Issues in Implementing PARKA using the techniques of Chaos

Evan Golub
Department of Computer Science
University of Maryland at College Park

The motivation behind the creation and continuing development of PARKA (PARallel Knowledge and Association) has been to demonstrate how parallel machines can be utilized by large AI systems to achieve the rapid response time required by realistic applications. PARKA is a symbolic, semantic network knowledge representation system which has been used to demonstrate the effects of parallelism on common types of inferencing.

Since property inheritance is a major part of most representation systems, much of the work done with PARKA has been related to ISA-hierarchy property inheritance. PARKA's basic inference mechanism is called activation wave propagation. Each propagation step consists of every node in the current wave front simultaneously transmitting the wave to their neighbors.

PARKA was originally implemented on a CM-2, using an isomorphic mapping between the processors and the nodes of the semantic network. It used pvars to represent the links and the node names. At each wave, every processor which was being used to represent a node would receive the names of nodes to be involved, and would determine if it was one of them. If it wasn't, it would turn itself off for that stage. This wave propagation would continue across the network until the wave front became empty. Full details of this implementation of PARKA can be found in [1].

The CM-2's SIMD architecture and large number of small memory processors were exploited well by the version of PARKA which was implemented on it. However, the implementation was strongly connected to the CM-2's architectural design and the use of *LISP. Since then, PARKA has been implemented, and continues to develop, on a CM-5 at the University of Maryland. While the concepts being developed are portable, the software implementation is not. This served as the motivation for looking into how PARKA could be implemented in a way such that it would be highly portable.

Work is also currently being done at Maryland with a portable parallel library called Chaos. This library is intended to help parallel system programmers efficiently code irregular problems on distributed memory machines. The philosophy behind Chaos is to extract interprocessor communication patterns at run time, and use this information to allow a distributed shared memory to effectively exist with low communication overhead.

An example of how this is achieved is the taking of a single logical array of data and distributing it amongst the processors memories, and accessing the data via an indirection array. Each element of a data array has a logical position in the global representation of the array, as well as a physical position in the array segment stored on the processor that owns it. For example, if we had an array of characters

index	1	2	3	4	5	6	7	8	9
arr	A	B	C	D	E	F	G	H	I

Figure 1

which we wanted to store for use on a parallel system with three processors, we might store it as

local_index	1	2	3	1	2	3	1	2	3
local_arr	A	B	C	D	E	F	G	H	I
processor		P0			P1			P2	

Figure 2

It is important to note that although the character E is now in the second position of the array it lives in on processor P1, it is still the fifth element of the complete logical array arr. It is also important to realize that it would be equally valid to store the data as

local_index	1	2	3	1	2	3	1	2	3
local_arr	A	D	H	B	C	F	E	G	I
processor		P0			P1			P2	

Figure 3

The character E is still the fifth element of the logical array, but now resides in the first position of the array portion on P2.

During data initialization, the logical array is allocated across the processors. Each processors' local array leaves room at the end of itself to act as a buffer to hold off-processor data which is brought in by data requests. The reason the buffer space resides in the tail end of the array will be made clear below.

Prior to a parallel computation block, the indirection arrays which will be used in that block are inspected. These indices are in terms of the elements' positions in the complete logical array. The inspection process accomplishes several things. It determines which elements that will be accessed reside in off-processor memory, and generates a schedule to be used to transfer the appropriate data to the processors which will be using them. It also generates a new indirection array in which references to off-processor data elements are updated to refer to the buffer position in which it will be placed.

For example, using Figure 1, if processor P0 had an indirection array, referring to elements of the logical array “arr”, which was

ind_array $\boxed{2}\boxed{5}$

the data it will be requesting from “arr” resides in the local array in position number 2 (logical arr[2]) and in the array on processor P1 in position number 2 (logical arr[5]). The communication schedule will do nothing for the data which is already local, but will arrange to transfer P1.local_arr[2] into P0’s local array, in the first position of the buffer space. So, if P0's array was originally

