CHAOS++: A Runtime Library For Supporting Distributed Dynamic Data Structures

Chialin Chang
Joel Saltz
Alan Sussman

CRPC-TR95624
November 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

CHAOS++: A Runtime Library for Supporting Distributed Dynamic Data Structures *

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 solely 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.

CHAOS++ is a portable object-oriented runtime library that supports applications using 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 four applications 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 [15]. 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. Throughout this chapter, we assume that there is only one process executing on each processor of the distributed memory machines. This is simply for ease of presentation, as the techniques described can also be applied to parallel executions with multiple processes running on each processor.

*This research was supported by the National Science Foundation under Grant #ASC 9318183, NASA under Grant #NAG 11485 (ARPA Project #8874), and the Office of Naval Research under Grant #N00014-93-1-0158 (Rice University Subcontract #292122193).

Traditionally, parallel applications utilize (multi-dimensional) data arrays, which are often partitioned dynamically during program execution. Optimizations that can be carried out by compilers are thus limited, and runtime analysis is required [38]. 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 [20], particle-in-cell (PIC) codes for computational aerodynamics [28], and computational fluid dynamics [12].

Unfortunately, many existing runtime systems for parallelizing applications with complex data access patterns on distributed memory parallel machines fail to handle pointers. Pointers are frequently utilized by many applications, including image processing, geographic information systems, and data mining, to synthesize complex composite data types and build dynamic complex data structures. We refer to data structures constructed with pointers as *pointer-based data structures*. Without proper support for pointers, existing runtime systems only allow the transfer of primitive data types, such as integers and floating point numbers, and of simple objects that contain no references to other objects. Most of these runtime systems also rely on the existence of global indices, which makes them applicable only to distributed arrays.

CHAOS++ is a runtime library targeted at object-oriented applications with dynamic communication patterns. It subsumes CHAOS [14], which is a runtime library developed to efficiently support applications with irregular patterns of data accesses to distributed arrays. In addition to providing support for distributed arrays through the features of the underlying CHAOS library, CHAOS++ also provides support for distributed pointer-based data structures, and allows flexible and efficient data exchange of complex data objects among processors.

CHAOS++ is motivated by the way pointers are often used in many real applications. In these applications, hierarchies of data types are defined, such that ones at the higher levels serve as containers for the ones at lower levels. Pointers are often used by container objects to point to the objects they contain. Objects that are *solely* contained within such a container object are referred to as *sub-objects*. A sub-object is effectively part of its container object, though it does not necessarily occupy memory locations within that of its container object. Objects of data types at the top of the hierarchy (i.e. objects of the outermost container class) can further be connected through pointers, forming complex 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 runtime optimization techniques are required.

```
class Pixel {                 // a single pixel of an image
  int x, y;                   // x,y coordinates
};
class Region {                // a region consisting of pixels
  int    num_pixels;          // number of pixels
  Pixel  *pixels;             // an array of pixels
  int    num_neighbors;       // number of adjacent regions
  Region **neighbors;         // list of pointers to adjacent regions
};
class Region_Map {            // a connected graph consisting of regions
  Region *region;             // a pointer to some Region in the graph
};
```

Program 1.1: An example of complex objects and pointer-based data structures.

As an example, Program 1.1 shows the declaration of a set of C++ classes, which can be used to describe how pixels of a given image are clustered into regions, and how regions containing pointers to adjacent regions form a map. In this example, the `Region` class is implemented as a container class for the `Pixel` class, so that a `Pixel` is a sub-object of a `Region`. Since different regions may consist of different numbers of pixels, the `Region` class uses a pointer to an array of its constituent pixels. A set of regions interconnected with pointers then form a graph, defined by the class `Region_Map`.

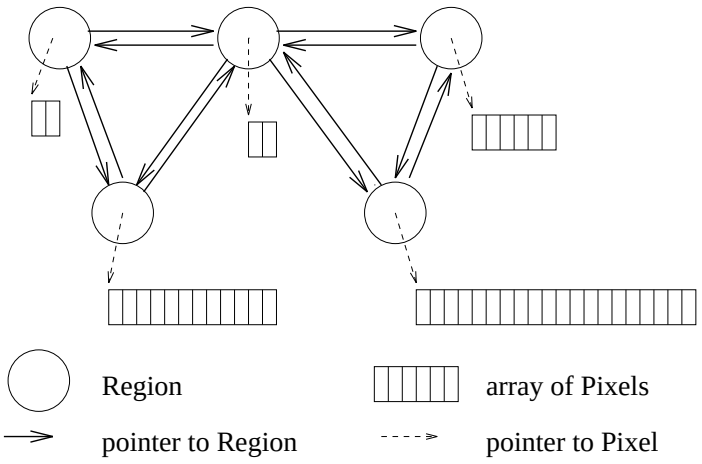

Figure 1: A graph consisting of Region objects.

Figure 1 gives an example of such a graph. When a graph is partitioned among multiple processors, the runtime system must be able to traverse pointers to support remote data accesses. Without efficient support for pointers, parallel applications utilizing such complex data structures cannot achieve good performance on distributed memory architectures.

CHAOS++ is implemented as a C++ class library, and can be used directly by application programmers to parallelize applications with adaptive and/or irregular data access patterns. 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 several distributed memory machines, including 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 [25, 29, 41].

