

High Level Support For Divide-and-Conquer Parallelism

Attila Gürsoy

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

L.V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

In this paper we present a simple language based on C for expressing divide-and-conquer computations. The “language” consists of a few simple extensions to C. It allows for many variations in the standard divide-and-conquer paradigm. It is implemented using the Chare Kernel parallel programming system. The Chare Kernel supports dynamic creation of work with dynamic load balancing strategies, and machine independent execution. As a result, implementation of languages and systems such as that described in this paper is simplified significantly. A translator translates divide-and-conquer programs to Chare Kernel programs, handling details of synchronization and communication automatically. The design of the language is presented, followed by a description of its implementation, and performance results on many parallel machines, including NCUBE/two, iPSC/2, and the Sequent symmetry. User programs do not have to be changed to run on any of these machines.

1 Introduction

The dramatic advances in parallel computer architectures have led to an expectation that most computation-intensive problems will be routinely speeded up using parallel processing. Although many commercial systems have appeared in the market, programming them to meet this expectation is still a challenging task. Parallel programming is obviously more difficult than sequential programming. It is necessary to simplify and support the task of writing parallel applications, and also to ensure that the investment in parallel software is protected through architectural advances and new generation of parallel machines.

The Chare Kernel [11] is a machine-independent MIMD parallel programming system that is aimed at this objective. The system provides an explicitly parallel language - which uses C [12] as its base lan-

guage. The system takes the responsibility of dynamic load balancing, scheduling (when to execute an eligible sub-computation), resource management, and provides machine dependent expression.

The base language provided by the Chare Kernel is a general purpose parallel programming language. However, with the research and experience of the past years in parallel programming, a few specific parallel paradigms have been identified - such as data-parallel programming, compute-communicate-aggregate [15], dynamic programming, etc. The approach we have taken is that of developing sub-languages on top of the Chare Kernel that are suitable for each specific paradigm. The programmers can then use the language that is most effective for expressing their specific problem, translate it using a compiler to the chare kernel language, and run it on the MIMD machine of their choice. As the “languages” are often simply extensions of the chare kernel language, which itself can be thought of as an extension of C, the compilation is relatively straightforward. (We use the term “language” if an extension requires compilation, to distinguish from the case where an extension simply provides a library of system calls. The term “co-ordination languages”, used by Gelernter et al [5] can also be used to describe such languages.) Thus building such application specific tools is relatively easy with this approach.

In this paper, we describe a language to express the divide-and-conquer programs concisely and with ease. After the discussion of divide-and-conquer method in section 2, we give the description of the language in section 3, and we discuss a programming example to show how the system is used in section 4. The system is implemented on top of the Chare Kernel parallel programming system [11]. The Chare Kernel supports dynamic creation of medium grained tasks with dynamic load balancing strategies, and provides machine independence. It is a general purpose machine independent parallel programming system, which can be used to develop specific languages, such as the one discussed here, with relatively little effort. Thus the im-

plementation, discussed in Section 5, concerns mainly translating the user program into a Chare Kernel program. Performance on various shared and non shared memory machines, including Sequent Symmetry, intel's iPSC/2, and NCUBE/two are described in section 6, which is followed by a section summarizing the paper.

2 Divide-and-conquer paradigm

Divide-and-conquer is a naturally parallel paradigm and is considered to be a broadly applicable one [15, 1, 2]. In a typical divide-and-conquer computation, a computational problem is broken down into smaller subproblems, some of which may be of the same type (of lesser complexity) as the original problem itself. This process is continued recursively as many times as necessary. When the sub-computations are simple enough they are solved directly without further subdivision. The results from the subcomputations are passed to the parents which created them. The parent node composes the solutions to subproblems to form the solution to itself, which it then sends to its parent.

A few variations on this theme are also possible within the paradigm. In search-type problems the composition-of-subproblems is either trivial or absent (if solutions are directly printed). In some other domains, solutions to some sub-problems may lead to creation of new sub-problems which must be solved. This can happen, for example, due to a data or control dependency among the sub-problems.

