Support for Distributed Dynamic Data Structures in C++ [*]

Chialin Chang Alan Sussman
Joel Saltz
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park, MD 20742
{chialin, als, saltz}@cs.umd.edu

Abstract

Traditionally, applications executed on distributed memory architectures in single-program multiple-data (SPMD) mode use distributed (multi-dimensional) data arrays. Good performance has been achieved by applying runtime techniques to such applications executing in a loosely synchronous manner. However, many applications utilize language constructs such as pointers to synthesize dynamic complex data structures, such as linked lists, trees and graphs, with elements consisting of complex composite data types. Existing runtime systems that rely on global indices cannot be used for these applications, as no global names or indices are imposed upon the elements of these data structures.

A portable object-oriented runtime library is presented to support applications that use dynamic distributed data structures, including both arrays and pointer-based data structures. In particular, CHAOS++ deals with complex data types and pointer-based data structures by providing *mobile objects* and *globally addressable objects*. Preprocessing techniques are used to analyze communication patterns, and data exchange primitives are provided to carry out efficient data transfer. Performance results for applications taken from three distinct classes are also included to demonstrate the wide applicability of the runtime library.

1 Introduction

A large class of applications execute on distributed memory parallel computers in single-program multiple-data (SPMD) mode in a loosely synchronous manner [6]. That is, collections of data objects are partitioned among processors, and the program executes a sequence of concurrent computational phases. Each computation phase corresponds to, for instance, a time step in a physical simulation or an iteration in the solution of a system of equations by relaxation, and synchronization is only required at the end of each computation phase. Therefore, once the data for a computation phase (which is typically produced by a preceding computation phase) becomes available, a collective communication phase can be performed by all processors, after which each processor will contain a local copy of the data needed to carry out the computation phase. The computation phase can then be executed completely locally on each processor.

Traditionally, such applications utilize (multi-dimensional) data arrays, which are often partitioned dynamically during program execution. Optimizations that can be carried out by compilers

[*]This research was supported by the National Science Foundation under Grant #ASC 9318183, and NASA under Grant #NAG 11485 (ARPA Project #8874).

are thus limited, and runtime analysis is required [20]. Good performance has been achieved by applying such runtime techniques to various problems with unstructured data access patterns, such as molecular dynamics for computational chemistry [8], particle-in-cell (PIC) codes for computational aerodynamics [14], and computational fluid dynamics [4].

However, many applications, such as image processing, geographical information systems, and data mining, utilize constructs such as pointers to synthesize complex composite data types, and build dynamic complex data structures such as linked lists, trees, and graphs. We refer to these as *pointer-based data structures*. Such data structures are dynamic in the sense that data elements in the data structures are often created and/or deleted during program execution, and accesses to these data elements are done through pointer dereferences. As a consequence, access patterns to such data structures cannot be determined until runtime, so only runtime optimization techniques can be performed. In addition, unlike array elements, elements in a pointer-based data structure do not have global names or indices, precluding the use of many existing runtime systems that rely on the existence of global indices.

CHAOS++ is a runtime library targeted at object-oriented applications with dynamic communication patterns. It subsumes CHAOS [5], which is a runtime library developed to efficiently handle adaptive and irregular problems that use arrays as their main data structures. CHAOS provides interfaces for use by both C and Fortran programs. In addition to making use of features of the underlying CHAOS library, CHAOS++ also provides support for pointer-based data structures, and allows flexible and efficient data exchange of complex data objects among processors.

CHAOS++ is implemented as a C++ class library, and can be used directly by application programmers to port adaptive and/or irregular codes. The design of the library is architecture independent and assumes no special support from C++ compilers. Currently, CHAOS++ uses message passing as its transport layer, and is implemented on distributed memory machines such as the Intel iPSC/860 and Paragon, the Thinking Machines CM-5, and the IBM SP-1 and SP-2. However, the techniques used in the library can also be applied to other environments that provide a standard C++ compiler and a mechanism for global data accesses, including various distributed shared memory architectures [12, 15, 22].

2 Runtime Support for Distributed Dynamic Data Structures

In this section, we give an overview of the CHAOS runtime library, and discuss the additional issues that must be addressed to efficiently support distributed pointer-based data structures.