The remainder of the chapter is structured as follows. In Section 2 we discuss how CHAOS++ supports complex objects and distributed arrays, and in Section 3, we discuss how distributed pointer-based data structures are supported in CHAOS++. Section 4 describes the implementation of the library. Performance results for the polygon overlay problem and three complete applications are presented in Section 5. Section 6 discusses some related work, and we present conclusions in Section 7.

2 Runtime Support for Distributed Arrays

CHAOS++ provides runtime support for distributed arrays through use of the Maryland CHAOS [37] and Multiblock Parti libraries [42]. The CHAOS library is employed as the underlying support for arrays with irregular distributions, while Multiblock Parti is used to support regular data distributions. Neither library is visible to the user of CHAOS++; they are used internally to provide efficient, high-performance runtime support for arrays distributed across the processors of a distributed memory machine.

The CHAOS runtime library has been developed to efficiently support applications with irregular patterns of access to distributed arrays, running on distributed memory machines. In these problems, arrays are frequently partitioned in an irregular manner for performance reasons (e.g. to reduce communication costs or to obtain better load balance), and accesses to off-processor array elements are carried out by message passing. Furthermore, the data access patterns to the arrays are not known until runtime, since array elements are accessed through one or more levels of indirection. Program 1.2 illustrates a typical irregular loop. The data access pattern is determined by arrays, `ia` and `ib`, whose values are not known until runtime, and we call them *indirection arrays*.

```
double x[max_nodes], y[max_nodes];          // data arrays
int ia[max_edges], ib[max_edges];           // indirection arrays

for (int i = 0; i < max_edges; i++)         // a parallel loop
   x[ia[i]] += y[ib[i]];
```

Program 1.2: An example with an irregular loop.

For the loosely synchronous applications we have described in the introduction, 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* [27, 38]. During program execution, the CHAOS inspector routines examine the data references expressed in the indirection arrays, given in global indices, and convert them into host processor numbers and local indices. This is done by a lookup into a *translation table* that CHAOS constructs for arrays partitioned in an irregular fashion. The translation table 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 on all processors, or distributed across the processors. Duplicate irregular references are removed through simple software caching, and unique references are aggregated 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 [39].

2.1 Mobile Objects

CHAOS++ defines an abstract data type, called `Mobject`, for *mobile objects*. These are objects that may be transferred from one processor to another, so they must know how to pack and unpack themselves to and from a message buffer. In general, the object model that CHAOS++ supports is one in which an object is owned by one processor, but other processors may possess shadow copies of an object, as will be discussed in Section 3.2. This implies that a distributed array of objects is treated by CHAOS++ as multiple objects, so that it can be distributed across multiple processors.

The `Mobject` class contains two pure 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. CHAOS++ employs the user-provided `pack/unpack` functions to move or copy `Mobject`s between processors. In C++, virtual functions allow the dynamic binding of a function call, so that the `pack` or `unpack` functions actually invoked will be based on the type of the object at runtime. Consequently, CHAOS++ can always invoke the proper methods with `Mobject::pack()` and `Mobject::unpack()`.

An implication of requiring the user to provide `pack` and `unpack` functions for all `Mobject`s is that CHAOS++ does not allow distributed arrays of C++ base types (e.g. `double`, `int`, etc.), because C++ does not allow a user to define member functions for base types. One way for a user to implement such a distributed array would be to define a class derived from `Mobject` consisting solely of a member with the base type, and then provide the `pack` and `unpack` functions for that class. In the applications we have investigated so far, this is not a major problem, because all the distributed arrays have been arrays of complex structures.

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. 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 message buffer, packed by the sender's `pack` function, 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 important because a local pointer on one processor is not valid across processor boundaries on distributed memory architectures. This means that sub-objects must be dynamically allocated on the receiving processor, and the pointers to the sub-objects must be initialized to point into local memory. In other words, an object transferred to another processor may not retain the same pointer values for its sub-objects, but the new pointers will point at its new sub-objects with the same user data values (and connectivity, through pointers) as in the original object.

2.2 Partitioning Data Arrays

The CHAOS runtime support library [32, 37] contains procedures that

- support static and dynamic distributed array partitioning,
- partition loop iterations and indirection arrays,
- remap arrays from one distribution to another, and
- carry out index translation, buffer allocation and communication schedule generation.

The partitioning of arrays during the data distribution phase can be done with partitioners supplied by either CHAOS or the application programmer. CHAOS supports a number of parallel partitioners that use heuristics based on spatial positions, computational load, connectivity, etc. The partitioners return an irregular assignment of array elements to processors; this is stored as a CHAOS construct called the *translation table*.

When an array is irregularly distributed, a mechanism is needed to retrieve required elements of that array. CHAOS supports a translation mechanism using a data structure called the *translation table*. A translation table lists the home processor and the local address in the home processor's memory for each element of the irregularly distributed array. In order to access an element $A[m]$ of distributed array A, a translation table lookup is necessary to find out the location of $A[m]$.

A translation table lookup, which is aimed at computing the home processor and the offset associated with a global distributed array index, is known as a *dereference request*. Any preprocessing aimed at communication optimizations needs to perform dereferencing, since it is required to determine where elements reside.

Several considerations arise during the design of data structures for a translation table. Depending on the specific parameters of the problem, there is usually a trade-off involving storage requirements, table lookup latency and table update costs. Of these, table lookup costs are of primary consideration in adaptive problems, since preprocessing must be repeated frequently, and must be efficient.

