Supporting Machine Independent Parallel Programming on Diverse Architectures

W. Fenton B. Ramkumar V. A. Saletore* A. B. Sinha
L. V. Kalé

Department of Computer Science,
University Of Illinois at Urbana–Champaign,
Urbana, Illinois 61801

Abstract

The Chare kernel is a run time support system that permits users to write machine independent parallel programs on MIMD multiprocessors without losing efficiency. It supports an explicitly parallel language which helps control the complexity of parallel program design by imposing a separation of concerns between the user program and the system. The programmer is responsible for the dynamic creation of processes and exchanging messages between processes. The kernel assumes responsibility for when and where to execute the processes, dynamic load balancing, and other "low" level features. The language also provides machine-independent abstractions for information sharing which are implemented differently on different types of machines.

The language has been implemented on both shared and nonshared memory machines including Sequent Balance and Symmetry, Encore Multimax, Alliant FX/8, Intel iPSC/2, iPSC/860 and NCUBE/2, and is being ported to NUMA (Non Uniform Memory Access) machines like the BBN TC2000. It is also being ported to a network of Sun workstations. We discuss the salient features of the implementation of the kernel on the three different types of architectures.

1 Introduction

The Chare kernel is a runtime system that supports an explicitly parallel language. It was designed with two major objectives. First, it should allow machine independent parallel programming over the class of MIMD machines, without losing efficiency. Second, it should help control the complexity of parallel programming. The language is suitable for programming both shared and nonshared memory machines. The Chare kernel creates a division of labor between the programmer and the system, wherein the programmer is responsible for the creation of parallel sub-computations, while

*Currently at Oregon State University

the system decides when and where to execute them. In addition, the language provides machine-independent abstractions and features (particularly those for "specific information sharing modes") that are implemented differently but efficiently on different types of machines.

Care has been taken to ensure that the abstractions are not overly specialized or narrow. In this sense, the Chare kernel represents the lowest level at which it is possible to abstract over resource management and machine-dependent expression. In addition to serving as a direct application programming language, the Chare kernel can serve as a "back-end" for other explictly or implicitly parallel higher-level languages, parallelizing compilers, and domain-specific application packages.

The language has been implemented on many shared memory machines (Sequent balance, Symmetry, Encore Multimax, Alliant FX/8), and nonshared memory machines (Intel's iPSC/2, iPSC/860 and NCUBE/2), and is being implemented on NUMA (Non Uniform Memory Access) machines like BBN TC2000 and a network of SUN workstations.

In this paper we describe how this machine independent language is supported efficiently on these different architectures. We first give a brief description of the important language features. The implementation techniques and algorithms used for supporting each one these features on different classes of architectures constitute the rest of the paper.

2 Language Features

The basic entity in a Chare kernel program is a *chare* (old English for chore or a small task). It is a medium- grained process that can dynamically create other chares, send messages to other chares, and share information with other chares using specific information sharing primitives (described below). The kernel is free to schedule the chares on any processor it chooses to use. The language provides for another type of chare, called a branch office chare (BOC for brevity). This is discussed below in Section 2.2. A Chare kernel program consists of chare definitions, BOC definitions, declarations of specifically shared objects, function definitions (in the language C), and associated type definitions.

2.1 Chares and Messages

A chare definition (such as the one shown in Figure 1) consists of the name of the chare type, its local variables, followed by a sequence of entry point definitions. Each entry point definition begins with a unique label, an associated message declaration, and a block of C-code that deals with this message. This code may access the local variables of the chare. The code may contain the following system calls.

The call *CreateChare(chareName,entryPoint, message)* is used for creating an instance of a chare named *chareName*, and directs the system to begin its execution with the given *message* at the given *entryPoint*. This call returns immediately; sometime in the future the system will actually create and schedule the new chare. The call *MyChareID* returns the chare-ID of the calling chare; this id can be sent in messages. The call *SendMsg(chareID, entryPoint, message)* deposits the message to be sent to the given entry point of the chare with the given chareID. In addition, the

```
BranchOffice Example1 {
    manager {
        /* Local Variables */
        entry EPM1 :  (message MESSAGE_TYPE1 *msg)
            C-code-block
        ..
        entry EPMX : (message MESSAGE_TYPEx *msg)
            C-code-block
        FunctionM1(..)
            C-code-block
        ..
        FunctionMZ(..)
            C-code-block
    }
    branch {
        /* Local Variables */
        entry EPB1 :  (message MESSAGE_TYPE1 *msg)
            C-code-block
        ..
        entry EPBY : (message MESSAGE_TYPEy *msg)
            C-code-block
        FunctionB1(..)
            C-code-block
        ..
        FunctionBZ(..)
            C-code-block
    }
}
```

Figure 1: Syntax of a Branch Office Chare

kernel provides functions to allocate and free messages and storage, to terminate the execution of the calling chare, and for signaling termination of the whole parallel program.