2.1 Overview of the CHAOS Runtime Library

The CHAOS runtime library [5] has been developed to efficiently handle adaptive and irregular problems that use arrays as their main data structures. In these problems, arrays are frequently partitioned in an irregular manner for performance reasons, for example to reduce communication costs or to obtain better load balance. To enable remote accesses on distributed memory architectures, the CHAOS library constructs a translation table that contains the host processor number and the local address for every array element. Since the translation table can be large (the same size as the data array), it can be either replicated or distributed across the processors.

For loosely synchronous applications, the data access pattern of a computation phase is usually known before entering the computation phase and is repeated many times. CHAOS thus carries out optimization through two phases, *the inspector phase* and *the executor phase* [20]. During program execution, the CHAOS inspector routines examine the data references, given in global indices, and convert them into local indices by using the translation table. Duplicate references are then removed through simple software caching, and unique references are coalesced to reduce communication latency and startup costs. The result of these optimizations is a communication schedule, which is later used by CHAOS data transportation routines in the executor phase to efficiently collect the data needed for the computation phase. CHAOS also provides primitives to redistribute data arrays efficiently at runtime. Special attention has been devoted towards optimizing the inspector for adaptive applications, where communication patterns are not reused many times [21].

2.2 Issues in Runtime Support for Pointer-Based Data Structures

The CHAOS library has been successfully applied to irregular and adaptive problems that use distributed arrays of primitive data types (integers, double precisions, etc.). Data access patterns in these applications are typically represented as global indices, and a simple assignment or a function call to an efficient block copy routine is used to pack and unpack array elements in buffers for message passing. However, in an object-oriented model, where programmers are allowed to define complex composite data types and build data structures that contain pointers to other objects, two main problems must be addressed for any runtime system to support applications that make use of such complex data structures. The CHAOS++ library provides solutions to both problems, as will be described in Section 3.

First, the runtime system must provide support for arbitrarily complex data types that may contain pointers to other nested objects. This typically happens when a hierarchy of data types are employed, and *sub-objects* within an object are instantiated dynamically at runtime. As a consequence, remote accesses to these objects, which are carried out by message passing on a distributed memory architecture, require more sophisticated functions than a simple memory block copy routine to pack and unpack the complex objects from message buffer. The runtime system must follow pointers when packing an object into a message for a send operation, and restore the pointers when unpacking an object from a message for a corresponding receive operation.

The second problem that must be addressed is support for naming and finding off-processor objects. Since elements (objects) may be added to and removed from pointer-based data structures dynamically, no static global names are associated with the elements and accesses to those elements are done through pointer dereferencing. Thus, partitioning a pointer-based data structure may assign two elements connected via pointers to two different processors, and raises the need for *global pointers*. A global pointer, as supported by several language extensions including Split-C [11], CC++ [3], and pC++ [13], may point to an object owned by another processor, and effectively consists of a processor identifier and a local pointer (that is only valid on the named processor).

3 The CHAOS++ Runtime Library

CHAOS++ is designed to effectively support applications that contain complex data types and pointer-based data structures. As for applications that utilize the CHAOS library, a parallel application executing in loosely synchronous mode using CHAOS++ performs a preprocessing phase before each computation phase to determine the communication required to execute the computation. A communication schedule is generated, and is used by CHAOS++ data exchange routines to perform the required communication.

As was described in Section 2, CHAOS++ must be able to deal with complex data types and global pointers. These two problems are handled by *mobile objects* and *globally addressable objects*. We will now discuss these techniques in more detail. The approach that CHAOS++ takes relies heavily on class inheritance, as provided by the C++ language.

3.1 Mobile Objects

CHAOS++ defines an abstract data type, called **Mobject**, for *mobile objects*. These are objects that may be sent from one processor to another, so must know how to pack and unpack themselves to or from a message buffer. The **Mobject** class contains two virtual member functions, **pack** and **unpack**, and is designed as a base class for all objects that may migrate between processors, and/or will be accessed by processors other than the ones they are currently assigned to. In C++, virtual functions allow for dynamic binding for a function call, so that the **pack** or **unpack** function that the CHAOS++ runtime system calls to pass an object from one processor to another will be based on the actual type of the object at runtime (so CHAOS++ can always use the type of the base class, **Mobject**, in its routines).

For an object that only occupies consecutive memory locations, the **pack** and **unpack** functions consist of a simple memory copy between the object data and the message buffer. CHAOS++ provides default implementations of the virtual functions that can be used for this case, although the size of the object must be given.