The fastest table lookup is achieved by maintaining an identical copy of the translation table in each processor's local memory. This type of translation table is a *replicated translation table*. Clearly, the storage cost for this type of translation table is $O(NP)$, where P is the number of processors and N is the array size. However, the dereference cost in each processor is constant and independent of the number of processors involved in the computation.

Due to memory considerations, it is not always feasible to place a copy of the translation table on each processor. The approach taken in these cases is to distribute the translation table between processors. This type of translation table is a *distributed translation table*. Earlier versions of CHAOS supported a translation table that was distributed between processors in a blocked fashion. This is accomplished by distributing the translation table by blocks, i.e., putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc. When an element $A[m]$

of the distributed array A is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $(m \times P)/N$. The dereference requests may require a communication step, since some portions of the translation table may not reside in the local memory. Similarly, table re-organization also requires interprocessor communication since each processor is authorized to modify only a limited portion of the translation table.

Besides supporting replicated and distributed translation tables, CHAOS also supports an intermediate degree of replication with *paged translation tables*. In this scheme, the translation table is divided into pages, and pages are distributed across processors. Processors that refer to a page frequently receive a copy of the page, making subsequent references local. A more detailed description of this scheme is presented in Das *et al.* [14].

For applications with more regular data access patterns, the translation tables employed by the CHAOS library are not required. Instead, *distributed array descriptors* that contain complete information about the portions of the arrays residing on each processor can be generated. For regularly distributed data, the data distribution across the processors (e.g. block, cyclic), and its associated descriptor, can be represented compactly, for example by using regular section descriptors [19]. This form of descriptor has been used in the Multiblock Parti library [1, 2, 42], which CHAOS++ uses to partition data and generate communication schedules for regularly distributed arrays. Multiblock Parti has been designed to support the parallelization of applications that work on multiple structured regular grids, such as those that arise in multiblock and/or multigrid codes. While Multiblock Parti produces distributed array descriptors at runtime, because it allows the sizes of arrays to be determined at runtime, compilers for data parallel languages such as Fortran 90D [5] and High Performance Fortran (HPF) [22], can produce such descriptors at compile time. In either case, this compact representation allows the descriptors to be replicated across all processors, so that communication schedules can be built without additional communication being performed, as is required when using a CHAOS distributed translation table.

Multiblock Parti conforms to the inspector/executor model of parallel computation, building communication schedules in the inspector phase and performing communication in the executor phase. The library supports two forms of inspectors: one for intra-array communication, for handling the communication required because a single array is partitioned across processors, and one for inter-array communication, for handling *regular section moves* between different arrays or different portions of the same distributed array. A regular section move copies a regular section (e.g. as represented by Fortran 90-style triplets) of one distributed array into a regular section of another distributed array, potentially involving changes of offset, stride and index permutation. Intra-array communication is handled by allocation of extra space at the beginning and end of each array dimension on each processor. These extra elements are called *overlap*, or *ghost*, cells [16]. Both inspectors produce the same form of communication schedule, which can be used by the Multiblock Parti collective data movement routines.

2.3 Data Movement

The preprocessing, or inspector, phase of a program parallelized using CHAOS results in a data structure called a *communication schedule* [13], which stores the send/receive patterns of off-processor elements. The computation, or *executor*, phase uses the schedules to carry out communication. A communication schedule is used to fetch off-processor elements into a local buffer and to scatter these elements back to their home processors after the computational phase is over. Communication schedules determine the number of communication startups and the volume of communication, so it is important to optimize them.

CHAOS also supports specialized communication schedules [39]. For some adaptive applications, particularly those from the particle-in-cell domain, there is no significance attached to the placement order of incoming array elements. Such application-specific information can be used to build much cheaper *light-weight* communication schedules. In inspectors for such applications, index translation is not required, and a permutation list need not be generated for the schedule data structure. Besides being faster to construct, light-weight schedules also speed up data movement by eliminating the

need for rearranging the order of incoming off-processor elements. Light-weight schedules have been used in the parallelization of the Direct Simulation Monte Carlo code discussed in Section 5.2.

While communication schedules store data send/receive patterns, the CHAOS data transportation procedures actually move data using these schedules. The procedure `gather` can be used to fetch a copy of off-processor elements from the processors that own the elements into a local buffer. The procedure `scatter` can be used to send off-processor elements from a local buffer to the owners.

A set of data movement primitives has also been developed that can perform irregular communications efficiently using the light-weight communication schedules. The procedure `scatter_append` can be used to perform data movement when the order in which elements are scattered into their destination processors is not relevant (i.e. when using a light-weight schedule). While the cost of building light-weight schedules is much less than that of regular schedules, light-weight schedules and data migration primitives still provide all the CHAOS communication optimizations, such as aggregation and vectorization of messages. Multiblock Parti also has data transportation routines similar to those in CHAOS for performing the communication specified by a schedule produced by either the ghost cell fill or regular section move routines. A more detailed description of the CHAOS procedures can be found in Saltz *et al.* [37].

