Mechanisms for Programming SMP Clusters *

A. Gürsoy
Computer Science Department
Bilkent University
Ankara, Turkey

İ. Cengiz
Computer Science Department
Bilkent University
Ankara, Turkey

Abstract *Cluster of symmetric multiprocessor systems (SMP Cluster) are becoming increasingly attractive for cost effective high performance computing. Besides building such platforms, providing programming mechanisms, layers of abstractions, or libraries to enable programmers to gain the power of clusters is another challenging field of research. In this paper, we discuss mechanisms on how to take advantage of SMP clusters in a parallel object oriented programming environment. Particularly, we discuss node level replicated parallel objects, parallel objects with a representative per SMP node, as a reusable pattern to perform a set of common collective communication/computations.*

Keywords: smp cluster, parallel programming

1 Introduction

Symmetric Multiprocessor (SMP) platforms are going towards being a general interest in research. As workstations having multiprocessor architectures with shared-memory appear on market, it becomes attractive to build larger multiprocessor systems by connecting such workstations using fast networks, such as Fast Ethernet, ATM switches. Taking SMP nodes as basis and building clusters of them leads a new way of thinking in parallel computing research. Modeling cluster of n k-way SMP nodes as a flat network of nk processors would not be sufficient to extract possible gains of that architecture. Shared-memory structure of an SMP node and the availability of dynamic load balancing for processors within a node are points worth to take into account in designing software systems for SMP clusters. Both shared-memory and distributed-memory paradigms apply in such systems. Processors within the same node (SMP node) share memory, while the nodes within the same system are subject of distributed memory programming. In order to achieve high performance on these machines, the programming model must allow us to exploit advantages of these machines.

There are several programming models for SMP clusters being employed. The simplest one is to treat SMP cluster as a flat network of processors and use message-passing or distributed memory programming models such as using message passing libraries like MPI or PVM. Within an SMP node, however, shared memory programming models can be used for better performance. Such hybrid programming models have been proposed to program SMP clusters [1, 10]. In the hybrid programming model, the SMP cluster is modeled as a network of nodes. An application is developed with the distributed memory approach where nodes exchange messages, but within a node the application employs a multithreaded model to take advantage of multiple processors and shared address space. Higher level programming models also are currently being developed to exploit SMP cluster architectures. In [2], a runtime system with a set of programming abstractions for block-structured applications to take advantage of SMP clusters is presented.

*This work partially supported by TÜBİTAK under the grant EEE-AG 247.

Some other work on SMP clusters emphasizes fast or low overhead communication protocols such as Active messages for SMP clusters [8]. There are other programming models which does not deal with (explicitly) message passing and multithreading. Parallel object oriented languages encapsulates these in the object based model. Charm++ [5] system, developed at UIUC, is an example of such an object oriented parallel programming environment. In this paper, we will discuss, how we can take advantages of SMP clusters within an object-based parallel programming environment. Our emphasis is on reusable patterns or libraries for collective communication and computations that can be used commonly in parallel computations. We introduce and present implementation of node level replicated parallel objects to perform such operations efficiently on SMP clusters. The paper is organized as follows: Section 2 discusses mechanism that we needed for some parallel applications to exploit SMP clusters. Section 3 gives a summary of the Charm++ programming environment [5] and the runtime system that we have used in our parallel application development work. Section 4 introduces node level replicated objects (NLBOC) and their efficient implementation with preliminary results. Finally, we conclude and present further work in this direction.

2 Motivation

This work was motivated while we were developing parallel object-oriented applications with Charm++. In this section, we will briefly discuss the common paradigm that we came across in those applications and the motivation behind developing a framework to exploit the features of the SMP clusters. One of the applications is the hierarchical radiosity [3] calculations for computer vision. In this application, a collection of polygons (will be called patches) which represent a three-dimensional scene to be visualized is distributed across processors. The basic computation is the calculation of interactions (flow of light) between every pair of patches which are visible to each other. Due to the irregular structure of the computation, neither the number of patches nor the pairs that are involved in interactions are known in advance. As the computation continues, the patches might be divided into subpatches depending on the input scene, therefore a static mapping scheme of patches/interactions to processors will not result in good performance.