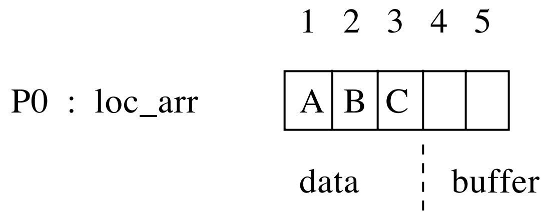


Figure 4

then after the data is brought in, the array will be

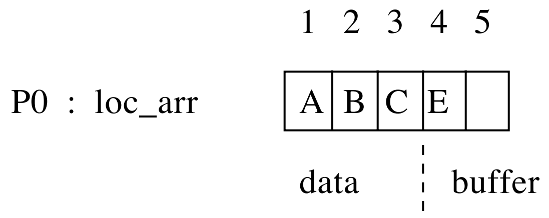


Figure 5

and the new indirection array which P0 will be using would be

new_ind_array	2	4

where the index values now strictly refer to positions of the local array, but still refer to the same positions in the global, logical array, as the original indirection array did. When used with loc_arr on P0, this will allow the program to access the second and fifth elements of the logical array via a local memory access.

This manner of storing and accessing data allows a program to utilize the combined memory space of all processors assigned to the job while limiting the amount of communication done. If it is possible for each processor to be assigned data in such a way that many or most of its data requests will be on-processor ones, communication costs can be reduced even further. Since the data can be distributed in an irregular way, as shown in Figure 3, it is possible to do this for many programs.

The Chaos library provides routines for both generating schedules and translation tables as well as ones for executing the data transfers in an organized fashion. These are described in detail in the Chaos manual [2]. The library currently runs on a variety of machines such as the Intel Paragon, Thinking Machines CM-5 and

IBM SP-2. Additional ports are currently underway, including one to the Cray T3D.

The philosophy behind Chaos, as well as the library itself, is to be portable between parallel supercomputers. The fact that a version of PARKA implemented using these techniques would be portable is a nice benefit, but not the original inspiration for this implementation. The Chaos library is currently being used in the solving of irregular problems such as structured mesh sweeps and upper triangular solves[3]. The computation patterns associated with the basic inference mechanism of PARKA, activation wave propagation, appears to be one which will access data in a structured yet irregular order which can not be known until run time.

Implementing PARKA using the ideas of Chaos is not simply a matter of converting *LISP code into C. While the basic algorithm concepts remain the same, the data representation system has to be designed in such a way that the data in our semantic network are stored in arrays, and accesses to it are done via indirection arrays.

The second of these two requirements is the simpler, since array positions have unique values as did the node addresses in the

CM-2 implementation. The more challenging goal is the choice of a structure or set of structures to use to hold the network. This structure needs to have the ability to grow in three ways. The first is that as new nodes are added, the network must integrate them into itself. The second is that if a new property is added, the appropriate links must be added for that new property for nodes which will now use it. The last is that the number of ISA links for a single node is not known at design time. These ISA links have to be represented in a structure that can expand non-uniformly, since not all nodes will have the same number of ISA links. Additionally, since it is these ISA links which will be followed during the wave front activation, it is important to be able to utilize the structure in a highly parallel fashion.

The following traces through the design process as it occurred. While the first idea presented is clearly not a good choice, it served as a starting point upon which the final structure was based. The first structure which was considered was a two-dimensional array in which each column would represent a frame or property, and each row would represent a property link or ISA link. Columns would be added for new frames and rows would be added when new properties or ISA links were defined. While this structure would work conceptually, the costs of maintaining a two-dimensional array which would need to be expanded in both directions were not acceptable.

The next structure which was considered was also a two-dimensional array, but one which would only expand in one dimension. The idea was to have each column be associated with some frame. Each frame would have its "origin" column. There would be some fixed number of rows, R. In the "origin" column, row R-1 would be a pointer to either another column of to NULL. If it was to another column, this would signify that there were more properties than could be fit in a column, so this next column was a continuation of the property links. Since all frames have the same number of potential property links, a table could be kept telling us how many auxiliary property columns would need to be traversed to get to a specific property link's position.

Similarly, row R would be a pointer to the next column of ISA links. The link type ISA needs to be treated separately since the number of ISA links which a given frame has is unrelated to the number which the other frames have. Using this column linking, each frame could have its own custom allocation of ISA links.