Many problems such as combinatorial optimizations, searches [13], many problems in computational geometry, in numerical methods [10, 16], and problem-reduction in AI are formulated naturally as divide-and-conquer computations. Stout [19] identifies divide-and-conquer as an important strategy for higher level image processing algorithms, and stresses the significance of software support for expressing parallelism. Finkel [8] has implemented a software package that supports tree computations. It is a library of services that are linked to user supplied code. Similar approaches have been discussed in [3, 4, 17]. Gabber [9] discusses implementation of a software system that supports divide-and-conquer paradigm. Its implementation on nonshared memory machines has not been completed yet, and the author states that it is a challenging task. Also, neither approach supports arbitrary dependencies among subproblems.

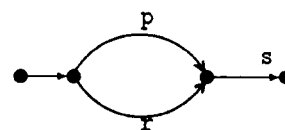


Figure 1: A simple DJG

3 Language definition

A divide-and-conquer program is expressed as a set of node definitions along with usual C functions. An instance of a node definition corresponds to a node in the computation tree. A node can also be visualized as a Data Join Graph (DJG). A DJG is a dependency graph where edges represent subcomputations and vertices represent synchronization points. A subcomputation originating from a vertex can start after all immediate predecessor subcomputations have been completed. For example, subcomputation labeled as *s*, in Figure 1, can begin execution after *p* and *r* have completed, and can receive data from them. A node definition is expressed in C syntax with a few extensions. It has a number of components (similar to fork/join mechanism [7]) to conveniently represent a DJG and the data it operates on. A BNF-like definition of the node syntax is shown Figure 2.

A simple toy example of node declaration is shown in Figure 3. Different components of the node definition are described below.

3.1 Data declarations

in : { *declaration-list* }

It specifies the formal parameters to be received by value.

out : { *declaration-list* }

It specifies the formal parameters to be sent to the parent instance by value.

node *node-declaration-list*

Each subcomputation (or edge in the DJG) should have a distinct label to differentiate it from others. *node-declaration-list* declares all the labels used in the node, and specifies the node type each label refers to. Input and output parameters of a subcomputation are accessed through the pointers *label.in*, *label.out* respectively. A node can access its own input and output values through the pointers called *in* and *out*.

cond *cond-declaration-list*

Any *condition variable*, that are used in the

```

node ::=
  node node-name {
    in : { declaration-list }
    out: { declaration-list }
    node node-declaration-list ;
    cond cond-declaration-list ;
    local-declaration-list ;
    init-block
    when-block*
  }

init-block ::= init : { init-body }
when-block ::= when condition-list : { when-body }
node-declaration-list ::= node-name : label |
  node-name : label , node-declaration-list
cond-declaration-list ::= label |
  label , cond-declaration-list
condition-list ::= condition | condition, condition-list
condition ::= identifier | identifier [ range-list ]
range-list ::= range | range , range-list
range ::= intconstant | intconstant - intconstant
node-name ::= identifier
label ::= identifier | identifier [intconst ]
local-declaration-list ::= declaration-list
init-body ::= statement-list
when-body ::= statement-list
declaration-list is a list of C
  declaration statements
statement-list is a list of C statements

```

Figure 2: Syntax of a node

condition-list's of *when-block*'s, or in the **set** statements (see below) must be declared here.