2.4 An Example

We now discuss how to parallelize Program 1.2 using the CHAOS runtime library. The transformed code for Program 1.2 that carries out the inspector and executor is shown in Program 1.3. The arrays `ia` and `ib` contain the globally indexed reference patterns used to access arrays `x` and `y`, respectively. Because some referenced data might reside on other processors, non-local values must be transferred to local memory. However, communication between two processors involves a non-trivial communication latency, or message startup cost, on distributed memory machines. Vectorizing communication reduces the effect of communication latency and software caching reduces communication volume. To carry out either optimization, it is helpful to have *a priori* knowledge of data access patterns. In irregular problems, it is generally not possible to predict data access patterns at compile time. For example, in Program 1.2, the values of indirection arrays `ia` and `ib` are known only at runtime.

```
// Create the required schedules (Inspector)
localize(dist_of_x, schedule_ia, ia, local_ia, n_local_iterations, &off_proc_x)
localize(dist_of_y, schedule_ib, ib, local_ib, n_local_iterations, &off_proc_y)
// The actual computation and communication (Executor)
zero_out_buffer(&x[begin_buffer_x], off_proc_x)
gather(&y[begin_buffer_y], y, schedule_ib)
for (int i = 0;  i < n_local_edges; i++)
  x[local_ia[i]] = x[local_ia[i]] + y[local_ib[i]]
scatter_add(&x[begin_buffer_x], x, schedule_ia)
```

Program 1.3: Node code for simple irregular loop.

In source Program 1.2, values in array `x` are updated using the values in array `y`. Hence, a processor may access an off-processor array element of `y` to update an element of `x`. A processor may also need to update an off-processor array element of `x`, if loop iteration `i` is not assigned to the processor owning `x[ia[i]]`. The two *localize* function calls in Program 1.3 perform the inspectors for arrays `x` and `y`. The goal of the inspector is to compute the schedules to both pre-fetch off-processor data items before executing the loop and carry out off-processor updates after executing the loop. Hence, two schedules are computed in the inspector:

1. *a gather schedule* – a communication schedule that can be used to fetch off-processor elements of `y`, and

2. *a scatter schedule* – a communication schedule that can be used to send updated off-processor elements of `x`.

Copies of the off-processor elements are stored in the on-processor buffer area. The on-processor buffer for off-processor data array elements immediately follows (in physical memory) the on-processor data for that array. For example, the buffer for data array `y` begins at `y[begin_buffer_y]`, and is of size `off_proc_y`. The value of `off_proc_y` is is returned by the *localize* function call. The CHAOS procedure *localize* also translates `ia` and `ib` to `local_ia` and `local_ib`, respectively, so that valid references are generated when the loop is executed. All references to off-processor elements are translated so that they point to the on-processor buffer. The communication schedule returned by *localize* is used by the CHAOS `gather` operation to fetch the off-processor data and store them in the on-processor buffer. The data are stored in a way such that execution of the loop using `local_ia` and `local_ib` accesses the correct data. CHAOS data movement routines, such as `gather`, are collective operations. That is, all processors must participate to complete the operation, but synchronization occurs only through matching sends and receives between processors. No explicit barrier occurs either in the CHAOS data movement routines, or needs to be inserted by the application programmer. The CHAOS `scatter_add` procedure is a variant of the basic scatter operation that adds values from the on-processor buffer into the corresponding off-processor array elements, instead of overwriting them.

3 Runtime Support for Distributed Pointer-Based Data Structures

In addition to arrays, CHAOS++ is designed to effectively support applications that contain pointer-based data structures. It is more difficult to support distributed pointer-based data structures because they usually have more dynamic behavior than arrays. In this section, we first describe the type of runtime support required for distributed pointer-based data structures, and then briefly discuss how CHAOS++ provides the required functionality. The approach that CHAOS++ takes relies heavily on class inheritance, as supported by the C++ language.

3.1 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. In distributed arrays, each element is associated with a global index, which can be used as a unique identifier for that element. In pointer-based data structures, however, since elements (objects) may be added to and removed from the data structures dynamically, no static global names or indices are associated with the elements, and accesses to those elements are done through pointer dereferencing. It is therefore not feasible for the runtime system to rely on the existence of global indices. Furthermore, partitioning a pointer-based data structure may assign two elements connected via pointers to two different processors, raising the need for *global pointers*. A global pointer, as supported by several languages, including Split-C [11], CC++ [7], and pC++ [26], 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.

In CHAOS++, these problems are addressed by introducing an abstract data type, called *Globally Addressable Objects*, which we now discuss in detail.

3.2 Globally Addressable Objects

Elements in a pointer-based data structure are linked and accessed through pointers. As we previously described, global pointers are needed on a distributed memory parallel machine 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 whenever a global pointer is dereferenced,

the necessary interprocessor communication is generated transparently to the application program. This approach, however, does not allow collective communication, which is an important technique for achieving high performance using a loosely synchronous execution model. Furthermore, dereferencing a global pointer requires a conversion between a reference to a remote object and a reference to a local buffer. This imposes additional overhead with *every* dereference of a global pointer. It is more desirable to perform the conversion only when the binding between the global pointer and the local buffer changes.