A parallel object-oriented design of such an application includes calculation objects which handles interactions between patches. If the patches that an interaction object needs are local (i.e. within the same processor where interaction is calculated) then the interaction can be completed without any communication. Otherwise, the data of the remote patches must be brought, and then after the interaction calculation, the remote patch data must be updated.

In [9], a parallel object-oriented approach for hierarchical radiosity is presented. The main focus was to develop a framework such that the details of parallelization are hidden from the computational algorithm. The framework uses the idea of *proxy objects*, that are representative of remote objects. With proxy patches, local representative of a remote patch, the design of the interaction objects and other computational parts of the radiosity algorithm are greatly simplified. The computational part is freed from how the patches are distributed or when they migrate from one processor to another due to dynamic load balancing. As shown in Figure 1, interaction objects has no idea if the patches they are dealing with are local or remote. However, there must be an efficient mechanism, a proxy patch manager, which maintains proxy patches at required processors. The proxy manager is replicated on each node, and makes sure that proxy objects are created at the node where they are needed, and the manager exchanges information with other replicas to maintain consistency of the proxies. Secondly, the calculation objects can be run immediately whenever the patches they

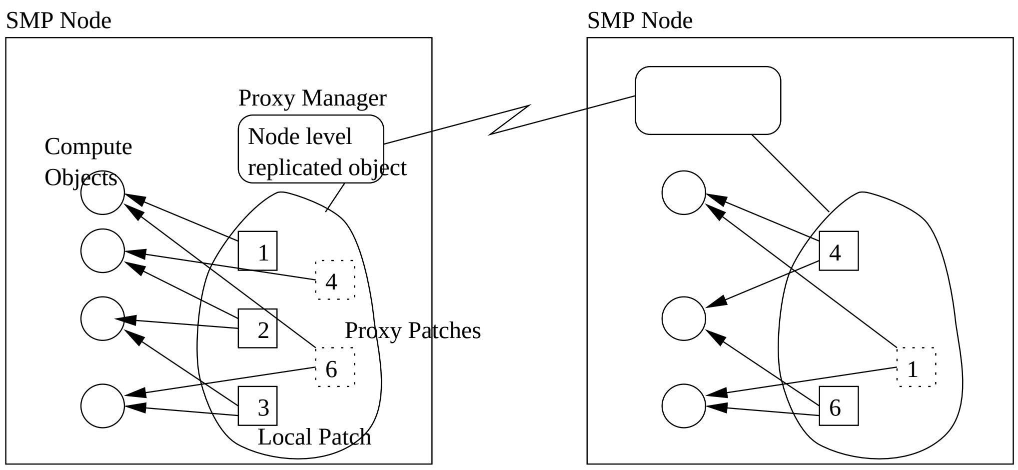

Figure 1: Node level replicated objects and proxies

need are ready to be used. Another application which needed the same framework is a molecular dynamics application [6] being developed at UIUC. In this application, also, the computation domain is divided into patches, a three dimensional space, in this case, and the part of the molecule within a patch interacts with the parts in other patches.

If the underlying machine is an SMP cluster, and if the programming model on the cluster treats the SMP cluster as a network of processors, we end up using the SMP cluster inefficiently. First of all, if an interaction needs a patch assigned to a different processor within the same node, a proxy of the patch need to be maintained at the processor where the interaction is calculated. However, on an SMP node, patches can be accessed directly if they are in the same address space. Secondly, since interactions are assigned to processors, an idle processor cannot calculate another ready interaction which might be waiting for its processor to become idle.

Using an object-based programming environment which supports a network of processors prohibits us to exploit these features. What we need is node level parallel objects, in addition to processor level ones. We need some abstractions with efficient implementations to allow us:

- ability to share the memory across objects on the same SMP node
- ability to run a method of a parallel object by any processor
- a framework which will support the remote-object access easily and collective operations efficiently

3 Charm++

Charm++ [5] is a portable object-oriented parallel programming language. A Charm++ program/computation consists of potentially small-grained objects, called *chares*, and a special type of replicated object, called *branch-office chares* (BOC). A chare object consists of private and public data members and methods like in C++. A public method (*entry method*) of a chare object, on the other hand, can be invoked asynchronously from objects on the same or remote processors. Invoking a method in a remote object can also be thought of as sending a message to it. The message sent to a remote object is passed as argument to the corresponding entry method.