3.2 Blocks

init : { *init-body* }

When an instance of a node is created, the *init-body* is executed first. It usually contains the initialization code and a termination check which decide whether to subdivide the problem further or solve it directly. It spawns a set of subcomputations, if necessary. After completing *init-body*, node suspends itself until one of the *when-block*'s is satisfied.

when *condition-list* : { *when-body* }

The labels in the *condition-list* refer to subcomputations or condition variables. In other words,

```

node fib {
  in : {int n;}
  out: {int result;}
  node fib : p, fib : q;

  init : {
    if (in->n < 2) {
      out->result = in->n;
      send result;
    }
    else {
      p.in->n = in->n - 1;
      q.in->n = in->n - 2;
      fire p;
      fire q;
    }
  }

  when p,q : {
    out->result =
      p.out->result + q.out->result;
    send result;
  }
}

```

Figure 3: Node definition to compute fibonacci numbers

when-block is a synchronization point (vertex) in the DJG, and *condition-list* represents all incoming edges. The *condition-list* of all *when-block*'s of a node is checked when one of its subcomputations finishes, and also after the execution of any *when-block*. If all the subcomputations listed in a *condition-list* have been completed and the condition variables listed in the *condition-list* have been set (by the **set** instruction), then the corresponding *when-body* is executed. If more than one *when-block* are satisfied, they are executed sequentially with no particular order.

3.3 Statements

fire

The syntax of a **fire** statement is:

fire *label*;

It creates an instance of the node which is associated with *label*. Before invoking a **fire** statement, it is necessary to assign the required input values to *label.in*. With the execution of the **fire** statement, control of the data area pointed to by *label.in* is transferred to the subcomputation, and it should not be accessed subsequently. Similarly, the data pointed by *label.out* is valid only after the subcomputation *label* has been completed. Therefore, *label.out* should be used in only proper *when-block*'s. Nodes may be indexed

for convenience. For example, Figure 4-a shows the code for firing 10 subproblems. Instead of writing 10 fire statements with 10 distinct labels, the code can use a simple loop to fire them using the indexed label *p*.

send result

A node sends its output, (data pointed by *out*), to its parent node with the statement:

send result;

In addition to that, memory space allocated to all responses that are received from subcomputations are released, and execution of that instance is terminated.

```
node f:p[10]
...
for(i=0;i<10;i++)
    fire p[i];
(a)
```

```
node f:p[10];
...
for(i=0;i<10;i++)
    if (condition(i))
        fire p[i];
    else
        set p[i];
...
when p[0-9] :{...}
(b)
```

Figure 4: Variable number of subcomputations

set

set label;

A label listed in *condition-list* is accepted as true if it is set by a **set** instruction. A label may refer to a subcomputation or a condition. Node labels are allowed to be set explicitly in order to permit variable number of subcomputations to be activated. In Figure 4-b, the for-loop activates a subset of subcomputations *p*[0]...*p*[9]. Assume that it is not known at compile time which subset of *p*'s will be activated. How can specify conditions for a *when-block* that should be activated when all the *fired* instances of a node have completed? A solution to this problem utilizes **set** instruction as follows: the *condition-list* of the *when-block* includes all labels that can be potentially activated. However, the labels that

are not actually activated are set explicitly in the **else** part to satisfy the *condition-list*.

A **set** statement is useful for control of the order of *when-block*'s. Consider the example in Figure 5-a: Let's assume that *p* has been completed already, and the corresponding *when-body* has been executed. When *q* is completed, both the second *when-block* and the third one are eligible for execution. In order to use the new value of *b*, the third *when-block* should wait for completion of the second one. Correct order of execution is achieved by utilizing condition variables as in Figure 5-b.

```
Ambiguous version:
/* a and b shared */
when p : { a = ...}
when q : { b = ...}
when p,q : { f(a,b) ....}
(a)
```

```
Correct Version:
/* a and b shared */
cond c[2];
when p : { a=..; set c[0];}
when q : { b=..; set c[1];}
when c[0,1] : {f(a,b) ...}
(b)
```

Figure 5: Control of order of when-blocks

3.4 Main node

The source program should have one specially designated node named **main**. The definition of this node does not have in or out declarations since it is the root of the computation tree. In the *init-block* of the main node, readonly variables are initialized. Read-only variables can be accessed from any other node.

