Information sharing mechanisms in parallel programs[*]

Laxmikant V. Kalé
Department of Computer Science,
University Of Illinois at Urbana–Champaign,
Urbana, Illinois 61801.
email: kale@cs.uiuc.edu

Amitabh B. Sinha
Department of Computer Science,
University Of Illinois at Urbana–Champaign,
Urbana, Illinois 61801
email: sinha@cs.uiuc.edu

Abstract

Most parallel programming models provide a single generic mode in which processes can exchange information with each other. However, empirical observation of parallel programs suggest that processes share data in a few distinct and specific modes. We argue that such modes should be identified and explicitly supported in parallel languages and their associated models. The paper describes a set of information sharing abstractions that have been identified and implemented in the parallel programming language Charm. It can be seen that using these abstractions leads to improved clarity and expressiveness of user programs. In addition, the specificity provided by these abstractions can be exploited at compile-time and at run-time to provide the user with highly refined performance feedback and intelligent debugging tools.

1 Introduction

A typical parallel computation can be characterized as a collection of processes running on multiple processors. Depending on the programming model and language, it may have just one or many processes on each processor. As they are part of a single computation, they often have to exchange data with each other. The mechanisms used for exchanging data is the topic of this paper.

One of the most popular information sharing mechanisms is a shared variable. Two or more processes may exchange information by setting and reading the same shared variable. This model offers great simplicity as it appears to extend the sequential programming model in a natural manner, e.g., ... However information exchange through shared variables suffers from one major drawback: the difficulty of efficient implementation on large parallel machines. Shared variables can

[*]This research was supported in part by the National Science Foundation grants CCR-90-07195 and CCR-91-06608.

be implemented efficiently on small parallel machines which physically share memory across a bus. Such machines can provide hardware support for a single global address space. Many large scale machines available today, such as Intel i860, NCUBE, and CM-5, include hundreds of processors and do not provide a single global address space. Implementing shared variables on such machines is difficult and inefficient.

Messages provide another important means of exchanging information between processes. Messages, containing necessary information, can be sent from a "sender" process to a known "receiver" process. Most commercial distributed memory machines provide hardware support for message passing, so this mechanism to exchange information can be easily implemented. However message passing as a means of exchanging information can suffer from the following drawbacks:

1. Inadequacy: Message passing is sometime inadequate as a means of exchanging information, e.g., in order for a message to be sent the identity of the receiving process must be known. This can be a severe restriction in many cases, where such information may not be easily available.

2. Expressiveness: In some cases, messages can be used to implement a specific information exchange mechanism. However, this implementation might be cumbersome. E.g., message cannot easily express the sharing of read-only information (information that is accessed only in the read mode after being initialized) between multiple processes on the same processor.

In general, the problem with a single generic means of information exchange is that all modes of information exchange in an application must be implemented using the single universal mechanism. As a result, there can be a loss in expressiveness. In addition, when the user is writing the program he/she knows the nature of information exchange in the program. If all modes of information sharing were implemented using a generic mechanism, then this knowledge about the nature of information sharing would be lost. A compiler-writer might then have to automatically detect various modes of information sharing, imperfectly and conservatively, in order to produce more efficient object code.

There exist other mechanisms to exchange information amongst parallel processes. The information sharing mechanisms provided by Actors [1], Strand [2], and Linda [3] all suffer from the same problem: they are each only a universal information exchange mechanism. In an Actor program, messages are the sole means of information exchange. An actor can be used to implement the read-only mode of information exchange. A good Actor compiler may even detect that the actor implements read-only information exchange. However it would be more intuitive and convenient for the programmer to specify that the mode of information exchange needed was read-only, rather than trying to fit that into the single mode of information exchange. Similar problems occur with Linda, where information exchange is done through depositing data into a shared tuple space.

We have identified and implemented many specific modes of information exchange in the parallel programming language Charm. We will demonstrate that using these abstractions leads to improved clarity and expressiveness of user programs. In addition, the specificity provided by these abstractions can be exploited at compile-time and at run-time to provide the user with highly refined performance feedback and intelligent debugging tools.

In Section 2, we briefly describe Charm [4, 5, 6]. In Section 3, we present some specific modes of information sharing that are available in Charm. In Section 5, we discuss how these specific modes of information sharing have been implemented on different parallel machines. Section 4 contains example of the usage of these specific information sharing mechanisms. In Section 6, we discuss how the information contained in these specific modes of information sharing can be used to provide better performance analysis and debugging tools for Charm programs.