A BOC is an object with a branch on every processor; all of the branches answer to the same name. One can call public functions of the local branch of a BOC, send a message to a particular branch of the BOC, or broadcast the message to all of its branches. BOCs provide a versatile abstraction that can be used to implement static load balancing, local services (e.g. memory management), distributed data structures, and inter-module interfaces. Two library modules, each spread over all of the processors of the system, can exchange control and data easily through public calls to each others local branches. This provides a simple mechanism for distributed flow of data across modules. Of course, function calls used in the traditional programming model also allow such exchange but they do not support encapsulation of the state of the library — so, for example, names of global variables (which *have* to be used to express state) in the two libraries can conflict in such situations. Such usage of BOCs are discussed in [7].

The Charm++ programming environment is built on top of a parallel runtime system called Converse [4]. Converse is an interoperable framework which includes components that supports communication, scheduling, and load balancing. A converse message is associated with a handler function. When the message is received at a processor, it is stored in the incoming-messages queue. The customizable converse message scheduler then dispatches these messages by invoking their handler functions. The Charm++, which runs on Converse runtime system, supports message-driven execution. It repeatedly selects one of the available messages from a pool of messages in accordance with a user selected queueing strategy, restores the context of the chare to which it is directed, and initiates the execution of the corresponding method.

4 Techniques for Programming SMP Clusters

Charm++ language models the parallel machine as network of processors. In this section, we discuss mechanisms needed in Charm++ and Converse layers to use SMP clusters effectively.

Ability to share memory space: In order to share memory (i.e, access objects directly) within a node, the objects must be within the same address space, i.e., belong to the same process. In the network of processors model, Converse runtime creates one process per processor (which executes Converse scheduler) on an SMP node, and parallel objects are mapped to processors, hence they belong to different address spaces. Instead, if one process is created in each node and each scheduler is run by a thread (provided by the operating system) within the same address space, then parallel objects that are mapped to different processors can potentially access each other within the SMP node directly. In a k-processor node, there will be k-threads each running one Converse scheduler, and one communication thread to handle incoming messages from the network. This mechanism was already existing in the version of Converse that we have used. However, it was not sufficient to use effectively SMP clusters

Ability to execute a method of an parallel object on any processor within a node: Since Charm++ system models the machine as a collection of processors, there is no notion of an SMP node. A message has a processor number as the target, and when the message is received, it is put in the scheduler's queue of that processor. Now, only the scheduler which owns that queue can process the message even though some of the other schedulers might be idle in the same SMP node. This problem is solved by creating a node level message queue and by defining node level messages. That is, a message can have a node number instead of a processor as a destination. The node-level message will be inserted into the node level message queue by the communication thread. Scheduler is also modified to check both its own message queue and the node level queue. With this modification, a node-level message can be picked up by any scheduler and corresponding method of the target object will be invoked. These modifications, however, are still in the lower part of the Charm++

programming system (Converse) which deals with messages and schedulers only. These changes do not allow a Charm++ program to use node level messages. We need to augment Charm++ language/programs such that a parallel object is assigned to a node but not a particular processor and when a method of such an object is invoked (possibly remotely), it will be executed on any idle processor on the node. As discussed previously, we have found BOC's extremely useful abstraction to implement certain operations. The proxy-manager discussed before also needs such a structure. So we extended the BOC concept and defined *Node Level Branched Chares — NLBOC*.

4.1 Node Level Branched Chares (NLBOC)

The effective use of branched chares in collective operations, and usability in irregular parallel algorithms brought a new concept; Node-level branched chare classes (NLBOC). A NLBOC is a group of chares that has a branch on each node. Having an instance of a NLBOC means there exists a representative on each node, such that these representatives can be reached just by using the NLBOC handle. Communication can take place between the branches of a NLBOC. Moreover NLBOCs may used to encapsulate node level shared objects. Instead of modifying Charm++ language, we preferred to implement NLBOC using the BOC mechanism. We have developed a base class, `NodeBOC`, which can be used by Charm++ programs. When a message received for a method of a NLBOC, any scheduler can select the message and invoke the corresponding method.