Instead of defining a class for global pointers, CHAOS++ supports global pointers by defining an abstract C++ base class for *globally addressable objects*, or `Gobjects`. A `Gobject` is an object whose ownership is assigned to one processor, but allows copies to reside on other processors. The copies are referred to as *ghost objects*, and each processor other than the one assigned ownership of the `Gobject` may have a local copy of the `Gobject` as a ghost object. Figure 2 illustrates an example of partitioning a graph between two processors. The dashed circles represent ghost `Gobjects`. Each `Gobject` has a member 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). The ghost object is used to cache the contents of the remote counterpart, so that once the ghost object is 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. Deciding when to update a ghost object from a real object is made by the application. This description implies that all `Gobjects` must also be CHAOS++ `Mobjects`, to support transfer of data between real and ghost objects that are owned by different processors.

In the object model supported by CHAOS++, a pointer-based data structure is viewed as a collection of `Gobjects` 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. In the partitioned data structure, pointers between two `Gobjects` residing on the same processor are directly represented as C++ pointers. Pointers to `Gobjects` residing on other processors correspond to global pointers, and are represented as C++ pointers to local ghost object copies of the remote `Gobjects`. The outermost layer of ghost objects on a processor can be thought of as the boundary of the distributed data structure assigned to that processor, and they would be the nodes where a local traversal of the data structure terminates. In Figure 2, note that 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 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 constructed by CHAOS++ for each distributed pointer-based data structure on each processor, to manage the ghost objects residing on that processor. The mapping structure maintains the possible remote accesses from the local processor by creating a list of all the ghost objects. The mapping structure also records for each ghost object the processor number and the local address of the remote object that the ghost object represents. The mapping structure is used during the inspector phase of the parallel execution for translating global references into processor and local address pairs to generate communication schedules. In Figure 2, 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 `Gobjects` and ghost `Gobjects` in the executor phase.

3.3 Building Distributed Pointer-Based Data Structures

A distributed pointer-based data structure is defined by its nodes and edges, as well as by how it is partitioned among the processors. To enable the parallel execution of an application that contains a distributed pointer-based data structure, the program must be able to build the distributed data structure based on application-specific information. This is usually done in two steps: all processors

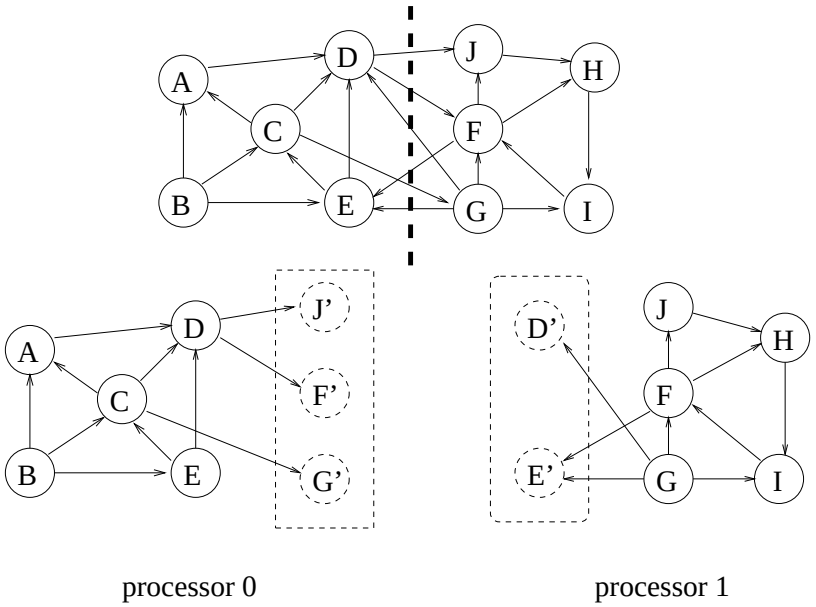

Figure 2: Partitioning a graph between two processors. At top is the original graph, partitioned at the vertical bar, with the resulting processor assignment shown at bottom.

first construct their local connected components, and then compose those components to form the final distributed data structure. In general, there are two possible scenarios. One is a situation where each node in the pointer-based data structure is associated with a globally unique identifier. The other is the case where no such identifier exists. In both cases, CHAOS++ provides primitives to assist in the construction of a distributed pointer-based data structure, and to create the corresponding mapping structure.

3.3.1 Nodes With Unique Identifiers

In many applications, each node in a pointer-based data structure is associated with a globally unique identifier. In this case, the nodes can be named by their associated identifiers, the edges can be specified by the identifiers of their two end points, and the partitioning information can be described by pairs of processor numbers and node identifiers. One such example is an unstructured CFD code, where a node table is used to record all the node information for the graph, including initial values, node identifiers and the assigned processor numbers, and an edge table is used to specify the graph connectivity. When node identifiers are available, CHAOS++ provides a hash table on each processor that stores, for each node of the local component of the data structure, the node identifier, its local address, and its assigned processor number if available. The records are hashed by the node identifiers, so accesses through node identifiers are fast. Program 1.4 demonstrates how the CHAOS++ hash table assists in constructing a distributed data structure. Applications can store information about their distributed pointer-based data structures in any format. For simplicity, the application in this example uses replicated C++ arrays `Node_Table` and `Edge_Table`.

```
class Graph_Node : Gobject { ... };
chaosxx_hash_table<Graph_Node> htable;      // declare a CHAOS++ hash table
Graph_Node  *node, *from_node, *to_node;    // pointers to graph nodes

// assume a replicated node table and edge table
// go through the node table
for (i=0; i<num_nodes; i++) {
   if (Node_Table[i].owner_processor == MY_PROC_ID)
```

```
      node = new Graph_Node(...);           // create and initialize the node
   else
      node = NULL;

   // register the node with CHAOS++ hash table
   htable.add(Node_Table[i].node_id, node, Node_Table[i].owner_processor);
}

// go through the edge table
for (i=0; i<num_edges; i++) {
   j = Edge_Table[i].from_node_id;
   k = Edge_Table[i].to_node_id;
   if ((from_node = htable.get_local_pointer(j)) != NULL) {
      // an edge from a local node
      if ((to_node = htable.get_pointer(k)) == NULL) {
 // a ghost node needs to be created
 to_node = new Graph_Node(...);    // create a ghost node

 // register the node with CHAOS++ hash table
 htable.set_pointer(k, to_node);
      }

      // add a pointer from *from_node to *to_node
      from_node->neighbor[from->number_of_neighbors++] = to_node;
   }
}

// create the mapping structure for CHAOS++
TPMapping<Graph_Node> *map = htable.create_map();
```