For a more complex object that contains pointers to sub-objects, and thus has parts to be copied scattered throughout the program memory (runtime heap), the application programmer must provide an implementation of **pack** and **unpack** that supports a *deep copy*. To be more specific, the **pack** function should copy both the contents of the object and its sub-objects into the message buffer. A straightforward implementation of **pack** for an object with sub-objects can be done by also deriving the classes for all sub-objects from **Mobject**, and having the **pack** function for an object recursively call the **pack** function for each of its sub-objects. On the receiving processor side, the **unpack** function must perform the inverse operation. That is, it must interpret the flattened object data in the buffer, packed using the sender's **pack** functions, and restore the complete structure of the object. This includes recursively unpacking all the sub-objects, and setting up all the pointer members properly. Restoring the pointer members properly is not trivial because a local pointer on one processor is not valid across processor boundaries on distributed memory architectures.

In many applications, the contents (sub-objects) for multiple **Mobject**s will be disjoint. On the other hand, special caution must be used in the implementation of the **pack** and **unpack** functions when more than one **Mobject** may contain a pointer to the same sub-object. Some scheme for ensuring that only one copy of an object (sub-object) exists at any point during the program execution must be used in these cases (e.g. reference counting). On the other hand, such problems can also be alleviated by the use of globally addressable objects, as is described in the next section.

3.2 Globally Addressable Objects

Elements in a pointer-based data structure are linked and accessed through pointers. As described in Section 2.2, global pointers are needed in an SPMD execution model to successfully partition pointer-based data structures. One obvious approach is to define a C++ class for global pointers, and overload the dereference operator ("*") so that when a global pointer is dereferenced, the necessary interprocessor communication is generated transparently to the application program. This approach, however, complicates buffer management in the loosely synchronous model, and imposes overhead for converting global pointers between references to remote objects and references to local buffer.

Instead of defining a class for global pointers, CHAOS++ supports global pointers by defining an abstract C++ base class for *globally addressable objects*, or **Gobject**s. A **Gobject** is an object that is assigned to one processor, but allows copies to reside on other processors. The copies are referred to as *ghost objects*. Each **Gobject** has a function that determines whether it is a real object (the permanent version of the object) or a copy of a remote object (a ghost object). Each processor other than the one assigned ownership of the **Gobject** may have a local copy of the **Gobject** as a ghost object. The ghost object is used to cache the content of the remote real counterpart, so that once filled with data from its real counterpart, accesses to the object can be carried out locally. The contents of ghost objects are updated by explicit calls to CHAOS++ data exchange routines, and the decision of when to update a ghost object from a real object is made by the application. Note that this implies that all **Gobject**s are also **Mobject**s.

In this model, a pointer-based data structure is viewed as a collection of **Gobject**s interconnected by pointers. Partitioning a pointer-based data structure thus breaks down the whole data structure into a set of connected components, each of which is surrounded by one or more layers of ghost objects. Pointers between two **Gobject**s residing on the same processor are represented as local pointers, while those that conceptually go across processor boundaries (global pointers) will point to the local ghost object copies of the remote objects. The outermost layer of ghost objects can be thought of as the boundary of the distributed data structure assigned to the local processor, and would be the nodes where a local traversal of the data structure terminates. Figure 1 illustrates an example of partitioning a graph between two processors. The dashed circles represent ghost **Gobject**s . Note how the edges that get partitioned in the original graph are pointing to ghost objects in the partitioned graph.

Since accesses to elements of a pointer-based data structure are done through pointers, the layers