Every Chare kernel program must have a main chare definition. The main chare definition is like any other chare definition except that it must contain an *Init* entry point, in addition to other application specific entry points. Program execution begins at the *Init* entry point. BOC initialization (see Section 3.1) and the creation of specifically shared objects (see Section 2.3) is performed here.

2.2 Branch Office Chares

Every BOC consists of a manager chare and a branch chare on every processor (see Figure 2 for a sample definition). A branch chare's definition is similar to the regular chare definition, except that the functions defined in a branch can be called by other chares running on the same processor. These functions as well as the code at the entry points may use the *SendMsgBranch(entry, msg, pe)* call to send a message to the instance of the calling BOC on the given processor (numbered *pe*).

```
BranchOffice Example1 {
    manager {
        /* Local Variables */
        entry EPM1 :  (message MESSAGE_TYPE1 *msg)
            C-code-block
        ..
        entry EPMX : (message MESSAGE_TYPEx *msg)
            C-code-block
        FunctionM1(..)
            C-code-block
        ..
        FunctionMZ(..)
            C-code-block
    }
    branch {
        /* Local Variables */
        entry EPB1 :  (message MESSAGE_TYPE1 *msg)
            C-code-block
        ..
        entry EPBY : (message MESSAGE_TYPEy *msg)
            C-code-block
        FunctionB1(..)
            C-code-block
        ..
        FunctionBZ(..)
            C-code-block
    }
}
```

Figure 2: Syntax of a Branch Office Chare

The main chare interacts with the different manager chares via their respective access functions.

All the branch chares of a BOC have the same ID. This ID is assigned when a BOC instance is created, and may be passed in messages by other chares. Thus, a BOC instance may be created in the main chare with the call:

bocID = CreateBoc(bocName).

Now, the *bocID* can be passed in messages to different chares and two chares running on two different processors, may use the same *bocID* to access the (functions of the) local representative of the BOC instance. The BOC representatives on different processors may coordinate information among themselves using messages (via SendMsgBranch call).

2.3 Specific Information Sharing Abstractions

In addition to messages and BOCs, chares may share information with each other using the following abstract data types, each designed for a specific mode of information sharing.

A **read-only** variable is initialized at the *Init* entry point and can only be accessed via the call *ReadValue* from any other chare. This call simply returns the (fixed) value of the variable. A read only variable may be a scalar (e.g. integer) , array or a structure.

A **write-once** variable is created and initialized any time (and from any chare) during the parallel computation. Once created, its value can only be read. The creation is done via a non-blocking call *WriteOnce(dataptr, datasize, entryPoint, ChareID)* which immediately returns without any value. Eventually, the variable is "installed", and a message containing a unique name assigned to the new variable is sent to the designated *entryPoint* of the designated chare. This ID can be passed to other chares, which can access the variable by calling *DerefWriteOnce*.

A **monotonic** variable is global variable that "increases" monotonically in some metric by the application of an idempotent function. It is used typically in branch-and-bound computations. Its (approximate) current value can be read by any chare at any time using the call *MonoValue*, and a potential new value for it can also be provided by any chare using the call *NewValue*. The supplied value replaces the old value if it is "better" than the old value, using a user-supplied comparison function. It is only guaranteed that the value read from any other chare will be *eventually* better or equal to the new value supplied.