4 A parallel programming example

In this section, we consider the problem of multiplying two $n \times n$ matrices. A simple divide-and-conquer strategy for matrix multiplication is as follows: Let *A* and *B* be two $n \times n$ matrices. The product matrix $C = A \times B$ can be computed by decomposing *A* and *B* into submatrices of size $n/2 \times n/2$ and then computing multiplication of those submatrices recursively

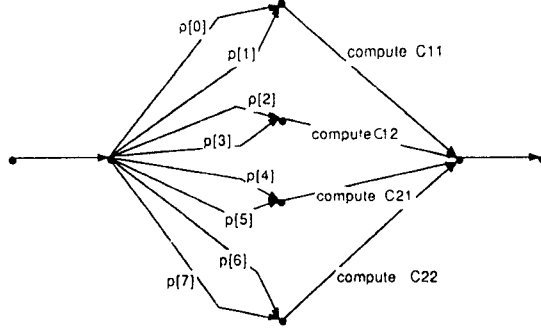


Figure 6: DJG of the matrix multiplication

according to the following equations:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

The formulation can be optimized further to encode Strassen's algorithm [1], which creates 7 (instead of 8) subproblems. As the purpose in this paper is to illustrate the language, we will stay with this simple formulation.

Figure 6 shows the data dependencies in the matrix multiplication algorithm. The corresponding node definition is given in Figure 8.

Matrices A and B are declared as readonly so they are shared among nodes. The input to each node consists of the size and the row and column coordinates of the left-upper corner of the submatrices to be multiplied. As the code at the *init-block* shows, grain-size control is used to stop further division of the problem. If size of matrices gets smaller than a threshold value, multiplication is carried out sequentially. Otherwise, matrices are divided into smaller blocks. If the average granularity (defined as the sequential execution time / total number of node instances) is too small, the overhead due to creation of large number of nodes and messages causes performance degradation. If it is too large then potential parallelism cannot be exploited. In divide-and-conquer programs, user can easily tune grain-size by comparing total number of nodes and serial execution time. The labels $p[0] \dots [p7]$ refer to the subcomputations. The **decompose** function divides the input problem into eight subproblems and fills input fields of each subcomputation. Then, the **for** loop fires subcomputations. As product of submatrices arrive, they are added pairwise to construct one

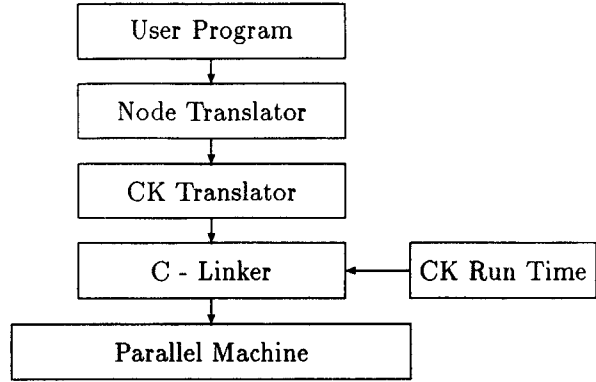


Figure 7: Layers of program development

quadrant of the local result in *when-block*'s. Although in principle the add operations can be performed in parallel, here they are carried out sequentially.

As illustrated by this example, parallel implementation of divide-and-conquer algorithms is significantly simplified by the node construct, and the user is freed from the following laborious tasks:

- synchronization management : keeping track of responses from subcomputations, and execution of *when-block*'s if their conditions are met.
- tree communication : handling parent-child communication.
- allocation : automatic allocation and deallocation of messages.
- Dynamic load balancing.
- Machine dependent expression.

Once the algorithm is implemented with node constructs, it is translated into a Chare Kernel program. Then, Chare Kernel translator produces the C code. Finally, the C code is compiled and linked with Chare Kernel Runtime environment, as illustrated in Figure 7.