2 Basic language features of Charm

Charm is a machine independent parallel programming language. Programs written in Charm run unchanged on shared memory machines including Encore Multimax and Sequent Symmetry, non-shared memory machines including Intel i860 and NCUBE/2, UNIX based networks of workstations including a network of IBM RISC workstations, and any UNIX based uniprocessor machine.

```
module Module1 {

    Type declarations
    Message definitions, information sharing declarations
    chare main {
        Local variable declarations

        entry CharmInit: C-code-block
        [entry QUIESCENCE: C-code-block]
        Other Entry points, functions and their code
    }
    chare Example1 {
        Local variable declarations

        /* Entry Point Definitions */
        entry EP1: (message MESSAGE_TYPE1 *msgPtr)
            C-code-block
        ..
        entry EPn: (message MESSAGE_TYPEn *msgPtr)
            C-code-block

        /* Local Function Definitions */
        private Function1(..)
            C-code-block
        ..
        private Functionm(..)
            C-code-block
    }
    BranchOffice Chares and other function declarations
}
```

Figure 1: **Syntax of a CHARM program**

The syntax of a Charm program is shown in Figure 1. A Charm program consists of the definition of messages, chares and branch office chares.

The basic unit of computation in Charm is a *chare*. A chare's *definition*, shown in Figure 1, consists of a data area and entry functions that can access the data area. A chare *instance* can be created dynamically using the *CreateChare* system call. This call returns immediately; sometime in the future the system will actually create and schedule the new chare. Each chare instance has a unique address.

Every Charm program must have a main chare definition. The main chare definition is like any other chare definition except that it must contain an *CharmInit* entry point, in addition to other application specific entry points. Program execution begins at the *CharmInit* entry point. In this paper, we refer to creation of variables or processes as static if it occurs inside the *CharmInit* entry point; all other creation is dynamic.

```
message msg_name {
    int x;
    float y[100];
    varSize float y[];
} MSG_NAME;
```

Figure 2: **Example of a Message Declaration**

The basic information exchange mechanism in Charm is a message. A message can be allocated using the *CkAllocMsg* call. Figure 2 shows an example of a message declaration. In some messages fields may need to variable sized arrays. Charm provides the *varSize* mechanism to define one dimensional variable sized arrays — the size of the array needs to be specified when the message is being allocated.

In general, there may be messages which cannot be easily handled by one dimensional arrays. In such cases, Charm allows for more complex message definitions, which may include dynamically allocated spaces. A message with pointers to dynamically allocated memory blocks may need to be packed (into a contiguous array) if it crosses memory boundaries. For such messages, users need to provide a pair of *pack* and *unpack* functions. The *pack* function takes a message with pointers to dynamically allocated memory, and "packs" it up in a single character array; the *unpack* function takes the contiguous character array and "unpacks" it to generate the message with pointers.

The execution model of Charm is message-driven, i.e., an entry point is executed when a message addressed to it arrives at a processor. Entry functions in a particular chare instance can be executed by addressing a message to the desired entry function of the chare. Messages can be addressed to existing chares using the *SendMsg* system call.

Charm also provides another kind of process called a **b**ranch **o**ffice **c**hare (BOC). A BOC is a replicated chare: there exists a branch or copy of the chare on each processor. All the branches of a BOC are referred to by a unique identifier. This identifier is assigned when a BOC instance is created, and may be passed in messages to other chares. The definition of a BOC is similar to that of a chare, except that a BOC definition can have *public* functions. A *public* function can be called by other chares running on the same processor. Branch office chares can be statically or dynamically created using the *CreateBoc* call. Branches of a BOC can interact with each other using the *SendMsgBranch* system call. Branch office chares provide a means for sequential as well as parallel interfaces in a program.

3 Specific Information Sharing Abstractions

Our experience with parallel programs has shown that they tend to use only a limited variety of information sharing modes. The 'completely general' shared variable is rarely used because it cannot be implemented in an efficient and scalable fashion. Generic universal information exchange mechanisms (such as shared variables, messages, tuple-spaces etc.) force the user to implement different modes of information sharing in the sole mechanism available. Consequently, any information that the user might have had about the nature of information exchange is lost. Often the efficient implementation of these programs needs knowledge about the nature of information exchange. In such cases, the knowledge must be extracted from the program by compiler techniques, and this is often not possible.