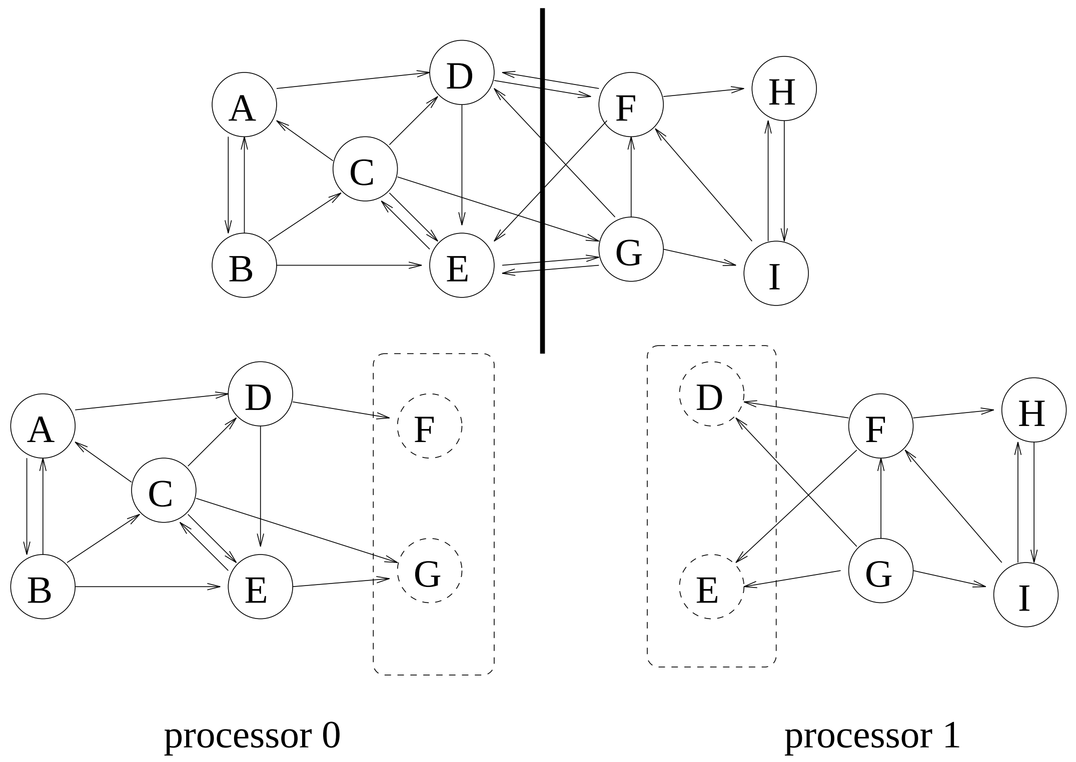

Figure 1: Partitioning a graph between two processors. At top is the original graph, partitioned at the vertical bar, with the resulting processor assignment shown at bottom.

of ghost objects surrounding each connected component encapsulate all the possible remote accesses emerging from that connected component. Accesses to remote objects that are more than one "link" away can be satisfied by creating ghost objects for remote objects that are pointed to by local ghost objects, and filled on demand. A mapping structure is employed by CHAOS++ for each pointer-based data structure on each processor, to manage the ghost objects residing on that processor. The mapping structure serves as the equivalent of the CHAOS translation table (for irregular data mappings), and is used during the inspector phase described in Section 2.1 for translating global references into processor and local address pairs to generate communication schedules. In Figure 1, the dashed box for each processor represents the CHAOS++ mapping structure. The CHAOS++ data exchange routines then use the schedules to transfer data between real **Gobject**s and ghost **Gobject**s.

3.3 Data Exchange Routines

The data transfer between real **Gobject**s and ghost **Gobject**s is carried out by the CHAOS++ data exchange routines, which use the **pack** and **unpack** functions of **Mobject**s to enable deep copies (all **Gobject**s are also **Mobject**s). The communication schedules generated during the inspector phase ensure that neither polling nor interrupts are needed at the receiving processors, so that communication can be performed efficiently.

Although the data exchanges required to manage **Mobject**s and **Gobject**s can be made quite efficient, additional optimizations are still necessary. For complex data types, objects may become quite large, and simply transporting the entire contents of the object for every communication operation may impose unnecessary overhead. CHAOS++ therefore allows an application to specify what data is exchanged during every communication phase. This is done by allowing applications to specify a pair of customized **pack** and **unpack** functions every time a CHAOS++ data exchange routine is invoked. The **pack** and **unpack** functions of a **Mobject** are then only used by default, when no customized routines are provided. This flexibility can increase the possibility of reusing a communication schedule, since different portions of an object may be needed during different computation phases, but the same schedule can be reused as long as different **pack** and **unpack** functions are provided for the data exchanges for each phase.

4 Applications and Performance Results

To provide examples of using CHAOS++, and to evaluate the CHAOS++ library, we will present performance results for three applications. The applications are taken from three distinct classes: computational aerodynamics (scientific computation), geographic information systems (spatial database processing), and image processing.