5 Implementation

A translator has been developed to transform the user program with node definitions into a Chare Kernel program. A Chare Kernel program consists of chare definitions, function definitions, and message definitions. A chare is a parallel action with several properties. They are not preemptible and execute for

```

readonly float R1;
R1 A[20][20], B[20][20];

node main {
  node mult:root;
  init : {
    InitMatrix(); /* read and initialize readonly matrices */
    InitRootInput(root.in); /* fill row,column and size fields of root.in */
    fire root;
  }
  when root:{PrintResult(root.out);}
}

node mult {
  in : { int rowA, colA, rowB, colB, m;}
  out: { float c[in->m*in->m];}
  node mult : p[8];
  cond c[4];
  init : {
    if ( matrix-size < grainsize) {
      sequentialmult(in,out);
      send result;
    }
    else {
      decompose(in,p);
      for(i=0;i<8;i++) fire p[i];
    }
  }
  when p[0,1] : { add(p[0].out,p[1].out,out,0); /* compute C11 */
                set c[0];}
  when p[2,3] : { add(p[2].out,p[3].out,out,1); /* compute C12 */
                set c[1];}
  when p[4,5] : { add p[4].out,p[5].out,out,2); /* compute C21 */
                set c[2];}
  when p[6,7] : { add p[6].out,p[7].out,out,3); /* compute C22 */
                set c[3];}
  when c[0-3] : { send result;}
}

decompose(in,p)
mult_IN *in, mult_NODE *p;
{ /* fill fields of p[].in */}

add(p1out,p2out,out,quad)
mult_OUT *p1out, *p2out, *out;
int quad;
{ /* add corresponding quadrant of the input matrices
   out->c = p1out->c+p2out->c */ }

sequentialmult(in,out)
mult_IN *in;
mult_OUT *out;
{ /* sequential matrix multiplication */}

/* mult_IN, mult_OUT, and mult_NODE are defined by the translator */

```

Figure 8: Matrix multiplication node definition

a very small time compared to a process in general. A chare consists of a local data declaration block, a number of entries and functions. A chare can send a message to an entry of any other chare. When a message is received, the code at the entry point specified by the message is executed.

Figure 9 depicts the translation of a node definition to a chare definition. Input and output definitions are converted to message definitions. In addition to user supplied data fields, input-messages contain two more fields for parent chare address and entry point number.

For each label that is declared in node-declaration statement, message pointers with the same label name are declared in local data declaration block as follows:

```
/* node declaration */
node node-name : label;

/* message pointers */
struct node-name_NODE {
    node-name *in;
    node-name *out;
} label;
```

For each label again, a response entry is created to receive messages from the subcomputations associated with the label. *init-block* is converted to an entry with a name *init*. *init* entry allocates memory for the outgoing messages, initializes other data structures used by the system before executing user code in *init-body*. Each *when-body* is converted to a private function of the chare.

In order to ensure synchronization as specified in the node definition, a counter for each *when-block* is initialized in the *init* entry to the number node and condition labels listed in the *condition-list*. When a response message is received from a subcomputation, the response entry that gets the message performs following synchronization-code:

```
entry label_response :
    for all when-blocks depends on the label
        decrement counter of the when-block
        if counter is zero invoke when-body
}
```

The response entry knows which *when-block* is dependent on it (by examining *condition-list* again). It decrements the counter of each dependent *when-block*. If the counter reaches to zero, the corresponding *when-body* is called. The *set* instruction also performs same synchronization procedure. The Chare Kernel code for *fire label* is :

```
label.in->parent = MyChareId();
```

```
label.in->epoint = label_response;
CreateChare(node-name,
            node-name@init,label.in);
/* where node-name is the one
   that is referred by label */
```

It initializes the parent address field to the address of the current chare, and the entry point field to the number of entry which is created for the *label*. Then it creates a chare and sends the message *label.in* to the *init* entry of the newly created chare. *send result* statement is translated into :

```
SendMsg(in->epoint,out,&(in->parent));
free-messages-received-from-subcomputations
CkExit();
```

It sends the message *out* to the parent chare using the address information in the *in* message.