In the Charm++ programming model, a method runs to the completion. Since there is only one scheduler in the address space, it is not possible two methods of the same object executing concurrently or in parallel. In other words, intraobject parallelism is not allowed. However, with SMP support, more than one method of a NLBOC can potentially be invoked at the same time. With NLBOC's intraobject parallelism, programmer need to deal with synchronized access to shared data with locks etc. Many applications might need a NLBOC where only one method can be executed at a given time. To use locks explicitly by the programmer to impose this restriction can worsen the performance and also makes programming difficult. We can give to the programmer a type of NLBOC where only one method could be executed at a given time. There might be various solutions to make sure that only one method being executed. If a scheduler picks up a message for a NLBOC object and finds out that a method of it is already being executed, the scheduler either blocks which we don't want, or it puts back the message in its own queue. This, however, prevents the message to be processed by other processors later. Or the scheduler puts the message back into the node level message queue. However, concurrently running schedulers then might keep picking the message up and putting it back into the queue repetitively as long as the object is busy. Another approach would be to store the message into a local queue owned by the NLBOC object. But this will result in schedulers checking many queues (as many as NLBOC objects created in the node). We developed a token based algorithm to solve this problem efficiently (illustrated in Figure 2). Each branch of a node level object has a local message queue and a token. In order to invoke a method , the scheduler must find the token of the NLBOC branch free, otherwise, it stores the message in NLBOC's local queue. The token is available only if none of the NLBOC's methods is executing. When the method is invoked and the token is free, then the execution of the method starts and the token is locked. When the execution of the method is finished, the NLBOC checks if its local queue has any messages. If so, it puts the token into the node level queue. Then, the next scheduler picks up the token and invokes the handler for that token which eventually dequeues a message from the local queue of NLBOC. Table 1 illustrates the effectiveness of the token algorithm on a two-processor SMP node. With more processors within the SMP node, we expect the token

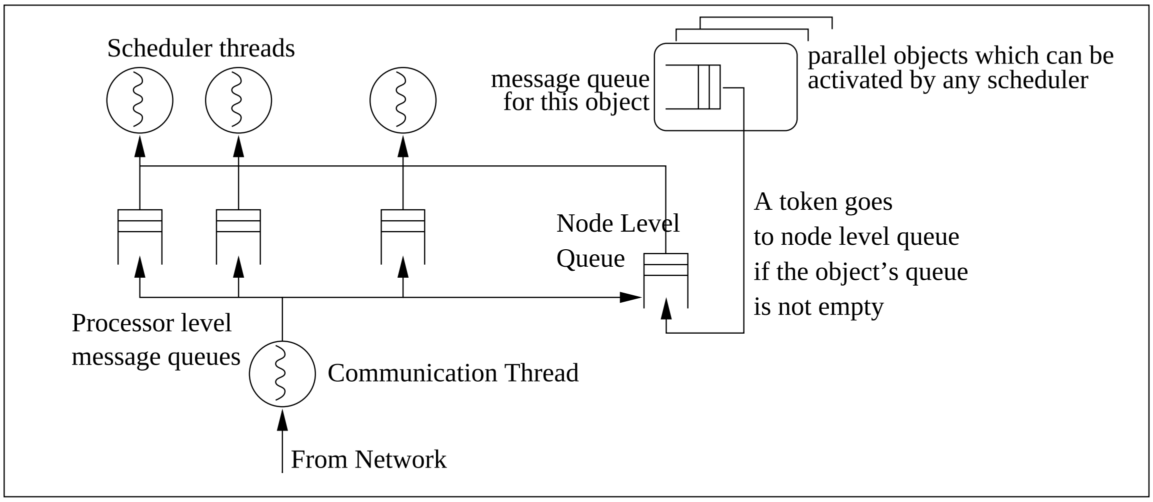

Figure 2: Efficient implementation of NLBOC

based algorithm to be even much better compared to the other approaches.