Accumulator variables are counters, with one difference. The initial value of an accumulator must be *zero*. Accumulator variables have associated with them two functions - an *accumulating* function that adds to the counter, and a *combining* function that combines two counter variables. An accumulator variable is initialized during initialization of the main chare, and can be read only once, destructively. It can be modified only via a function *Accumulate*, which adds a given value to the accumulator. The destructive read is performed via the (non-blocking) call : *CollectValue(accumulator-name, entryPoint, ChareID)* which results in eventual transmission of a message containing the final value of the accumulator to the named chare. It is easy to think of the accumulator as an integer to which we want to add other integers from time to time, although the language allows it to be any type, with any user-defined commutative associative operation.

A **dynamic table** consists of a set of entries, each with a key part and a data part. Various asynchronous access and update operations on entries in the table are provided. For example, one may call *Find(key, entryPoint, chareID)*; The call immediately returns, and eventually a message containing the data associated with the given key is sent to the specified chare at the specified *entryPoint*.

The language description above is necessarily sketchy. See [11, 12] for a complete description of the language. For the purpose of this paper, we have identified some key features of the language, so we can elaborate on how they are supported on different architectures.

3 The System Core

Conceptually, the kernel can be viewed as a work pool "manager". It manages a pool of messages representing seeds for new chares or messages to existing chares. Each message is destined for a specified entry point in the program code (see Figure 1), and processing a message involves executing the code associated with the entry point sequentially without interruption. Once a

message is processed, control returns to the kernel. Thus, the kernel is in a "pick-n-process" loop, constantly picking messages from the underlying work pool and processing them one by one. The kernel exits from this loop only when global termination is detected.

In this section, we discuss some of vital functions of the kernel and how their implementation differs from architecture to architecture. Many of the system functions have been implemented as branch office chares. So, one of the first tasks of the kernel upon start-up is the initialization of the system and user-defined BOCs, which is discussed in Section 3.1. In Section 3.2 we describe the "pick-n-process" loop for each different machine type. The processing of these picked messages is discussed in Section 3.3. On nonshared memory and NUMA type architectures, it is necessary to "pack" messages into a contiguous format before transmission. We discuss packing in Section 3.4.

3.1 Branch Office Initialization

The initialization of the BOCs is carried out by making *BocInit* calls during the execution of the code in the *Init* entry point in the main chare. The processing of these messages is different for shared memory and nonshared memory architectures, as is shown in Figure 3. On shared memory machines a separate list of the BOC creation messages is maintained in the work pool to ensure that the *BocInit* messages are processed first. All *BocInit* messages are created during system start-up and inserted into this list, and are therefore processed first.

On NUMA type architectures like the BBN, the initialization will be done slightly differently. During start-up, for each processor, one *local* copy of the *BocInit* message is created and inserted in a processor-private *BocInit* message list. Each processor then simply picks messages off its own private list and processes them. It differs from shared memory machines only in that on NUMA machines an explicit *remote* to *local* copy of the message is carried out during start-up.

On nonshared memory machines, the main node (processor 0), after executing the *Init* code, sends one message containing the values of all *ReadOnly* variables and one message each for every BOC instance initialized. It then sends any other messages (e.g. a CreateChare message). On the other nodes in the system these messages may be received out of sequence. Yet all the BOCs and the readonlys must be installed before any other messages are processed. To accomplish this node 0 sends a message containing the count of the number of initialization messages it sent. When *BocInit*, *ReadVar*, or *InitCount* messages arrive at a node, they are processed immediately. Other messages are buffered at this point to be processed during the pick-n-process loop. The initialization phase is complete when the *InitCount* message has been received **and** the number of initialization messages remaining to be received is zero.

3.2 The Pick-n-Process Loop

Once initialization is complete, the system enters the the "pick-n-process" loop (see Figure 4). On shared memory machines, the work pool is implemented using a shared queue. The loop is thus quite simple; it simply picks up the next available message in the work pool and processes it.

On nonshared memory machines, once again the protocol is a little more complicated. Unlike

```
Initialization() /* for nonshared memory */
countMsgArrived = FALSE; msgs_left = 0;
while (countMsgArrived == FALSE || msgs_left != 0)
       foundMessage = FALSE;
       while (not foundMessage)
       if (McProbe())
         msg-size = ArrivedMsgLength();
         message = Allocate(msg-size);
         McSyncReceive(msg-size, message);
         foundMessage = TRUE;
       switch (message->type)
       case BocInit:
         Create-and-Initialize-Boc(message->BocNum);
         msgs_left--;
       case InitCount:
         countMsgArrived = TRUE;
         msgs_left += message->expectedmessages;
       case ReadVar:
         Initialize-Read-Only-Data(message);
         msgs_left--;
       default:
         QSEnqueueMsg(&message);


Initialization() /* for shared and NUMA machines */
while (not QSBocQueueEmpty()) /* Boc fn call */
       QSDequeueBocMsg(&message); /* Boc fn call */
       CreateBranch(message->BocNum);
```

