages and drawbacks of both systems. An experimental assessment and performance study will be presented later in Section 5. The differences among both implementations can be summarised in the following categories: data distribution, process mapping, and distributed program termination. We discuss each of this issues in detail.

4.1 Data Distribution

The implementation of the data distribution policies differ substantially in both systems. Concerning read-many tuples, the update and invalidate protocols described in Section 2 were implemented in Panda in a clean and elegant manner using threads: The arrival of update/invalidate messages causes the creation of a thread (upcall) in the destination process to handle the message and perform the update or invalidate operation immediately. In MPI, on the other hand, update and invalidate messages are sent using the synchronous immediate send primitive - MPI_Issend() - which is basically a non-blocking send that terminates when a receive is issued in the destination - MPI_Recv(). This involves a careful management of handles associated with each MPI_Issend() issued, and the usage of a probe primitive such as MPI_Test() to check for send completion. As a consequence, these runtime system messages will only be received when the library gains control over the destination process. This eventually happens when the user application invokes an OpenTS primitive. Thus, while message delivery in Panda only depends on network latency, in MPI it is highly dependent on the application’s behaviour.

Producer and consumer tuple access classes are also handled differently in both systems. As explained in Section 2, producers attempt to send the tuples produced directly to the consumer before the consumer issues a specific request. In MPI, this is accomplished using synchronous immediate sends - MPI_Issend() - just as with read-many tuples. However, this scheme only works if messages are buffered at the receiver side, so that the tuple is effectively waiting to be retrieved at the destination node. If the sender buffers the messages, then this scheme does not optimise tuple movements. Of course, the side where messages are buffered is an internal implementation decision of MPI that is naturally hidden from the user (because MPI itself is not an Open Implementation). In addition, using MPI, the producers cannot keep the produced tuples locally waiting for an opportunity to send them to the consumers. This is because producers are single-threaded and since the library loses control of the producer process to the user application, no assumption can be made about when the application will invoke the library again. In OpenTS, we circumvent this problem by sending consumer tuples to the server, which is a separate process that just runs OpenTS code, who can probe its outstanding non-blocking sends regularly and keep the consumers busy by sending tuples. In Panda, the upcall model solves these problems in a clean manner: the tuples are sent by the producers to the consumers and immediately received and stored in the local cache of the consumer by a dedicated upcall thread. A control-flow protocol keeps the number of tuples received in advance at a constant level and avoids consumer unbalance.

To summarise, implementing OpenTS data distribution schemes in MPI required us to rely on several dirty tricks and sometimes to assume certain MPI undocumented implementation policies. With Panda, on the other hand, the implementation became naturally elegant and flexible.

4.2 Process Mapping

In large heterogeneous distributed environments, the differences in terms of network bandwidth, communication latency, availability, CPU performance, and load among the distinct components can severely unbalance applications and affect overall performance. In OpenTS, the programmer can tune implementation decisions concerning process-to-processor mapping through the meta-interface. Configurable decisions are: the location of the OpenTS central server, location of master process, and the policy to use when launching new user processes (threads) through eval. Depending on the role of the master within the application, the programmer can map the master on a separate process/node, or let it coexist in the same process with a worker or server. For instance, if the master is idle most of the time waiting for the workers to produce results, there is no sense in keeping it on a dedicated processor. The central
server tasks can also be assigned to any server in the system according to the user's needs. Finally, to decide where to dynamically launch eval processes, OpenTS can take into account several characteristics of the application: the location of the central server and master, the number of stored tuples and the frequency of tuple access in each server, and the possible coexistence of multiple processes in the same host. This information is managed and kept up-to-date by the central server, which regularly collects statistical data from the other processes. The set of criteria's described can be combined by the programmer and provided to the runtime system using the SetEvalLaunchPolicy() meta-interface primitive. If the programmer needs even more control over process mapping, the system can accept a configuration file, called a cluster map. This map contains the layout of the cluster and the desired mapping of clients and servers to machine names. The configuration file is external to the application, so the runtime system will adopt the default policies if the file is not provided for some reason.

The absence of threads in MPI has also a serious consequence. As processes in MPI are defined statically, the run-time system has to emulate dynamic process creation in order to handle evals. Of course, in practice there are no real processes being created; the eval request is simply forwarded to an idle client process that acts as an eval server. With MPI, each of these single-threaded processes can handle exactly one eval. Therefore, the number of evals that can be issued is limited to the overall number of MPI processes. If the total number of evals exceeds this limit, the runtime system freezes subsequent evals in a wait queue until a process becomes free. With Panda, on the other hand, the runtime system launches new threads in remote processes to handle evals and therefore there is no limit for the number of evals (except if the programmer specifies a threshold using SetMultiThreadLevel()).