The first application is a Direct Simulation Monte Carlo (DSMC) method. DSMC is a technique for computer modeling of a real gas by a large number of simulated particles. It includes movement and collision handling of simulated particles on a spatial flow field domain overlaid by a Cartesian mesh [18]. Depending upon its current spatial location, each particle is associated with a mesh cell, and moves from cell to cell as it participates in collisions and various boundary interactions in the simulated physical space. Mesh cells are distributed among the processors to achieve a good load balance, and since particles move between mesh cells, the cells are redistributed across the processors occasionally (once every few time steps) to maintain a good load balance. Moon [14] describes a parallel implementation of the DSMC application that uses the CHAOS runtime library. In the CHAOS implementation, various physical quantities associated with each particle are stored in separate arrays, and the association between the Cartesian mesh cells and the particles is represented by indirection arrays. We have reimplemented the DSMC application in C++, using the CHAOS++ library to maintain the distributed data structures required for parallel execution. In the C++ version, a *Cell* class is defined for the mesh cells, and the particles, represented as objects from class *Particle*, are stored as an array pointed to by a member in the Cell class. Both the Cell and Particle classes are derived from the CHAOS++ **Mobject** base class, and a deep copy of the Cell class is used in redistributing the cells to move all the particles associated with a cell between processors, along with other data for the cell.

Table 1 gives the performance for both the C++/CHAOS++ code and the Fortran/CHAOS code. The simulated space consists of 9,720 cells, and initially contains about 50,000 particles. 400 time steps are performed, and a chain partitioner [14] is used to partition the mesh cells at runtime. The Fortran code has been shown to be a reasonably good implementation, and is about 33% faster than the C++ version. Further analysis shows that the computation part of the C++ code was

Number of processors	C++/CHAOS++	Fortran/CHAOS
8	295.83	222.91
16	168.71	119.25
32	97.15	70.01
64	62.09	48.59
128	46.62	34.04

Table 1: Execution time (in sec) for Direct Simulation Monte Carlo method on Intel iPSC/860.

much slower than that of the Fortran code, and was the main reason for the slow down. The C++ code is currently being ported to the IBM SP-2, and the performance on the SP-2 will be given in the final paper.

The second application is called Vegetation Index Measurement (VIM). VIM is an application that computes a measure of the vegetation on the ground from a set of satellite sensor images. It has been developed as part of the on-going Grand Challenge project in Land Cover Dynamics at the University of Maryland [17]. The overall project involves developing scalable and portable parallel programs for a variety of image and map data processing applications, eventually integrated with new models for parallel I/O of large scale images and maps. The main focus of the Grand Challenge project is in applying high performance computing to the analysis of remotely sensed imagery, with the initial studies targeted at generating land cover maps of the world's tropical rain forest during the past three decades. The VIM application is an example of one form of processing that will contribute to such studies.

In the VIM application, a user specifies a query region of interest on the ground and a set of satellite images to process. The images may be images of the region taken from the same sensor over a period of time, or images from multiple sensors taken at the same time. The query region is overlaid with a mesh, whose resolution is very likely to be coarser or finer than that of the given images. For each mesh cell, the algorithm selects from the given images the set of data points that spatially intersect with the mesh cell, using a C++ class library that supports spatial operators (currently under development as part of the Grand Challenge project), and computes a vegetation index. CHAOS++ has been linked to this spatial operator class library to implement a parallel version of VIM. In the parallel version, a satellite image is defined as a vector of data vectors, each of which is represented by a class derived from **Mobject**. This was done primarily for efficient implementation of the spatial operators provided by the existing class library. The data vectors are first distributed across the processors by CHAOS++, based upon the given query region, to obtain good load balance. After local computation is carried out, the CHAOS++ library is then used to combine the results for each mesh cell across all processors, since the mesh cells may not be spatially aligned with the satellite data and are thus distributed differently from the satellite data.

The results of running VIM on both the Intel iPSC/860 and the IBM SP-1 are illustrated in Table 2. In this experiment, three satellite images, each of which consists of 600×100 data points, are used to compute the vegetation index for a query region consisting of 120×20 mesh points.

Number of processors	Intel iPSC/860	IBM SP-1
1	334.00	153.46
2	174.29	80.75
4	93.40	43.87
8	55.45	25.72
16	34.73	16.13
32	22.68	10.55

Table 2: Execution time for Vegetation Index Measurement (in sec).

Although the speedup appear to be quite good, it flattens out when more processors are used. This is because the satellite images are quite small, and each processor only gets a small data set. Therefore, in the current implementation, each processor ends up wasting a lot of time processing mesh cells that do not intersect with its assigned pieces of the satellite data. A new algorithm is now being developed to take advantage of the spatial continuity of the mesh cells so that work is done only on mesh cells that intersect with the locally assigned parts of the satellite data.