Figure 3: The *Initialization* procedure for nonshared and shared memory machines.

shared memory machines, it is necessary to explicitly send and receive messages between processors. Incoming messages need to be periodically received by every node and then inserted into the local work pool. This is done once every iteration of the loop by the *PumpMessages* function. The next available message is picked up for processing. If the work pool is empty, the node tries to 'pump' messages repeatedly until a message in the work pool is available. When the work pool is not empty, the next message in the pool is picked up and processed.

NUMA type architectures are interesting in that it turns out to be more efficient to implement the pick-n-process loop as a nonshared memory machine. We choose to pretend that explicit sends and receives are necessary as with nonshared memory machines, but implement the "sends" and "receives" differently. Each processor will have several arrival queues, typically partitioning all the processors into sets and mapping each set to one arrival queue. This bounds the overhead of mutual exclusion and also reduces the number of arrival queues necessary. These queues are "pumped" as on nonshared memory machines by the local processor. Also, sends and receives are implemented via "intelligent" block copying, as is explained in Section 3.4. But for these differences, the NUMA

```
Pick-n-Process() Initialize-System-Bocs();
Initialization();
while (not System-Done)
    message = NULL;
    while (message == NULL)
       TimerChecks(); /* Boc fn call */
       PeriodicChecks() /* Boc fn call */
       if (NONSHARED || NUMA)
            PumpMessages();
       PickNextMessage(&message); /* Boc fn call */
    if (NONSHARED || NUMA)
       ConditionallyUnpack(&message);
    ProcessMessage(message);


ConditionallyUnpack(message) /* for nonshared */
if (message->packid != NULL_PACK_ID)
    (UnpackTable[message->packid])(&message);


ConditionallyUnpack(message) /* for NUMA machines*/
if (message->packid != NULL_PACK_ID)
    if (message->type == NewChare)
       (PackTable[message->packid])(&message);
       (UnpackTable[message->packid])(&message);
    else /* message is either ForBoc or ForChare */
       (UnpackTable[message->packid])(&message);
```

Figure 4: The "pick-n-process" loop of the Chare kernel for shared, NUMA type and nonshared memory machines.

```
ProcessMessage(msg)
switch (msg->type)
caseNewChare:
    DataArea = Alloc-Chare-Data-Area(msg->datasize);
    (EntryPointTable[mse->InitEP])(msg, DataArea);
case ForChare:
    DataArea = Get-Chare-Data-Area(msg);
    (EntryPointTable[msg->ForEP])(msg, DataArea);
case ForBoc:
    DataArea = BocDataTable[msg->BocNum];
    (BocEPTable[msg->ForEP])(msg, DataArea);
case Terminate:
    CkSendStatistics();
    System-Done = TRUE;
```

Figure 5: The *ProcessMessage* function of the Chare kernel.

type architecture is more or less viewed as a nonshared memory architecture.

Inside the "pick-n-process" loop, shown in Figure 4, there are two BOC function calls, namely, *PeriodicChecks* and *TimerChecks*. These functions are implemented as BOCs on all three types of machines. They execute specified functions at periodic time intervals and specific times, respectively.

3.3 Processing Messages

A message picked from the work pool is either a *NewChare* message, i.e. a message created by a *CreateChare* call, a *ForChare* message, i.e. a message created by a *SendMsg* call, a *ForBoc* message, i.e. a message created by a *SendMsgBranch* call, or a *Terminate* message which initiates the global termination protocol. With the exception of *Terminate* messages, all messages have an entry point associated with them.

A translator translates the language described in Section 2 into C. Every entry point and the code associated with it is translated into a C function. Moreover, the translator generates code to create two tables of function pointers called *EntryPointTable* and *BocEPTable* for chare entry points and BOC entry points respectively. These tables are indexed by a unique entry point ID and the function corresponding to that table entry is invoked (see Figure 5).