We have identified and provided abstractions for specific modes of information sharing in parallel programs. These specific modes are implemented as efficiently as possible on a particular architecture, and they provide an efficient means for tasks in an application to share information. Currently, Charm provides five different kinds of shared variables: *read only*, *write once*, *accumulator*, *monotonic* and *distributed tables*. Other kinds of shared variables can be written as a combination of chares and branch office chares.

3.1 Read-only variable/message

In many computations, many processes need *read access* to data that is initialized at the beginning of the computation, and is not updated thereafter. This mode of information sharing can be specified using the read-only mechanism. Charm provides two kinds of read-only information sharing: read-only variables and read-only messages. Figure 3 shows how read-only variables and messages can be declared in a Charm program. The essential difference between a read-only variable and a read-only message is that the latter is treated like any ordinary message: it can contain variable size arrays and other dynamically allocated memory.

```
readonly Type readname;
readonly MsgType *readmsgname;
```

Figure 3: Syntax of a read-only abstract data type

Read-only variables and messages are initialized in the *CharmInit* entry point using the *ReadInit* and *ReadMsgInit* calls. Chares and branch-office chares can access read-only variables and messages via the *ReadValue* call. This call simply returns the (fixed) value of the read-only variable or message.

Read-only variables are the only true variables in a Charm program — all other mechanisms for information sharing are accessed through an identifier. Chare or branch-office chares can access other mechanisms only if they know the identifier, either passed in a message or as a read-only variable.

3.2 Write-once variable

In some computations, read-only information is available only after the computation has proceeded for some time: the value is not available when the initialization phase of the program is being executed. In Charm, write-once variables provide this mode of information sharing. Write-once variables are the dynamic counterpart of read-only variables. A **write-once** variable is created and initialized any time (and from any chare) during the parallel computation. Once created, its value cannot be changed. The creation is done via a non-blocking call *WriteOnce(msg, entryPoint, CharelD)* 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 unique name can be passed to other chares and branch office chares. They can access the variable by calling *DerefWriteOnce(name)*, which returns the value of the write-once variable.

3.3 Accumulator variable

In many computations, a variable is needed to count the number of occurrences of an event, or a property, etc. Such a variable is updated by a monotonic function. Charm provides this mode of information sharing through the accumulator variable. Figure 4 shows the syntax of an accumulator definition. An instance of an accumulator can be created using the *CreateAcc* call. This call can be made statically (inside CharmInit) or dynamically — if it is created statically the identity of the variable is available immediately, but if it is created dynamically the identity is returned to a specified address.

```
accumulator acc_type {
    Message_Type *acc;
    Message_Type *initfn ()
        C-code-block
    addfn ()
        C-code-block
    combinefn ()
        C-code-block
} ACC_TYPE;
```

Figure 4: Syntax of an accumulator definition

The accumulator abstract data type has associated with it a message containing the data area of the accumulator data type, an initialization function (that is used to set up the message) and two user defined commutative-associative operators.

The system is free to maintain multiple copies of the accumulator variable: in some cases there may be one copy per processor, while in other cases a few processors might share a copy. The initialization function, **initfn**, is called, possibly on multiple processors, upon invocation of the *CreateAcc* call). The first operator, **addfn**, *adds* to the accumulator variable in some user defined fashion, while maintaining commutativity and associativity. The second operator, **combinefn**, takes two accumulator variables as operands and *combines* those variables element by element, again in a commutative-associative manner.

The accumulator variable can be modified only via the *Accumulate* call, which adds a given value to the accumulator. The destructive read is performed with the *CollectValue* (non-blocking) call. This results in the eventual transmission of the value of the accumulator to a specified address.

3.4 Monotonic variable

In some computations, processes need access to a variable which increases monotonically with the application of an idempotent and monotonic function. Such a variable is typically used in branch&bound computations.

Charm provides this mode of specific information sharing with a monotonic variable. Figure 5 shows the syntax of a definition of a monotonic variable. An instance of a monotonic variable can be created using the *CreateMono* call. This call results in the function **initfn** being called; *initfn* initializes and returns the initial value of the monotonic variable. Monotonic variables can be created either statically (inside CharmInit) or dynamically — if it is created statically the identity of the variable is available immediately, but if it is created dynamically the identity is returned to a specified address.