Another application under development is image segmentation. This application segments a given image into a hierarchy of components, based on the border contrast between the components, and serves as a preprocessing phase of an appearance-based object recognition system developed at the University of Maryland [19]. The hierarchy this preprocessing generates is then used by a high-level vision phase to heuristically combine components from various levels of the hierarchy into possible instances of objects. Further analysis by shape delineation processes would select the combinations that correspond to the locally best instances of objects.

To be more specific, this image segmentation application first classifies all the pixels in a given image into components, based on some given criterion. The algorithm then collapses neighboring components into larger components, based on a series of weaker and weaker criteria on the border contrast. The history of component collapses is kept as a hierarchy of image segmentations for later use as described above. In the parallel implementation, the initial image is represented as an array regularly distributed across the processors. CHAOS++ routines preprocess remote accesses and fetch non-local pixels into a local buffer, so that the computation for clustering pixels into components can be carried out locally. Once formed, the components are stored as an undirected weighted graph, with nodes representing components and edges representing component connectivity. The weight associated with each edge represents the border contrast between a pair of components. A C++ class is derived from the CHAOS++ **Gobject** base class to represent components, and the CHAOS++ mapping structure is used to manage the ghost **Gobject**s of the constructed graph. Communication schedules are generated by CHAOS++ for efficient exchange of component data between real **Gobject**s and ghost **Gobject**s to perform graph contraction.

Table 3 shows the performance results for segmenting a 400×400 image on the Intel iPSC/860. The parallel implementation is based on sequential code that has not been optimized, so the speed-up is given relative to the execution time of running the parallel code on a single node (which is

Number of processors	Execution time (sec)	Speed-up
1	812.29	-
2	384.48	2.11
4	109.36	7.42
8	33.50	24.25
16	11.74	69.19
32	5.66	143.51

Table 3: Performance results for image segmentation on iPSC/860.

faster than the sequential C code). In this experiment, 7,935 components, connected with 17,816 edges, are initially generated by clustering the pixels, and then used to build a hierarchy of height five. In the current implementation, all edges are kept in a linked list, sorted by their weights, and new edges are added to the list as new components are formed. Note that inserting an edge to a sorted linked list takes time proportional to the length of the list. With a large number of edges, list sorting dominates the running time of the algorithm. Partitioning the graph reduces not only the number of components per processor, and thus the number of edges per processor, but also the number of new components per processor, and thus the number edges that need to be added to the list. This results in the super-linear speed up shown in Table 3. A new implementation is now in progress to reduce the cost of maintaining the edges. In this new approach, edges are stored separately by the level in the hierarchy they correspond to, and those that correspond to the same level do not need to be kept sorted. This should reduce the cost for maintaining the edges. Results of the new implementation will be given in the full paper.

5 Related Work

In this section, we briefly discuss related work from the area of object-oriented concurrent programming. Roughly speaking, there are two types of systems that are relevant.

The first type of system augments an existing language, usually C++, with parallel constructs. Parallelism is exploited by both a compiler and an associated runtime system. Examples of this type of system include Mentat [7], CC++ [3], Charm++ [9], and pC++ [13]. However, all of these systems, except pC++, consider program execution as completely unstructured interactions among a set of objects, and thus only support asynchronous communication. pC++ uses a *collection* to represent a structured set of objects, but still provides only asynchronous communication. CHAOS++, in contrast, is a user-level class library, and does not assume any language or compiler support. CHAOS++ uses a preprocessing phase to analyze and optimize communication patterns. The required objects are fetched in an efficient way and cached so that all computation can be carried out locally. Collaborative work is now under way on incorporating CHAOS++ into the pC++ runtime system.

The second type of system is a user-level class library that assumes no special support from the compiler, just like CHAOS++. Examples include P++ [16] and LPARX [10]. These libraries both provide efficient management of dynamic arrays distributed across processors. CHAOS++, however, performs optimization through preprocessing techniques, and provides efficient support for dynamic distributed data structures, including pointer-based data structures.