Upon encountering a *NewChare* message, the kernel creates the data area associated with the newly created chare and jumps to the entry point specified in the message. A chare once created is *anchored* to the processor it is created on (i.e. all messages for this chare are executed on the same processor from then on). The ID for any chare is the pair <*procID, data-area-ptr*>. *ForChare* messages are always sent to entry points of chares with a specified chareID as described in Section 2.1. *ForBoc* messages are similar to *ForChare* messages except that the system maintains a table of branch office data areas indexed by the branch office number.

On shared memory architectures, anchoring of chares provides mutual exclusion on chare and branch data areas, as multiple messages for the same chare instance cannot execute concurrently. On nonshared memory machines, the anchoring makes it possible for the kernel to direct the *ForChare* messages efficiently to the appropriate processors.

3.4 Conditional Packing

One of the problems that needs addressing when programming nonshared memory machines is that messages that are transmitted from one processor to another need to be packed in a contiguous format for transmission, since pointers are generally not valid across processors. This can be costly when the data structures are complex, for example, large graphs or trees. For efficiency reasons, it is only necessary to "pack" messages into this format when messages are transmitted, and not when they are kept in the same address space. Thus, for shared memory no packing is necessary, and in clustered memory architectures, no packing is necessary when messages remain in the same cluster.

The Chare kernel, based on load balancing needs, dynamically determines which messages are kept local and which are transmitted. However, it does not know the structure of user data and hence how it must be packed. This problem is overcome by requiring the user to provide "pack" and "unpack" routines for each message type used by the program. These routines are invoked by the kernel as and when necessary. If a decision is made to send a message out to another processor, the appropriate "pack" routine is called (and correspondingly the "unpack" routine is called by the receiving processor). This permits the kernel to avoid this packing overhead for messages considerably, since on nonshared memory machines, messages are not packed if they are eventually serviced on the sender's processor.

Packing and unpacking can be optimized considerably on NUMA type machines. On such machines, to reduce the overhead of accessing non-local memory, sends and receives are implemented using explicit copying across memory clusters. For example, certain messages (like *ForChare* and *ForBoc* messages) are fixed destination messages. If they need to be sent off to a processor in another cluster, the system packs them locally, and then enqueues the message in the destination processor's arrival queue. When this message is "received" from the arrival queue, it is block copied into the local memory and then inserted into the local work pool. *NewChare* messages may be serviced locally, or be relocated several times by the load balancing module before they are eventually picked up for execution. If it is finally processed on the same processor that created it (or on the same cluster), packing would be an unnecessary overhead. Therefore such messages are not packed but simply inserted in the destination (chosen by the load balancing module) processor's arrival queue. When such a message is picked up for execution by a different processor, the processor will first pack it into its local memory (effectively copying it from remote to local memory in the process), then unpack the message to restore any pointers if any, and process it (see Figure 4). We plan to use the scheme for machines like the BBN Butterfly (and TC2000). If the machine has a hierarchically clustered memory architecture (like CEDAR), then no packing is done unless a message crosses a cluster boundary.

4 Load Balancing As A Branch Office Chare

The dynamic load balancing (LDB) module of the Chare kernel is implemented as a BOC. The LDB-BOC provides entry points and function calls and interacts with other Chare kernel system BOCs to dynamically balance load. At the BranchInit entry point the number of neighbors and the list of neighbors are recorded and stored in the data area of the BOC. This information depends on the interconnection network amongst the processors and is later used by the LDB-BOC to balance load.

Figure 6 shows the *Adaptive Contracting Within Neighborhood* (ACWN) [10] load balancing scheme implemented in the **nonshared memory** version of the Chare kernel as a BOC. In the ACWN scheme when a *NewChare* message is generated, the LDB-BOC function *Schedule-NewChare()* is called. The function determines the least loaded neighbor in the processor's neighborhood and requests the Communication-BOC (BOC responsible for communicating with BOC's on other processors) to send the message to that neighbor. The processor may also want to send its own load status to this neighbor with the message. If the local processor is itself the least loaded, the message is enqueued in its local queue. Processors also exchange explicit load status messages periodically to keep their status information updated on their neighbors. An entry point is provided for this purpose.