With this structure, the network could be represented with a good utilization of space, and with a fairly easy method of traversing the links. However, even though the two-dimensional array would only grow in one dimension, it would still be costly to do these expansions. While looking at ways to make this structure work, a different method was realized, and eventually used to store the network. This structure is a set of two one-dimensional arrays.

The first is an array, “frame_tbl,” of strings which holds the frame and property names. A property name is identified through a character “P” prefixed to the name, and a frame through the character “F.” The second array, “network,” is a variable sized array which is conceptually composed of multiple fixed length arrays. Each element of this array is a pointer to a frame, and these pointers represent the links which form the network. For array position i, where i is even, positions i and i+1 form the link between two frames of the network. The arrays “frame_tbl” and “network” are used as parallel arrays, with each frame or property name being aligned with one of the fixed length arrays.

Each of the arrays used in the representation of the network is conceptually a single array. The “frame_tbl” array is replicated across all processors being used in the system. However, when implemented on an N-processor system, the “network” array is actually N sets of local arrays. There are two phases associated with this distribution, the construction of the distributed network, and the use of the network in a search request.

Initially, the network was built and used on a single processor. Once that version was working, the task was to move to a multiple processor setup. This was first done by having one of the processors act as the startup host. The entire network was loaded and built on the single processor. Once that was done, the network was "re"distributed amongst all of the processors. This was accomplished by creating an initial mapping which showed all frames held by P0, and a new mapping which had the frames distributed, and then calling the appropriate Chaos routines to effect the redistribution :

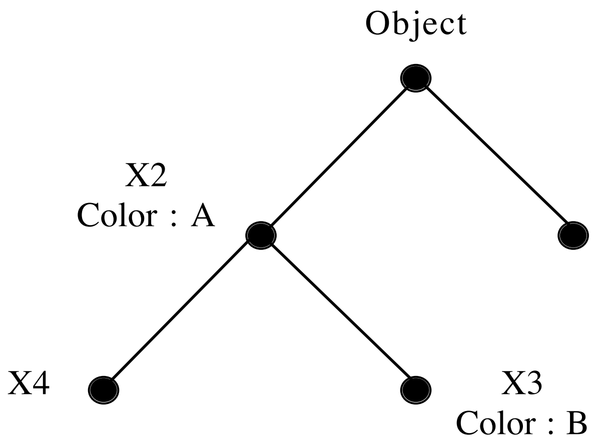
```
tt=init_ttable_with_proc(1, NewDistribution, myNetsize);
remap(tt, InitialDistribution, &sch, NewDistribution,
   &myNewNetsize);
tt=build_translation_table(1, NewDistribution, myNewNetsize);
PARTI_gather(sch, newNetwork, network, FRAME_SIZE);
```

While this method worked well, it did limit the size of the network which could be easily loaded to the size of a network which could be loaded into a single processor's memory. In later versions, the network is built in a distributed fashion.

Once the network exists in the distributed arrays, it must be possible to search the network. The first specific search which was implemented was based on Touretsky's Inferential Distance Ordering (IDO) inheritance system[4]. This system is designed to deal with ambiguities over which property value to choose, caused by a frame being connected to several other frames via ISA links, which are explicitly valued for that property. It tells the system that in the event that a frame can inherit a property from one of several other frames, to choose the frame which is semantically closest. A formal definition of Touretsky's IDO inheritance scheme is given by Matt Evett in [5] as :

Assume the frame in question, X, is not explicitly valued for the given property, P(if it is, the explicit value is X's value for P). Let **B** be the set of ancestors {B_1, B_2, ...} of X that are explicitly valued for P. X takes B_i's value for P as its own, provided B_i is an element of **B** such that there is no B_j, $j \neq i$, that is an IS-A descendant of B_i. If more than one element of **B** meets this criterion, X is said to be ambiguously valued for property P.

As an example of a semantic network of ISA links where there is potential for ambiguity, we can look at the following :



In this case, the frame Object can either get it's value for "Color" from frame X2 or X3. According to the IDO rules, since X2 is semantically closer to the object, it will inherit the "Color" value of "A."

In order to accomplish this, PARKA must be able to know the semantic distance between two nodes in a knowledge base. The way in which this has been done in the CM implementation is through a topological ordering of the frames. This technique leads to different results than those which would be obtained if using Touretsky's exact algorithm in some rare cases, but [5] states that these exceptions do not seem to appear in typical knowledge bases, and as a result, this difference should have little impact on most inferences.