Program 1.4: Constructing a distributed graph with a CHAOS++ hash table.

The program consists of three steps. In the first step, the program scans through the node table and registers node information into the hash table. Nodes assigned to the local processor are created and properly initialized in this step. Nodes that are not assigned to the local processor are marked by recording their assigned processor numbers. If a node is known to be remote, but the owner processor is not yet known, the constant `CHAOSXX_REMOTE` can be used as the owner processor. CHAOS++ uses this information to bind local copies with remote copies, in the final step of the program. Exact knowledge of the owner processors of remote nodes makes that process more efficient, since a single request suffices to locate a remote node. When the owner processor is not known to the runtime system, locating a remote node may require the local processor to send out multiple requests. In the second step, the program scans through the edge table and creates the specified pointers. Only edges that originate from a node assigned to the local processor are of interest. The CHAOS++ hash table is used to find the addresses of the end nodes, specified by the node identifiers stored in the edge table. Nodes that are assigned to the local processor are created in the first step, and their addresses can be retrieved from the hash table through their node identifiers. Nodes that are not assigned to the local processor can be created as ghost objects and registered with the hash table upon their first appearances in the edge table. At the end of the second step, each processor has constructed its local component of the distributed data structure, containing both objects assigned to the local processor and ghost objects. In the final step of Program 1.4, an appropriate mapping structure is constructed by CHAOS++. The mapping structure is of type `TPMapping<Graph_Node>`, and records the association between ghost objects and real remote objects, using the information stored in the hash table. This is done via a collective communication, in which all processors exchange the node identifiers and the local addresses stored in their CHAOS++ hash tables.

3.3.2 Nodes With No Unique Identifiers

For applications with nodes that do not have unique identifiers, CHAOS++ provides another interface. Since there is no natural way to name the nodes in distributed pointer-based data structures, the connectivity of the data structures in these applications is usually induced from application-dependent information. For example, the initial graph built by the image segmentation problem, which is discussed in Section 5.4, is defined by the input image. In this case, the CHAOS++ library assumes that each processor running the target application is able to build its assigned component of the distributed data structure, and has the information necessary to "linearize" its boundary nodes with the components on the other processors. CHAOS++ primitives can then be used to associate the corresponding boundaries, to compose the local components into a global data structure and generate an appropriate mapping structure.

To be more specific, each processor i provides, for each other processor j, two lists of node pointers, $local_{ij}$ and $ghost_{ij}$. $local_{ij}$ consists of the nodes that are owned by processor i but have ghost objects on processor j, and $ghost_{ij}$ consists of the ghost nodes residing on processor i that correspond to real objects on processor j. To compose the components between two processors correctly, node pointers in the corresponding lists must be listed in the same order. That is, node pointers in $local_{ij}$ must match exactly with the node pointers in $ghost_{ji}$, one-to-one, and those in $ghost_{ij}$ must match with those in $local_{ji}$, one-to-one. As an example, to compose the two components on processor 0 and 1 in Figure 2, the processors would construct the following matching boundaries.

$$\begin{array}{lll} processor\ 0: & local_{01} = \{D, E\} & ghost_{01} = \{J', F', G'\} \\ processor\ 1: & local_{10} = \{J, F, G\} & ghost_{10} = \{D', E'\} \end{array}$$

The ordering of the lists implies that nodes D and E on processor 0 are associated with ghost nodes D' and E', respectively, on processor 1, and similarly, nodes J, F and G on processor 1 are associated with ghost nodes J', F' and G', respectively, on processor 0.

Given the information for the boundaries between every pair of processors, the CHAOS++ runtime library is able to associate real objects with their corresponding ghost objects (i.e. compute the local addresses on each processor, for later communication), through collective communication, and store that information in the mapping structure.

3.4 Data Movement Routines

The data transfer between real `Gobjects` and ghost `Gobjects` is carried out by the CHAOS++ data movement routines. CHAOS++ allows processors to either update ghost objects with data from their corresponding remote objects on other processors (as in a CHAOS gather operation), or modify the contents of remote objects from the contents of ghost objects (as in a CHAOS scatter operation). The data movement routines use the `pack` and `unpack` functions of `Mobjects` to enable deep copies. Recall that all `Gobjects` are also `Mobjects`. The communication schedules generated from the mapping structure, constructed using either of the two methods discussed in Section 3.3, ensure that neither polling nor interrupts are needed at the receiving processors, so that communication can be performed efficiently.