Subsequent updates to the monotonic variable can be carried out through the *NewValue* call. This results in the corresponding **updatefn** function being called. The function **updatefn** must be a monotonic and idempotent function for the domain over which the monotonic variable is defined. The following is guaranteed for any *NewValue* call:

1. The value supplied in a *NewValue* call replaces (using the **updatefn**) the old value, if it is "greater" than the old value.
2. The updated value (if at all) after a *NewValue* call will be eventually propagated to all other processors.

The (approximate) current value of a monotonic variable can be read by any chare at any time using the *MonoValue* call. The value returned by the *MonoValue* call will satisfy the following properties:

1. The value will be true, i.e., it will either be the value at the time of initialization, or provided thereafter by a *NewValue* call.
2. The value returned will be the greatest value provided by any *NewValue* call on the same processor.

```
monotonic mono_type {
    Message_Type *msg;
    Message_Type *initfn ()
        C-code-block
    updatefn ()
        C-code-block
} MONO_TYPE;
```

Figure 5: Syntax of a monotonic variable declaration

3.5 Distributed Table

In many applications, data can be split into many portions, and each portion can be accessed by a subset of processes in the system. Further, the subset of processes that access a portion of the table may not be pre-determinable. In other applications, processes that do not know each other's identity may need to share information. Charm provides distributed tables as a means of sharing information in these modes. The syntax of a distributed table definition appears below. The data in an entry in the table is a message. Like all other messages, data items in a table can have dynamically allocated areas either declared explicitly by the user or through Charm constructs.

```
table table_type {
    Message_Type *msg;
    hash_to_pe()
        C-code-block
    hash_to_index()
        C-code-block
    updatefn ()
        C-code-block
} TABLE_TYPE;
```

Figure 6: Structure of a distributed table abstract data type

A **distributed table** consists of a set of entries, each with a key part and a data part. The key uniquely identifies its corresponding data. There are various asynchronous access and update operations on entries in distributed tables. An entry can be inserted using the *Insert* call. The user can search for a particular entry (using its key) using the *Find* call. And the user may delete an entry using the *Delete* call. The current implementation of distributed tables in Charm is a restricted version of this more general formulation: the hash function is specified by the system and the data is a string of characters.

A distributed table also provides a good distributed interface between two components of a parallel program. In sequential programming, data exchange between two phases of computation in an application is achieved through a sequential point of transfer, e.g., parameters in a function call. Such a mechanism of exchanging data between two phases of a parallel application can pose to be a bottleneck, e.g., suppose the first phase of an application is a matrix multiplication, the result of which is used in the second phase. If the result matrix were computed in a distributed fashion in the first phase and used in a distributed fashion in the second phase, then in order to transfer the result matrix, it must be collected on one processor after the first phase, then transferred to the second phase and then re-distributed. This collection and re-distribution can be avoided by exchanging data in a distributed manner — distributed tables are a suitable mechanism to achieve this exchange.

4 Examples of usage of specific information sharing

In this section, we will illustrate the usage of specific information sharing mechanisms via various example programs. The general guidelines that one follows in deciding which specifically shared variable must be used in a program are:

- If the information needs to be passed to one process whose identity is known, then a message should be used.
- If the information needs to be passed to one or more processes whose identity is not known, then a distributed table should be used.
- If the information needs to be shared among most of the processes in the system, and it does not get accessed in the write-mode after being initialized, then a read-only or a write-once variable should be used.

4.1 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) [7] is a typical example of an optimization problem solved using branch&bound techniques. In this problem a salesman must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with them (if $i = j$, then C_{ij} is assumed to be of infinite cost).

In the branch&bound scheme originally proposed by Little, et. al. [8]. one starts with an initial partial solution, a cost function (C) and an infinite upper bound. A partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. The cost function provides for each partial solution a lower bound on the cost of any solution found by extending the partial solution. The cost function is monotonic, i.e., if S_1 and S_2 are partial solutions and S_2 is obtained by extending S_1, then $C(S_1) <= C(S_2)$. Two new partial solutions are obtained from a partial solution by including and excluding the "best" edge (determined using some selection criterion) not in the partial solution. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. The upper bound is updated whenever a solution is reached. Each partial solution is a node in a branch&bound tree where an edge exists from a partial solution S_1 to a partial solution S_2, if S_2 is obtained by existing S_1.