In the new version, the distances are calculated based upon the wave front in which an ISA-attached frame which is explicitly valued for the property in question is an element of, rather than the topological ordering. This distance metric should not lead to any additional differences from the “pure” IDO system.

The sweep across the network was implemented by using an owner-computes philosophy at each wave front. For a given wave front W_i, each processor knows which of the frames that it has locally are involved. It then computes the list of frames which will be involved in the next wave front W_{i+1} based on the frames which it examined. The processors then communicate to each other the partial information they have about wave front W_{i+1} and as a result, each processor will know which of its frames will be involved in that front. This essentially gives us the same information as a separate topological ordering sweep across the network would have given us, but it is done in-progress rather than in advance.

In order for this technique to work well, for each wave front the frames involved must be well distributed across the processes. The initial networks which were used for testing were of a constant branching factor and created a tree-like network. For this reason, it was relatively straight forward to choose a cyclic distribution which would spread out each level of the network, and therefore each wave front, across all processors. When dealing with an actual knowledge base, however, it may not be as simple to determine a distribution, and it is most likely that an irregular distribution pattern would be required.

The task of choosing a distribution which will serve our purpose has two levels to it. The first of these is the need to develop an appropriate heuristic which will distribute the network in an advantageous way. One idea as to how to proceed on this is to use the topological ordering of the network to identify the wave fronts, and then distribute each front in a way to minimize cross edges between waves Wi and Wi+1. However, since the network needs to be initially created in a distributed manner so as to take advantage of all of the involved processors' memory, there will need to be a way for all processors to collaborate on deciding how to redistribute the data.

This could be a very costly procedure depending on the amount of interprocessor communication which would be required. This analysis would need to be done in two situations. The first would be upon initial creation of the network. Once it was read in, the initial distribution would need to be chosen. However, we would also need to deal with the situation where new nodes and/or links were added to the network after it was initially built. While it should be possible to add to the network without having to re-analyze the distribution each time, we would most likely want to either used a modified version of the algorithm to place new elements, or establish a condition which would cause us to re-analyze and re-distribute the network. Examples of algorithms for data partitioning exist for other problems to which the Chaos library has been applied[6].

In the first set of experiments done with our partial implementation of PARKA, knowledge bases were generated with the following characteristics : A network described as B x D represents a knowledge base composed strictly of frames and ISA links. Each non-leaf frame has B ISA links coming out of it, each connecting to a distinct frame one level down. There are a total of D levels in the network associated with the generated knowledge base.

Once a network is generated, it is distributed across the processors using a CYCLIC distribution. For the first few levels, this causes an uneven distribution of work. The number of levels for which the work is unevenly distributed depends on both the branching factor B, the period of the cycle used and the number of processors being used. However, for networks of reasonable size, the amount of work which is distributed fairly is exponentially larger than that which is not.

The first tests have shown promising empirical results. This version of PARKA was run on networks of dimensions (and sizes) of 9x3(29524), 10x3(88573), 11x3(265720) and 12x3(797161). It yielded the following results when run on the local SP-2 on 1, 2, 4, 8 and 16 processors :

size of network \ # nodes	1	2	4	8	16
9x3	0.115	0.060	0.033	0.021	0.017
10x3	0.345	0.175	0.091	0.050	0.031
11x3	1.037	0.523	0.256	0.137	0.088
12x3	-.---	1.557	0.783	0.403	0.216

Reference Notes

1 “Parallel Knowledge Representation on the Connection Machine”
- Matt Evett, James Hendler, Lee Spector

2 The Chaos Manual

3 “Runtime Support and Dynamic Load Balancing Strategies for Structured Adaptive Applications”
- Bongki Moon, Gopal Patnaik, Robert Bennett, David Fyfe, Alan Sussman, Craig Douglas, Joel Saltz, K. Kailasanath.

4 The Mathematics of Inheritance Systems
- D.S. Touretsky, 1986.

5 PARKA : A System for Massively Parallel Knowledge Representation
- Matt Evett

6 “Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library”
- Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, Milan Hodoscek.