6 Performance

Tables 1-4 depict the performance results of several programs on shared memory architectures (Sequent Symmetry, Encore Multimax) and on nonshared memory architectures (intel's IPSC/2, NCUBE/two). Description of the programs is as follows:

Adaptive Quadrature Integration [6] of the function $\frac{\sin x}{x^{3/2}}$ over the interval $[10^{-6}, 1]$ correct up to 10^{-14} . Interval is divided into two if the accuracy is not sufficient. If the difference between computed error and required error is less than 10^{-11} , computation continues sequentially.

Partition Counting the number of partitions of n identical objects into k piles, using the recursive formulation:

$$f(n, k) = f(n - 1, k - 1) + f(n - k, k), \quad \text{where } n=100, k=20 \text{ for this case.}$$

Clique Finding the largest clique for a given undirected graph. All potential cliques are generated with a divide-and-conquer approach [14].

Matrix Multiplication Multiply two 160×160 matrices as in explained in section 3. When size of the submatrices reaches below 20×20 , multiplication is done without further division.

All programs achieved almost linear speedups on shared memory machines. This indicates that enough parallelism is available. However, on nonshared memory machines, the performance is not as good as expected. The second version of the chare kernel system

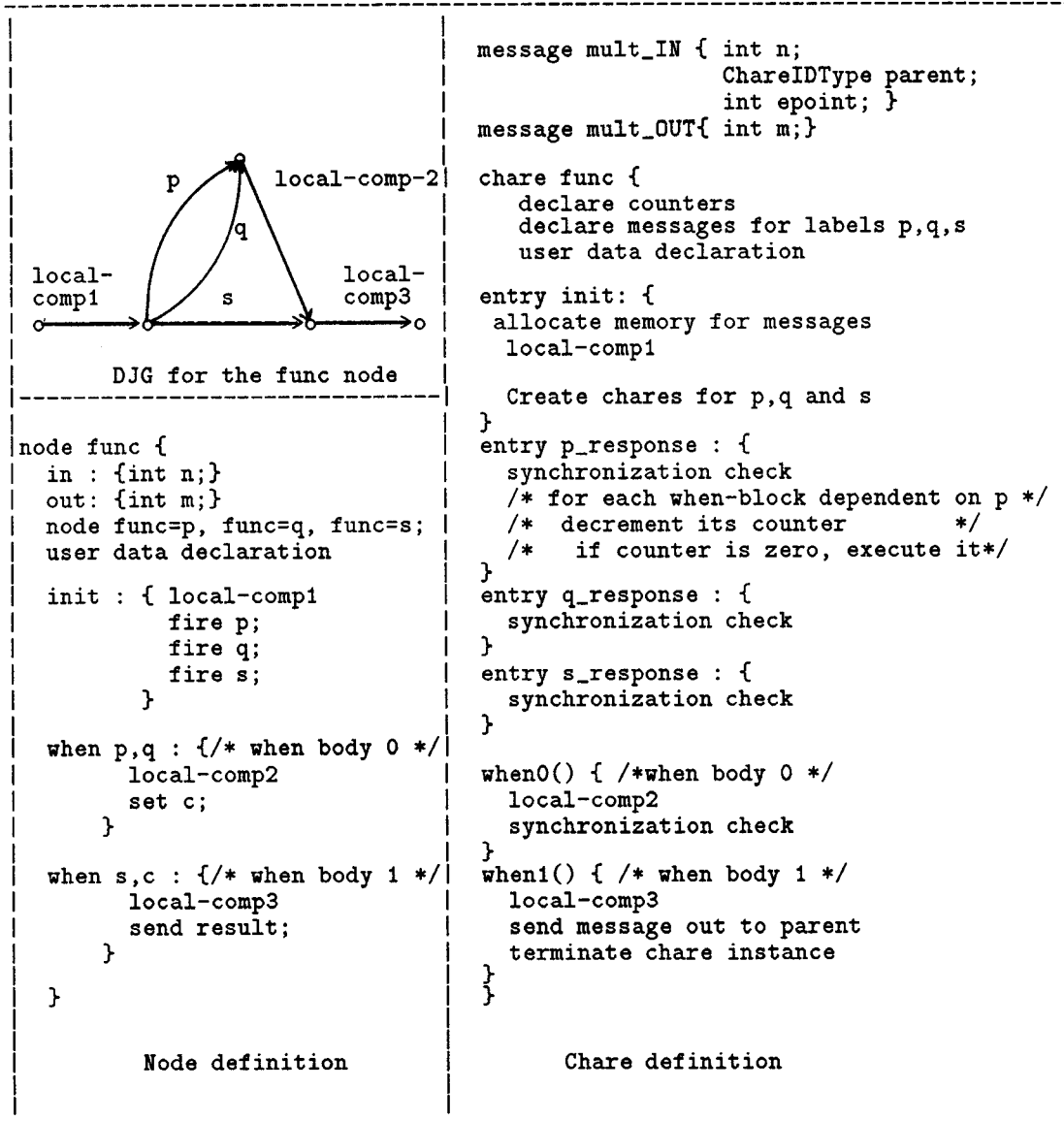


Figure 9: Node to chare translation

machine	serial	1	2	4	8	16	20
sequent	29.3	29.8	14.9(1.97)	7.5(3.91)	3.6(8.1)	1.8(16.2)	1.5(19.53)
max	81.7	83.3	41.7(1.96)	21.2(3.85)	10.2(7.99)	-	-