4.2 Design of Optimized Collective Communication and Computation Patterns

NLBOC pattern is a mechanism that we need to implement the proxy-manager for the hierarchical radiosity example. In addition, NLBOC's can be used to implement efficient collective operations on SMP clusters. These include broadcast, reduction, and gather/scatter type collective communications. For example, broadcast operation can be optimized to take advantage of shared memory within SMP nodes. Across SMP nodes, a spanning-tree based algorithm can be used (in Charm++, there are collective communication libraries which are implemented where BOCs are organized in a spanning tree form). In a k-processor SMP node, however, instead of having k branches, only one branch of the communication NLBOC can handle work to be done. A broadcast operation can be optimized, for example, if the broadcasted information is read-only large-sized data by keeping one copy within the NLBOC and distributing pointers to the shared area to the objects that are recipients of the broadcast information. Table 2 shows preliminary results towards developing efficient collective operations for SMP clusters. In addition to the performance, developing libraries or reusable patterns was another major motivation for implementing such algorithms and NLBOC mechanism.

5 Conclusion

A programming model which treats n node SMP clusters as a network of nk processors, where k is the number of processors in one node, might prevent us to extract maximum performance from SMP clusters. In this work, we stress on reusable patterns or libraries that model SMP clusters properly. Particularly, we discussed node level replicated objects and their efficient implementation in a parallel object oriented programming model. We believe that node level replicated objects provide a versatile abstraction to implement various computations. We plan to develop libraries for commonly used kernel communication/computations using node level replicated objects to facilitate development of parallel programs on SMP clusters.

Table 1: Timings for queueing strategy (in millisecs) on a node of two processors. Each request takes O(n^2) time (n is the size of the message), and two requests in a node are not executed concurrently(synchronization needed).

# requests	Scheduler Queue	Node-Level Queue	Local Queue with Token Alg.
100	564	562	340
500	2778	2876	1700
1000	5600	5828	3488
1500	8310	8710	5102
2000	10980	11580	6780

Table 2: Timings for Broadcast (in millisecs) on a 2-processor SMP node.

msg size(bytes)	BOC based broadcast implementation	NLBOC based broadcast implementation
1000	6.68	5.94
5000	29.14	24.4
10000	56.85	45.57
20000	113.35	93.15

References

[1] D. A. Bader and J. JaJa. SIMPLE: A Methodology for Programming High PerformanceAlgorithms on Cluster of Symmetric Multiprocessors (SMPs). UMIACS Technical Report 97-48, May 1997.

[2] S. J. Fink. A Programming Model for Block-Structured Scientific Calculations on SMP Clusters. Ph. D. Dissertation, UCSD, June 1998.

[3] P. Hanrahan et. al. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197-206, Jul 1991.

[4] L. V. Kale, M. Bhandarkar, N. Jagathesan and S. Krishnan. Converse: An Interoperable Framework for Parallel Programming. *Proc. of IPPS'96*, pp. 212-217.

[5] L.V. Kale and S. Krishnan. Charm++: A Concurrent Object-Oriented System Based on C++. *Proc. OOPSLA'93*, pp. 91-108.

[6] L. V. Kale et. al. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 1998. In press.

[7] L.V. Kale and A. Gursoy. Modularity, reuse, and efficiency with message-driven libraries. *Proc. of the Seventh SIAM Conference on Parallel Processing For Scientific Computing*, San Fransisco, California, Feb 15-17 1995, pp. 738-743.

[8] S. S. Lumetta et. al. Multi-Protocol Active Messages on A Cluster of SMPs. *Proc. of SC97*, 1997.

[9] A. R. Sireli. Parallel Hierarchical Radiosity. M.S. Thesis, Bilkent University, Ankara, Jan 1999.

[10] Y. Tanaka et. al. COMPaS: A Pentium Pro PC-based SMP Cluster and its Experience . *IPPS Workshop on PC-NOW'98*, LNCS, Vol. 1388, pp. 486-497, Springer-Verlag (1998)