3.5 An Example

An example of using the CHAOS++ runtime library is given in Program 1.5. In this example, a parallel computation is applied to a distributed graph consisting of objects defined by a user-defined class, Graph_Node. The user derives class Graph_Node from the CHAOS++ `Gobject` base class, and provides implementations of the two virtual functions `pack` and `unpack`.

```
class Graph_Node : public Gobject {
  ⋮
  public:
```

```
    int num_neighbors() const;
    Graph_Node* neighbor(int i);
    int& data1();
    int data2() const;
    void pack(Buffer&);     void unpack(Buffer&);
};

// build the partitioned graph, and create the appropriate mapping structure,
Graph_Node *root = ...;  TPMapping<Graph_Node> *map = ...;

// the inspector: build a communication schedule from the mapping structure
PSchedule<Graph_Node> *sched = map.schedule();

for (int i=0; i < num_iterations; i++) {
  // the executor: transfer data from remote real objects
  // to local ghost objects with CHAOS++ gather function
  sched->gather();

  // a local graph traversal, starting from the root of the local
  // component, to compute for every Graph_Node object,
  // the sum of data2 of its neighboring Graph_Node objects
  mark_all_nodes_as_not_visited(root); local_traversal(root);

  ...     // some other computation that might update the value of data2
}

void local_traversal(Graph_Node *node)
{
  if (node == NULL || node->isGhostObject() || node->been_visited())
     return;
  node->mark_visited();
  // computation on node data values
  for (int i=0; i<node->num_neighbors(); i++) {
      Graph_Node *next = node->neighbor(i);
      node->data1() += next->data2();
      local_traversal(next);
  }
}
```

Program 1.5: Parallel computation over a partitioned graph.

Every `Graph_Node` object contains a list of pointers to other `Graph_Node` objects that it interacts with. The member function `num_neighbors` returns the number of `Graph_Node` objects a `Graph_Node` object interacts with, and the member function `neighbor(i)` returns a pointer to the i-th neighboring `Graph_Node` object.

The program first constructs the distributed graphs, using either of the two methods described in Section 3.3, and generates a mapping structure. For simplicity, we assume that each processor is assigned one connected component, pointed to by a `Graph_Node` pointer, `root`. CHAOS++ then analyzes the mapping structure and builds a communication schedule to allow efficient data transfer between real and ghost objects. The for-loop traverses the local component of the distributed graph on each processor, to perform computation on the graph nodes. In this example, each `Graph_Node` object accumulates into its data member `data1` the values of data member `data2` from its neighboring `Graph_Node` objects. Remote data accesses are required to perform this computation over a distributed graph.

To achieve good performance, the loop body is thus broken into a communication phase and a computation phase. During the communication phase, all ghost nodes are filled with data from

their remote objects by invoking the CHAOS++ `gather` routine with the communication schedule generated before the for-loop. The computation phase follows, with every processor traversing its local component of the distributed graph to apply the computation to each of the `Graph_Node` object it owns. Program 1.5 uses a recursive depth-first search algorithm to traverse the local graph. The local traversal of the graph terminates when encountering either the end of the graph (`node == NULL`), a ghost object (`node->isGhostObject() == TRUE`), or a node that has already been visited (`node->has_visited() == TRUE`). All the computations applied to the local graph nodes are performed locally, since all remote data required by the computation was fetched during the communication phase.

4 Implementation

CHAOS++ provides support for efficient data transfer between remote objects and local buffers. This requires the runtime system to be able to translate a global reference into a processor number and a local reference on that processor. This process is referred to as a *dereference*, and is discussed in Section 4.1. After dereferencing, communication optimizations are applied and the result is a communication schedule, which is later used by the CHAOS++ data transfer routines to carry out interprocessor communication. CHAOS++ communication schedules and data transfer routines will be discussed in Section 4.2.

4.1 Dereferencing Global References

The purpose of dereference operations is to determine the local address of the object that corresponds to a given global reference. For distributed arrays, the most convenient global references are global indices, which often appear as indirection arrays in programs with irregular array data access patterns. CHAOS++ uses a translation table to record the association between global indices and local addresses. On the other hand, since CHAOS++ supports global pointers implicitly through `Gobjects`, pointers to `Gobjects` that are actually ghost objects *are* the global references used by an application. CHAOS++ uses its mapping structure to carry out dereference operations for distributed pointer-based data structures.

4.1.1 Translation Tables

In distributed memory machines, large data arrays need to be partitioned between the local processor memories. These arrays can be partitioned with a regular distribution, such as a blocked distribution or a cyclic distribution. But it is frequently advantageous to partition them in an irregular manner to minimize interprocessor communication, or to improve load balance. Once the distributed arrays have been partitioned between processors, each processor is assigned a set of globally indexed distributed array elements, and CHAOS++ uses a translation table to store that information.

A translation table stores the information that is required for the dereference operation. For a regular array distribution, such as a block or cyclic distribution, a simple formula is stored that allows the dereference to be performed by local computation. For an irregular distribution, an entry that lists the *home processor* and the *local address* in the home processor's memory is created for every element of the distributed array, and a dereference is performed by table lookup. Since local addresses of array elements are described in terms of offsets from a local base address, rather than absolute memory addresses, a translation table can be shared by multiple distributed data arrays with the same distribution.