ipsc2	28.4	29	15.9(1.5)	9.7(2.97)	6.2(4.58)	4.2(6.76)	-
ncube	12.5	13.1	7.1(1.76)	3.9(3.2)	2.4(5.21)	1.8(6.94)	-

Table 1: Adaptive Quadrature - time (speedup)

machine	serial	1	2	4	8	16	20	32
sequent	178.9	179.7	89.9(1.99)	45(3.96)	22.9(7.81)	11.2(15.97)	9.1(19.66)	-
max	243.6	245.4	123.5(1.97)	62.8(3.88)	31.3(7.78)	-	-	-
ipsc2	168.1	169.3	87(1.93)	47.3(3.55)	27.4(6.14)	16.6(10.13)	-	-
ncube	116.4	117.8	x	31.3(3.72)	17.3(6.73)	10.4(11.19)	-	6.5(17.9)

Table 2: Partition - time (speedup)

machine	serial	1	2	4	8	16	20	32
sequent	87.8	88.5	39.3(2.23)	21(4.18)	11.1(7.85)	5.4(16.35)	4.4(19.95)	-
max	173	176.5	89.4(1.94)	45(3.84)	29.2(5.92)	-	-	-
ipsc2	102.1	103.1	52.5(1.94)	27.3(3.74)	14.8(6.9)	8.1(12.6)	-	-
ncube	110.5	113	55.6(1.99)	x	15.5(7.13)	x	-	6.6(16.74)

Table 3: Clique - time (speedup)

machine	serial	1	2	4	8	16	20
sequent	87.4	83.2	41.4(2.11)	22(3.97)	11.1(7.87)	5.5(15.89)	4.7(18.59)
max	99.1	98.8	49.2(2.01)	25.3(3.92)	12.6(7.87)	-	-
ipsc2	62.4	66.5	36.23(1.72)	19.9(3.13)	10.6(5.86)	6.7(9.31)	-

Table 4: Matrix Multiplication - time (speedup)

speedup = serial-time/parallel-time

execution time is in seconds.

entries marked with x are not available due to memory management faults.

was recently completed. The load balancing scheme [18] has not yet been fine tuned. Once that is done, we expect the programs to yield as high performance as reported for previous chare kernel programs. In some cases, superlinear speedups are achieved on the Sequent Symmetry. Since there is no speculative work in these examples, it is highly probable that this is due to higher data locality achieved by the parallel version.