In the Charm implementation of the branch&bound solution of TSP, shown in Figure 7, each partial solution is represented by the chare: start. A monotonic variable, identified by mono_bound,

```
message { int nodes; } ACC_MSG;
message { int bound; } MONO_MSG;
readonly int acc_nodes;
readonly int mono_bound;

accumulator { ACC_MSG *msg;
    ACC_MSG * initfn(data) ACC_MSG *data;
    { msg = (ACC_MSG *) CkAllocMsg(ACC_MSG);
    msg->nodes = data->nodes; return(msg); }
    addfn () { (msg->nodes)++; }
    combinefn(b) ACC_MSG *b; { (msg->nodes) += (b->nodes); }
} ACC_INT;
monotonic { MONO_MSG *msg;
    MONO_MSG * initfn(data) MONO_MSG *data;
    { msg = (MONO_MSG *) CkAllocMsg(MONO_MSG);
    msg->bound = data->bound; return(msg); }
    updatefn(new) MONO_MSG *new; {
    if (msg->bound > new->bound) {
    msg->bound = new->bound; return(1); }
    return(0); } } MONO_INT ;
chare main {
    entry CharmInit: {
        read_in_matrix(matrix, N);
        cost = reduce_matrix(matrix, N); ReadInit(matrix);
        mono_bound = CreateMono(MONO_INT, mono_msg); ReadInit(mono_bound);
        acc_nodes = CreateAcc(ACC_INT, acc_msg); ReadInit(acc_nodes);
        msg = (MSG1 *) CkAllocPrioMsg(MSG1, sizeof(int));
        msg->cost = cost; Reset(msg);
        CreateChare(start, start@LEAF, msg); }
    entry QUIESCENCE: { CollectValue(ReadValue(acc_nodes), MAINEP1, &myself); }
    entry RECEIVE : (message SolnMsg *msg) { print_out_solution(msg); } }
chare start {
    entry LEAF : (message MSG1 *msg) {
        n = ReadValue(N);
        DETERMINE_MATRIX(msg, ReadValue(matrix), A, n);
        cost = SOLVE_RELAXATION(A, soln, n, rows, columns);
        Accumulate(ReadValue(acc_nodes), addfn());
        if (cost < ((MONO_MSG *) MonoValue(ReadValue(mono_bound))->bound)
        if (isSolution(msg, soln, n)) {
            tmsg->bound = cost;
            NewValue(mono_bound, updatefn(tmsg));
            SendMsg(main@RECEIVE, soln, main_chare); }
        else BRANCH_OUT(msg, A, n, cost, rows, columns); } }
```

Figure 7: **Traveling Salesman Problem**

is used to maintain the upper bound. The cost matrix, matrix, is represented as a read-only variable, since the information is available initially and does not change thereafter. In speculative computations, such as branch&bound, information that is useful for performance analysis is the number of branch&bound nodes created in an execution — the greater the number of nodes as compared to a sequential execution of the code, the larger the extent of speculative computation. An accumulator variable, identified by acc_nodes, is used to determine the number of nodes in the branch&bound tree.

4.2 Bi-directional search

In a search problem, one starts with an initial state and attempts to determine a path towards a goal state through a set of operations which define transition from one state to the next. In some search problems, the goal layer is determined by a property it satisfies, e.g., N-queens problem where the goal state is one in which N queens can be arranged in a non-attacking position. In some other search problems, the goal layer is known, e.g., the goal state in the Rubik's cube problem is one in which each of the cube's sides has one color. In such cases, the strategy often adopted is to first conduct a "backward" search to determine the goal layer (set of states from which the solution can be reached), and then conduct a "forward" search from the initial state towards the goal layer.

During the forward search, for each state a check must be performed to determine whether the state belongs to the goal layer. One of the strategies to reduce the communication involved in checking whether a state belongs to the goal layer could be to replicate the goal layer on each processor — in this approach, all queries could be satisfied locally.

Once the goal layer has been created, it doesn't need to be modified. This suggests that the goal layer is a "read-only" variable. However it is created dynamically, i.e., some parallel computation needs to be performed before the goal layer can be determined. In Charm, this mode of information exchange can be captured exactly by the write-once variable. Once the goal layer is created as a write-once variable, any processor would have a constant access time to the goal layer.

The pseudo-code for the Charm program for bi-directional search is shown in Figure 8. The backward search is carried out by the chare: backward. Each state in the goal layer is communicated to the main chare. After all the states in the goal layer have been computed, quiescence is automatically detected by the system, and the write-once variable representing the goal layer is created. After the identifier for the write-once variable is available, forward search is initiated. During the forward search, checks to determine whether a solution is in the goal layer is carried out by de-referencing the write-once variable.