For a different load balancing scheme such as *gradient model* [13] the load balancing process may be awakened periodically to balance loads whenever the pressure gradient falls or rises above a certain threshold. Additional entry points may be added as desired for implementing different schemes. Thus, the BOC offers the versatility and ease in implementing different load balancing schemes in the Chare kernel.

The LDB-BOC on **shared memory machines** deposit all *NewChare* messages into a shared pool of work employing a stack or a priority queue. Since processors pick messages from a shared pool, the load balancing is trivial.

On NUMA machines, as well as on large shared memory machines, we use multiple stacks or queues to store messages. Each processor has its own queue. Dense graphs are used to 'interconnect' the processors [16]. During BranchInit the LDB-BOC initializes its list of neighbors using some software interconnection scheme. When a new chare is created the LDB-BOC may deposit it in the processor's local queue or that of a neighbouring processor based on the size of the queue. When a processor needs work, the LDB-BOC first checks its own queue and if it is empty then the processor scans the queues of its neighbors in a round-robin fashion. Work is accessed from the first processor with a non-empty queue.

5 Supporting Information Sharing Abstractions

In addition to messages, chares can share data with the five information sharing abstractions described briefly in Section 2. Their implementations on the various machines are briefly described in this section.

On **shared memory machines** with a small number of processors, each shared variable,

except the dynamic table, is implemented as a shared entity, with an associated lock. Operations are performed in a mutually exclusive manner using locks. Dynamic tables are managed as arrays of chains of entries. A *hashed chaining* scheme is used. The key of an entry is used to map into an index in the array, which is a chain of entries whose keys map to the same index. A lock is associated with each index in the array to provide mutually exclusive access to chains. The same scheme is used for both small and large shared memory machines, but the size of the arrays become larger for larger machines.

On **nonshared memory machines** Read Only, monotonic, WriteOnce and accumulator variables have a local copy of the variable on each node, while a dynamic table is split across nodes. Initialization, update and access are the three generic operations that can be carried out on these variables. A BOC is used to implement these operations for monotonic, accumulator, WriteOnce and dynamic tables. Since a Read Only variable is never updated it does not need a branch office chare to manage its updates.

In the execution of a Chare kernel program initializations are carried out in two phases. In the first phase, system variables and system BOCs are initialized. In the second phase user initializations, as specified in the *Init* entry point of the *main* chare are carried out. This consists of sending out initial values of read only, monotonic and accumulator variables, and initializing user defined BOCs.

Write Once variables are initialized by the *CreateWriteOnce* call. A copy of the variable is first sent to the manager of the corresponding BOC. The manager assigns it a unique index, which serves as the *WriteOnce ID*. It then broadcasts the value and ID of the variable to each of the branch nodes. Each node, after creating a copy of the write once variable, sends a message to the manager (along a spanning tree rooted at the manager to avoid bottlenecks) that it has created the variable. When it has received an acknowledgement message from all the nodes, the manager sends the ID of the write once variable to the supplied address. A write once variable can be read by means of the *DerefWriteOnce* call. This call returns the pointer to the local copy of the variable. The pointers to all the WriteOnce variables are stored in an array indexed by the WriteOnce ID.

An update on a monotonic variable is done by the *MonoValue* call. The call results in the branch chare updating its local value, and sending a copy of the new value up to its parent branch chare in the spanning tree created on the nodes. Every branch combines values it receives from its children with its own by waiting for some fixed interval of time before sending its local value up to its parent branch chare. The root of the tree sends its update to the manager chare, which broadcasts the value to all the branch chares. The value of every update may not be simultaneously available to every branch, but shall be eventually available. A monotonic variable can be accessed using the *MonoValue* system call. This call simply returns the value of the local copy of the variable on that node.

The *Accumulate* system call results in the application of the *adding* function on the local value on the branch chare. The *CollectValue* system call is used to read the value of an accumulator variable. This call results in the branch chares sending up their local values of the accumulator variable to the manager chare up a spanning tree on the nodes. Branches use the *combining* function to combine values they receive from their children nodes. The manager communicates the final value to the supplied chare.