7 Summary

We presented a high level language construct to support a convenient representation of parallel divide-and-conquer algorithms on MIMD multiprocessors. The end user is freed from tedious tasks such as communication setup, and synchronization. We also demonstrated that such systems can be built without much effort on top of the Chare Kernel system. Preliminary performance results encourage us to develop high level support systems for other widely used parallel computational models. Currently, a divide-and-conquer program written using the system described here is a stand alone program. In the future, we plan to allow linking different modules, each possibly written in a different paradigm (with its own translator), or using the Chare Kernel language itself, into a single program.

References

- [1] A.Aho, J.Hopcraft, and J.Ullman. *The design and analysis of computer algorithms*, Addison-Wesley, (1974) pp230-232.
- [2] J.L.Bentley, "Multidimensional divide-and-conquer," *Comm. ACM*, vol. 23, no. 4., April 1980.
- [3] F.W. Burton, M.M.Huntbach, "Virtual Tree Machines," *IEEE Trans. Comput.*, vol. C-33, no. 3, pp. 278-280, Mar. 1984.
- [4] F.W. Burton, "Storage management in virtual tree machines", *IEEE Trans. Comput.*, vol. 37, no. 3, pp. 321-328.
- [5] N.Carriero, D.Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444-458, Apr. 1989.
- [6] Conte and de Boor, *Elementary Numerical Analysis*, (1980) pp 328-332.
- [7] J.B.Dennis, E.C.Van Horn, "Programming semantics for multiprogrammed computations," *Comm. ACM* vol. 9, no. 3, pp.143-155, 1966.
- [8] R.Finkel, U.Manber, "DIB-A distributed implementation of Backtracking," *ACM TOPLAS* 9(2) pp.235-256, April 1987.
- [9] E.Gabber, "VMPP: A practical tool for the development of portable and efficient programs for multiprocessors", *IEEE Trans. Parallel and Distributed Sys.*, vol. 1, no.3, pp304-316, July 1990.
- [10] G.Golub, C. van Loan, *Matrix computations*, Johns Hopkins University Press, Baltimore, 1983.
- [11] L.V.Kale, "The Chare Kernel parallel programming language and system", *Proceedings of the International Conference on Parallel Processing*, Vol II, Aug 1990, pp17-25.
- [12] B.W.Kernighan, D.M.Ritchi, *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1978
- [13] V.Kumar, V.N.Rao, "Parallel depth first search,Part 2:Analysis," *Int'l. J. of Parallel Programming*, Dec 1987, pp. 501-519.
- [14] C.Mead, L.Conway, *Introduction to VLSI systems*, Addison-Wesley, (1980) pp 307-312.
- [15] P.A. Nelson, L.Snyder, "Programming paradigms for nonshared memory parallel computer," in *The Characteristics of Parallel Algorithms*, L.H.Jamieson, D.B.Gannon, and R.J.Douglas, Eds. Cambridge, MA: MIT Press, 1987, pp. 3-20.
- [16] J.M.Ortega, *Introduction to parallel and vector solution of linear systems*, Plenum Press, New York, 1988.
- [17] F.J.Peters, "Tree machines and divide-and-conquer algorithms," *Proc. Conf. Analyzing Problem-Classes Programming Parallel Computing*, Nuremburg, W.Germany, June 1981, pp. 25-36.
- [18] W.Shu, L.V.Kale, "Dynamic scheduling of medium-grained processes on multicomputers", *Tech. Rep. UIUCDCS-R-89-1528, Dept. of Computer Science, University of Illinois at Urbana-Champaign, July 1989.*
- [19] Q.F.Stout, "Supporting divide-and-conquer algorithms for image processing," *J.Parallel Distrib. Comput.* 4 (1987), pp.95-115.