4.3 Matrix Multiplication

There are many different matrix multiplication algorithms. We illustrate the use of distributed tables with one such algorithm. Matrices A and B need to be multiplied. The result matrix can be divided into rectangular blocks, and each block can be computed in parallel. The computation of each block may need multiple rows of A and multiple columns of B.

```
chare main {
    entry CharmInit: {
        read_in_initial_state();
        read_in_goal_state();
        Create_Goal_State(msg);
        CreateChare(backward, backward@start, msg); }
    entry backward: (message STATE_MSG *msg) { store(goal_layer, msg); }
    entry QUIESCENCE: { CreateWriteOnce(goal_layer, main@forward, main_chare); }
    entry forward : (message WriteOnceID *msg) {
        Add_Initial_State(msg); CreateChare(forward, forward@start, msg) }
chare backward {
    entry start : (message STATE_MSG *msg) {
        if (goal_layer(msg)) SendMsg(main@backward, msg, main_chare);
        else {
            CreateChare(backward, backward@start, backward_left_child(msg));
            CreateChare(backward, backward@start, backward_right_child(msg)); } } }
chare forward {
    entry start : (message STATE_MSG *msg) {
        if (belongs_goal_layer(DerefWriteOnce(name),msg) print_out_solution(msg);
        else {
        CreateChare(forward, forward@start, forward_left_child(msg));
        CreateChare(forward, forward@start, forward_right_child(msg)); } } }
```

Figure 8: **Bi-directional search**

In the Charm implementation, shown in Figure 9, the computation of each rectangular block is performed by a chare. Since Charm programs can be executed on a network of workstations, where each processor may have different execution speeds, there is a possibility that even blocks of the same size may take different amounts of time to be computed. Therefore parallel processes (chares) created to compute different blocks must be dynamically load balanced. A dynamic load balancing strategy makes it to difficult to estimate which rows of A and which columns of B are needed on a particular processor. The distributed table abstraction can be used to solve this problem. The rows of A and the columns of B are entries in distributed tables, row_table and col_table, respectively. The chare start is used to compute a block of the result matrix. When it is first created (multiple entry point), the chare uses the *Find* call to get the necessary rows and columns. Computation of entries in the particular block of the result matrix can be handled appropriately as a new row of A or a new column of B arrives, e.g., for each new row of A the chare must carry out a scalar product with each column of B that it already possesses.

5 Implementation of information sharing abstractions

In this section, we describe the implementation on different parallel machines of the five information sharing abstractions described in Section 3.

```
chare main {
    entry CharmInit: {
        read_in_matrix(matrix);
        for (i=0; i<rows; i++) {
            get_row(matrix, i, row); Insert(row_table, i, row); }
        for (i=0; i<cols; i++) {
            get_column(matrix, i,col); Insert(col_table, i, col); }
        for (i=0; i<rows; i+=row_grain)
        for (j=0; j<cols; j+=col_grain) {
            msg->row_index = i; msg->col_index = j;
            CreateChare(start, start@multiply, msg); } }
    entry QUIESCENCE: { CkExit(); }
}
chare start {
    entry multiply: (message MSG *msg) {
        recd_rows = recd_cols = 0;
        for (i=0; i<row_grain; i++)
            Find(row_table, msg->row_index+i, row_continue, myself);
        for (i=0; i<col_grain; i++)
            Find(col_table, msg->col_index+1, col_continue, myself); }
    entry row_continue: (message ARRAY_MSG *msg) {
        recd_rows++; store_row(msg->row, row);
        for (i=0; i<recd_cols; i++) dot_product(msg, col);
        PrivateCall(continue()); }
    entry col_continue: (message ARRAY_MSG *msg) {
        recd_cols++; store_col(msg->col, col);
        for (i=0; i<recd_rows; i++) dot_product(msg, row);
        PrivateCall(continue()); }
    private continue() {
        if ((recd_rows == row_grain) && (recd_cols == col_grain))
            Insert(result_table, row*cols+col, compute_block(row, col)); }
}
```

Figure 9: **Matrix Multiplication Problem**

5.1 Shared memory machines

On *shared memory machines*, with a small number of processors, each information sharing abstraction is implemented as a shared variable. Read-only variables have no locks to control access, since they will be accessed only in the read-only mode. Accumulators and monotonic variables have an associated lock, and operations on them are performed in a mutually exclusive manner using locks. Write-once variables are also shared variables without a lock to control access; however in order to establish a unique name for a write-once variable processors need a lock for synchronization. Distributed 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.