Updates on entries in a dynamic table can be carried out by calling the system calls *Insert* and *Delete*. Again as in the case of shared memory systems a *hashed chaining* hashing scheme is used. The key of an entry is hashed to obtain the processor number of the branch which stores the portion of the table to which this entry belongs, and the index in the table on that branch. An update message is sent to the required branch, which carries out the update operation and back-communication of update, if specified in the call options. The *Find* call is used to read entries in dynamic tables. The key provided is used (as described above) to determine the branch and index. A message is sent to the corresponding branch chare to find the entry and reply back to the supplied address.

We are still in the process of designing the implementation details of the information sharing abstractions for the **NUMA machines**.

6 Discussion

The Chare kernel runs as an application program on top of the vendor-supplied operating system on the different machines we have talked of in this paper. We also plan to port it to the CEDAR machine at CSRD (University of Illinois), IBM's RP3 and a transputer based machine. We faced only one serious problem while implementing the Chare Kernel. On the NCUBE the host and the nodes follow different byte-orderings. Our initial nonshared memory implementation (on Intel ipsc/2) used to place the main chare on the host and all other chares on the nodes. But, because we could not convert the byte-orderings of arbitrary user messages between the host and the nodes to conform to the NCUBE requirements, we had to move the main chare from the host to node 0 for the NCUBE. Thus only system level messages (with definite formats) were exchanged between the host and the nodes. Subsequently we carried over the change to the other nonshared machines. Now on all nonshared machines the host starts up the nodes, and then waits till it gets a termination signal from the nodes.

The performance data obtained on benchmarks has been very encouraging. A parallel program loses less than 5 percent of its speed compared to a sequential C program based on the same algorithm; Moreover, speedups with increasing number of processors are excellent. Performance evaluation of the Chare kernel is beyond the scope of this paper. Sample speedups are shown Table 1 for one shared and one nonshared memory multiprocessor for an instance of the 15-puzzle problem using parallel IDA^* that explores 1.4 million nodes in its search space. The execution times (in seconds) of the Sequential C program and the Chare Kernel program (1 processor) occur in parentheses.

Additional performance data can be found in [11]. A parallel Prolog compiler which exploits both AND and OR parallelism has been written using the Chare kernel system [15], and is one of the first such systems to run efficiently on both shared and non-shared memory system. A state-space search package was also developed using the Chare kernel. Many other parallel applications are being developed using this system.

Machine	C	1	2	4	8	16
Symmetry	(367.4)	0.99 (371.8)	1.97	3.92	7.78	13.92
iPSC/2	(379.3)	0.99 (385.0)	1.93	3.72	6.77	12.60

Table 1: Performance of parallel IDA* algorithm on a 15 puzzle problem that explores 1.4 million nodes.

7 Related Work

One of the early attempts at machine independence was the development of the "Argonne macros" [3] aimed at porting compilers for parallel Prolog across shared memory machines. This attempt, although significant in the direction it chose to take, did not go far enough, restricting its attention to a single type of architecture.

A project with similar objectives as ours is the recently published VMMP project [9]. Like the Chare kernel, it too is designed for developing portable and efficient software on both shared and nonshared memory machines. One of the primary differences appears in the implementation of different information sharing abstractions in the two systems. VMMP too supports different information sharing abstractions, but as "shared objects". On VMMP, shared objects are located at one "central" site and access involves blocking and an RPC type protocol, unlike the Chare kernel information sharing abstractions discussed in Sections 2.3 and 5. We believe that our choice will make it possible to implement each of the abstractions more efficiently, especially on distributed memory machines. Another significant difference is in the programmer's view of the system, and the programming paradigm. In the Chare kernel, the programmer is only concerned with creating new chares and sending messages to existing chares. Execution is completely message driven. The execution model of VMMP differs substantially from the Chare kernel in this respect in that it is a *blocked workers* model. Also important is that there is no context switching between explicit processes in our model and hence no associated overheads are incurred.

There are other efforts aimed at machine independence. Strand [8] is a language based on asynchronous processes and streams, and is portable across many parallel machines. Linda [4] is another language based on the notion of a shared tuple-space. Actors [1] and concurrent object-based languages are comprised of processes which communicate by depositing and/or removing tuples with specific patterns. The language described by us differs from these mainly in its rich set of diverse information sharing modes, and on its reliance on implicit dynamic load balancing for scheduling work.