4.3 Distributed Termination

In distributed computing a program can be considered as terminated when all processes are blocked waiting for communication from each other. In this case the program is said to be deadlocked. For example, in divide-and-conquer applications a master process generates an initial set of tasks that workers (executing an infinite loop) use to produce results and generate more tasks. The process responsible for collecting results (usually the master itself), doesn’t know a priori how many results will be generated by the workers for the problem being computed, and thus doesn’t know when it has already collected “all” the results and should terminate. In this example the master process lacks information to correctly terminate the application. It needs to know that the worker processes are all blocked waiting for tasks, and therefore cannot generate more tasks or results.

The OpenTS run-time system for Panda implements a distributed deadlock detection algorithm to solve this classical distributed termination problem and also to assist in application debugging. The algorithm uses active monitoring [21], and a modified snapshot protocol: the central server collects regularly the local states of the other processes and combines them in a consistent global state in order to reason about the existence of a deadlock.

In addition to the information required by this algorithm, after each snapshot interval the processes also send statistics of tuple usage, including tuple access frequency and number of stored tuples, to the central server. We are currently working on an enhanced runtime system that uses this information to perform redistribution of tuples among the servers to balance the server’s load. Some of the hints about tuple access patterns could even be given automatically by the runtime system itself. Unfortunately, all these current and planned mechanisms cannot be provided in MPI. An efficient snapshot protocol requires an entity to receive and handle snapshot requests in each process (as the Panda upcalls) and ordered broadcasts.

5. Performance

We conducted some experiments in order to assess the relative performance of the two OpenTS implementations. To accomplish this goal, we used a set of benchmarks representing different parallel programming paradigms. A short description of the benchmarks is given bellow:

TSP (task farming): Solves the Travelling Salesman Problem for a map of 18 cities using a branch and bound algorithm. The jobs are divided by the possible combinations of the first 3 cities.

π calculation (task farming): Computes an approximate value of π by numerically calculating the area under the curve $4/(1+X^2)$. The area is partitioned in N strips by a master process and each job is assigned a subset of the total strips. We divided the area in $32 \times 10^7$ strips.

SOR (grid): Successive Over-Relaxation is a parallel algorithm to solve the Laplace equation over a grid. The algorithm is based on the popular overrelaxation scheme with red-black ordering. We used a grid of size 1024-1024 and a fixed number of 200 iterations.

This set of benchmarks was executed in a cluster of 16 Sun-4 workstations connected by a 10 Mbits/s using OpenTS for MPI and Panda. Finally, the results collected in one host refer to a sequential version of the applications.
Figure 2. $\pi$ calculation performance results.

Figure 2 shows the results obtained for the $\pi$ calculation benchmark with two different inputs. The lower slopes represent a subdivision of work in large grained jobs while the higher slopes results from using fine grained jobs. Both systems have equivalent performance for large grained jobs and scale almost linearly with the number of hosts.

Figure 3. TSP performance results.

However, when the communication/computation ratio increases (with fine grained jobs), Panda has a clear advantage over MPI, both in scalability and absolute performance. Apparently, the multithreaded upcall-based servers can handle much more efficiently the high server loads generated in the fine grained application, when compared with single-threaded servers.

Figure 3 shows the results obtained for TSP. The performance of MPI is superior to Panda for less than 11 processors. In this context the overhead of thread switching in Panda processes prevails. However, as the number of processors increase and the message traffic rises MPI strangles while Panda withstands the increased load.

Figure 4 shows the results obtained for SOR. This benchmark has remarkably different communication patterns. The algorithm works in iterations involving global synchronization points, and therefore its very prone to load unbalancing. This is because prior to starting a new iteration, the faster nodes have to wait until the slower ones finish the previous iteration.

Figure 4. SOR performance results.

Figure 4 shows a similar behavior to the other benchmarks: MPI performs better in a small number of processors, and afterwards the two curves approach until Panda prevails and MPI strangles.

6. Conclusion

In this paper we reported on our experience implementing a high level parallel programming library, called OpenTS, on top of two portable systems: MPI and Panda. The open implementation philosophy of OpenTS gives the programmer the opportunity to perform incremental optimisation of parallel applications without compromising portability. The Panda implementation became naturally elegant and flexible using integrated multithreading and communication in the upcall model. In MPI, the single-threaded process model precluded the implementation of important features such as distributed program termination. Also, to implement some modules we had to descend to dirty tricks using certain MPI undocumented implementation decisions. Finally, the lack of dynamic process creation in MPI limited the normal use of the eval primitive. The experimental comparison of both systems showed that the MPI implementation performs better than Panda for applications with low communication to computation ratios, e.g. large grained jobs in master-worker applications. On the other hand, the Panda implementation performs better for highly communicative implementations. We expect these differences to be more noticeable when using the optimization mechanisms provided by the meta-interface. Panda OpenTS is likely to attain more benefits from optimizations since they were more efficiently implemented on Panda. In this paper we showed that the development of high-level parallel programming systems involving sophisticated and flexible run-time systems can largely benefit from the use of integrated multithreading and communications. These features are currently provided in portable run-time support systems such as Panda, but not in MPI.

7. References