Since write-access to an accumulator might be very often (it is read only once) a more efficient implementation would be one in which each processor had a local copy of the variable. When the variable is finally read the processors use locks to add the local copies together. This scheme might be more suitable for larger shared memory machines also.

5.2 Nonshared memory machines

On *nonshared memory machines*, a local copy of each read-only, write-once, monotonic and accumulator variable is maintained on each processor. Write-once, monotonic, accumulator and distributed tables are implemented as branch-office chares. For the latter three, the local copy is maintained by the corresponding branch of the branch-office chare. The entires in a distributed table are divided amongst the set of available processors, and the corresponding branch of the branch-office chare maintains a processor's part of the table.

Write Once variables are initialized by the *CreateWriteOnce* call. A copy of the variable is first sent to the branch on processor 0 of the corresponding BOC. This branch assigns the variable a unique index, which serves as the identifier for the write-once variable, and then broadcasts the value and identifier 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 branch on processor 0 (along a spanning tree rooted at processor 0 to avoid bottlenecks) that it has created the variable. When it has received an acknowledgement message from all the nodes, the root branch sends the identifier of the write once variable to the specified address. A write once variable can be read by 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 (with the corresponding `updatefn`), 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 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 *addfn* 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 branch on processor 0 up a spanning tree, rooted at processor 0, on the nodes. Branches use the *combinefn* function to combine values they receive from their children nodes. The branch on the root processor communicates the final value to the supplied chare.

Updates on entries in a distributed 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 distributed 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.

6 Parallel programming tools

A parallel program has its own characteristic attributes, such as the nature of shared variable access, degree of parallelism, placement of tasks, synchronization between tasks, etc. Techniques for characterization of parallel algorithms have been studied before. E.g., Jamieson [13] used the characteristics of parallel algorithm, in conjunction with the characteristics of parallel architectures, to provide an understanding of how well the algorithm is suited to different architectures. We believe that a knowledge of the characteristics of a parallel program can help make the task of debugging and performance analysis more focussed. The nature of information sharing in a parallel program is a crucial characteristic of a parallel program, and in this section we shall show how an understanding of the nature and modes of information exchange in a program can aid performance analysis of the program.

The usage of one of the five information sharing mechanisms in an application program, provides some insight into the nature of information exchange in the program. This insight can be utilized to provide a more accurate analysis of the performance of programs. Some of the performance concerns that could be addressed when one knows the nature of information sharing (through specifically shared variables) in a Charm program are:

- If distributed tables were being used, then it would be necessary to check whether the distribution of keys and access of table entries were uniform over all processors.
- Monotonic variables are often used in speculative computations, where the speculative component could depend to a large extent on the value of the monotonic variable. In such cases, it would be useful to provide the user with information about the speculative component of computation and when updates were made to the monotonic variable.
- If some information is represented as a distributed table, and it is accessed very frequently by many different processors, a performance analysis could suggest that the data be made write-once.

7 Conclusion

In this paper, we have presented five specific means of information sharing. These mechanisms add to the expressive power of any language. They have been implemented on different parallel machines for Charm.

In addition, knowledge about the nature of information exchange in a program can provide valuable insight into the performance of a program.

References

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

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

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

[4] Kale L.V. The Design Philosophy of the Chare Kernel Parallel Programming System. Technical Report UIUCDCS-R-89-1555, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1989.

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

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

[7] Edward W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[8] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[9] M. Bellmore and G. Nemhauser. The traveling salesman problem: a survey. *Operations Research*, 16:538–558, 1968.

[10] M. Held and R. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[11] B. W. Wah, G. Li, and C. Yu. Multiprocessing of combinatorial search problems. In V. Kumar, P. S. Gopalakrishnan, and L. N. Kamal, editors, *Parallel Algorithms for Machine Intelligence and Vision.* Springer-Verlag, 1990.

[12] B. Monien and O. Vornberger. Parallel processing of combinatorial search trees. *Proceedings International Workshop on Parallel Algorithms and Architectures*, Math. Research Nr. 38, Akadmie-Verlag, Berlin, 1987.

[13] Leah H. Jamieson. Characterizing parallel algorithms. In Leah H. Jamieson, Dennis Gannon, and Robert J. Douglass, editors, *The characteristics of parallel algorithms.* The MIT Press, 1987.