The language we described is different from the distributed system kernels such as the *V* kernel [5], or the *Amoeba* system [14] which essentially provide support for communicating processes. The Reactive Kernel/Cosmic Environment [2] also supports machine independent programming by providing communication mechanisms. The recently proposed Concurrent Aggregates (CA) language [6] bears some similarities with the branch-office chares in our language, although CA is

aimed at a fine-grained machine (The J-machine) [7] being built at MIT.

Implicitly parallel higher level languages such as Functional or Logic languages constitute another approach to machine independence. We believe that such languages should be built on top a system such as the Chare kernel to simplify the task of building them, and to render the implementation machine independent.

8 Summary

Architecture independence may seem an elusive property. However, we have shown that with an appropriate selection of primitives, and an implementation of these primitives tuned to each specific machine, it can be attained. The chare kernel recognizes locality of reference as the key principle that unifies MIMD machines, with or without shared memory. The chares with their local variables, and the message-driven execution enhance locality. Implementation of messages is tuned to, and an appropriate form of dynamic load balancing is used for each target architecture. The common modes of sharing are encapsulated in data abstractions, so that they are implemented efficiently on each target machine. Conditional packing ensures that programs written using this system can compete with even those written specifically for shared memory machines, as long as the grain size required is not too fine. Scalable techniques used in its design and implementation ensure that the system runs efficiently on machines with thousands of processors.

References

[1] Agha G.A. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[2] Athas W.C., Seitz C.L. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 9–24, August 1988.

[3] Boyce J., Butler R. *et al. Portable Programs for Parallel Processors.* Holt, Rinehart & Winston, New York, 1987.

[4] Carriero N., Gelernter D. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 323–357, September 1989.

[5] Cheriton D.R. The V Distributed System. *Communications of the ACM*, 314–333, March 1988.

[6] Chien A., Dally W.J. Concurrent Aggregates (CA). In *ACM SIGPLAN Principles and Practice of Parallel Programming*, Seattle, Washington, March 1990.

[7] Dally W.J. Fine-Grain Message-Passing Concurrent Computers. In *The Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, January 1988.

[8] Foster I., Taylor S. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[9] Gabber E. VMMP: A Practical Tool for the Development for Portable and Efficient Programs for Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 304–317, July 1990.

[10] Kale L.V. Comparing the Performance of Two Dynamic Load Distribution Methods. In *International Conference on Parallel Processing*, August 1988.

[11] Kale L.V. The Chare Kernel Parallel Programming System Programming System. In *International Conference on Parallel Processing*, August 1990.

[12] Kale L.V., et. al. The Chare Kernel Programming Language Manual. internal report.

[13] Lin F.C.H, Keller R. Gradient Model: A Demand Driven Load Balancing Scheme. In *International Conference on Distributed Systems*, pages 329–336, 1986.

[14] Mullender S.J., Tanenbaum A. The Design of a Capability-Based Operating System. *Computer*, 289–300, March 1986.

[15] Ramkumar B., Kale L.V. A Chare Kernel Implementation of a Parallel Prolog Compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, March 1990.

[16] Saletore V. A. *Machine Independent Parallel Execution of Speculative Computations*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1991.

```
BranchOffice LoadBalance {
manager {
Init() {
   msg = (InitMsg ) Allocate(InitMsg-size);
   CreateBranch(BranchInit,msg); }
}

branch {
   int NumNeighbors;
   int NeighborsList[MaxNeighbors];
entry BranchInit: (message InitMsg *msg) {
   NumNeighbors = GetNumNeighbors(MyNodeID);
   GetListOfNeighbors(MyNodeID, NeighborsList); }
entry NeighborStatus: (message StatusMsg *msg) {
   ReceiveUpdateLoadStatus(msg); }
}
ScheduleChare(message) {
   Neighbor = LeastLoadedNeighbor(MyNodeID);
   SendUpdateLoadStatus(message);
   SendMsgBranch(message, Neighbor); /* BOC send */
}
PeriodicStatus()
   if (NeedToSendStatus)
      for each neighbor in NeighborsList
         message = Allocate(StatusMsg-size);
         SendUpdateLoadStatus(message);
         /* send neighbour my status */
         SendMsgBranch(NeighborStatus,message);
   /* call to to periodically check status */
   CallBocAfter(PeriodicStatus,LdbBocNum,INTERVAL);

}
```

Figure 6: Load Balancing As a Branch Office Chare