6 Conclusions and Future Work

We have presented a portable object-oriented runtime library that supports SPMD execution of adaptive irregular applications that contain dynamic distributed data structures. In particular, CHAOS++ supports distributed pointer-based data structures, in addition to distributed arrays, consisting of arbitrarily complex data types. CHAOS++ translates global object references into local references, manages buffer allocation, generates communication schedules, and carries out efficient data exchange. The library assumes no special compiler support, and does not rely on any architecture-dependent parallel system features. Integration with the CHAOS runtime library, for array-based adaptive irregular applications has already been accomplished, and integration with the Multiblock PARTI runtime library [23, 1], for multiple structured grid applications, is currently in progress.

CHAOS++ is targeted as a prototype library that will be used to provide part of the runtime support needed for High Performance Fortran and High Performance C/C++ compilers. We are in the process of integrating CHAOS++ into the runtime software being developed by the Parallel Compiler Runtime Consortium. Finally, we also plan to link CHAOS++ to the Jovian I/O library [2], a library that aims at optimizing the I/O performance of multiprocessor architectures with multiple disks or disk arrays. The resulting software is intended to support disk-based data parallel applications with datasets that do not fit in the processor memory of even very large parallel machines, such as the VIM application from the Grand Challenge in Land Cover Dynamics project.

Acknowledgements

The authors would like to thank Richard Wilmoth at NASA Langley and Bongki Moon at Maryland for the use of their sequential and parallel DSMC codes; also Carter T. Shock at UMIACS and Sam Goward in the Department of Geography at Maryland for their help on the VIM application; and Larry Davis and Claudia Rodríguez at Maryland for many discussions on the image segmentation problem and the use of their sequential code.

The authors are also grateful to the National Institute of Health for providing access to their iPSC/860, and the Cornell Theory Center for providing access to their SP-1.

References

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[2] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1994.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, Department of Computer Science, California Institute of Technology, 1992.

[4] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *AIAA Journal*, 32(3):489–496, March 1994.

[5] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994. Also available as University of Maryland Technical Report CS-TR-3163 and UMIACS-TR-93-109.

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors, general techniques and regular problems*, volume 1. Prentice Hall, 1988.

[7] Andrew S. Grimshaw, Jon B. Weissman, and W. Timothy Stayer. Portable run-time support for dynamic object-oriented parallel processing. Technical Report CS-93-40, Dept. of Computer Science, University of Virginia, Charlottesville, Virginia 22903, July 1992.

[8] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodoscek. Parallelizing molecular dynamics programs for distributed memory machines: An application of the CHAOS runtime support library. Technical Report CS-TR-3374 and UMIACS-TR-94-125, University of Maryland, Department of Computer Science and UMIACS, November 1994. Submitted to IEEE Computational Science and Engineering.

[9] L.V. Kale and Sanjeev Krishnan. CHARM++ : A portable concurrent object oriented system based on C++. In *Proceedings of the 1993 Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, Washington, DC, October 1993. ACM.

[10] S.R. Kohn and S.B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 509–517. IEEE Computer Society Press, May 1994.

[11] A. Krishnamurthy, D.E. Culler, A. Dusseau, S.C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings Supercomputing '93*, pages 262–273. IEEE Computer Society Press, November 1993.

[12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings Supercomputing '93*, pages 588–597. IEEE Computer Society Press, November 1993.

[14] B. Moon and J. Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 176–183. IEEE Computer Society Press, May 1994.

[15] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[16] Rebecca Parsons and Daniel Quinlan. Run-time recognition of task parallelism within the P++ parallel array class library. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, 1993.

[17] Rahul Parulekar, Larry Davis, Rama Chellappa, Joel Saltz, Alan Sussman, and John Towhshend. High performance computing for land cover dynamics. In *Proceedings of the International Joint Conference on Pattern Recognition*, September 1994.

[18] D.F.G. Rault and M.S. Woronowicz. Spacecraft contamination investigation by direct simulation Monte Carlo - contamination on UARS/HALOE. In *Proceedings AIAA 31th Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, January 1993.

[19] Claudia Rodríguez. An appearance-based approach to object recognition in aerial images. Master's thesis, University of Maryland, College Park, MD 20742, 1994.

[20] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[21] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings Supercomputing '94*, pages 97–106. IEEE Computer Society Press, November 1994.

[22] J.P. Singh, T. Joe, J.L. Hennessy, and A. Gupta. An empirical comparison of the kendall square research KSR-1 and stanford DASH multiprocessors. In *Proceedings Supercomputing '93*, pages 214–225. IEEE Computer Society Press, November 1993.

[23] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.