A translation table can be replicated on all the processors, in which case a dereference can be performed locally on each processor. But with an irregular distribution, the translation table has the same size as the distributed array that it describes, so could be quite large. To minimize memory usage, a translation table can be distributed among the processors, in which case interprocessor communication is required for dereference. CHAOS++ uses the *paged* translation table [14] described

in Section 2.2. The advantage of a paged translation table is that each processor is allowed to choose which pages to replicate based on the access patterns to its distributed arrays, so that the communication cost for dereference can be minimized. Das *et al.* [14] discuss the performance behavior of two applications using paged translation tables, and to no surprise, performance improves as more of the table entries are cached locally.

4.1.2 Mapping Structures

Partitioning a pointer-based data structure among processors, as described in Section 3.1, generates global pointers that point from one processor to another. A mapping structure for such a distributed pointer-based data structure effectively records the incoming pointers and the outgoing pointers for each processor. That is, it keeps track of all the ghost objects on the local processor, as well as all the local objects that have corresponding ghost objects residing on other processors. The space required to store the mapping structure on each processor is proportional to the number of ghost objects on the local processor, and the number of local objects that have corresponding ghost objects residing on remote processors. With a good partitioning scheme that minimizes the number of edges between `Gobjects` assigned to different processors, the space overhead should be quite small.

The association between real objects and ghost objects is obtained by the CHAOS++ runtime system, either through the use of global node identifiers, when they are available, or through matching boundaries (see Section 3.3.2). By exchanging the recorded information between processors, the runtime system is able to obtain the processor number and local address of the corresponding remote object for each ghost object, and records the information in the mapping structure. The information can then be used to dereference pointers to ghost objects.

Furthermore, since the objects in a distributed pointer-based data structure can only be accessed by traversing pointers, the mapping structure effectively records all potential remote data accesses. With the information available for dereferencing pointers to ghost objects, the CHAOS++ runtime system can generate a communication schedule from a mapping structure, to generate efficient data transfers between the owners of `Gobjects` and the ghost copies of the `Gobjects`.

4.2 Data Movement

CHAOS++ provides efficient and flexible data movement routines to transfer data between remote objects and local buffers. This is carried out in an inspector and an executor phase. The inspector phase analyzes the communication requirements and applies several optimizations, such as removing duplicate references and aggregating messages, to create a communication schedule. Then, the executor phase uses the schedule to move data efficiently between processors via the CHAOS++ data transportation routines.

4.2.1 Communication Schedules

A communication schedule records, for each processor, the information required to fulfill a given communication request. In particular, a schedule for processor p stores the following information:

- send list – an array of lists of references, one list per processor, to local objects that must be sent to other processors,
- send size – an array that specifies the numbers of objects to be packed into each of the out-going messages from processor p to other processors,
- fetch size – an array that specifies the number of objects to be unpacked from each of the incoming messages to processor p from other processors.
- permutation list – an array of references to local buffer space that specifies the placement of incoming off-processor objects,

The send list on processor p stores, for each other processor, a list of local objects that must be packed into a message and sent to that processor, with the send size specifying the number of objects to be packed into each message. The fetch size gives the number of objects that each incoming message contains. After receiving all the incoming messages, the permutation list directs the placement in local memory of each object unpacked from the incoming messages.

The CHAOS++ runtime library provides communication schedules for both distributed arrays, whose elements can be accessed by indices, and pointer-based data structures, whose elements cannot be accessed by indices. Both types of communication schedules have the same structure, with the only difference being the way information is represented in the schedules.

CHAOS++ uses CHAOS communication schedules for distributed arrays [14], which refer to local objects and buffer space by local indices. The starting addresses of the distributed arrays involved are needed only when the actual data transfer occurs. Due to the abstraction provided by local indices, communication schedules for distributed arrays can be shared by different arrays that have the same distribution and are accessed via the same indirection arrays. CHAOS++ also provides, through the underlying CHAOS library, routines to build *incremental* schedules, which for a given indirection array, only fetch additional objects that have not been brought in by previously built schedules. To minimize message startup overhead, CHAOS++ also provides the capability of merging existing schedules.

In contrast, communication schedules for distributed pointer-based data structures cannot rely on local indices. The schedules must use local pointers to refer to local objects and buffer space, and, as a consequence, they cannot be shared among different data structures, even those with the same distribution of objects. A schedule of this type can be built out of a mapping structure, as described in Section 4.1.2.

4.2.2 Data Transfer

CHAOS++ data movement routines use the communication schedules described in Section 4.2.1 to perform the required interprocessor communication. They can be used either to obtain copies of data stored in specific off-processor memory locations (i.e. a *gather* operation), or to modify the contents of specified off-processor memory locations (i.e. a *scatter* operation). All the data transfer routines use the same basic algorithm:

1. for each remote processor that expects data from the local processor, pack local objects specified by the communication schedule into a message,
2. send to remote processors the sizes of the out-going messages,
3. allocate buffers for incoming messages,
4. send to, and receive from, remote processors messages that contain objects packed in step 1,
5. unpack all objects from the incoming message buffers

Figure 3 shows the time required, categorized by the steps in the algorithm, to gather a total of 100,000 integers on all processors. As can be seen, most of the time is spent packing objects into and unpacking from the message buffer.

In the data transfer algorithm described above, step 2 is required because it is not always possible to determine in advance the amount of space needed to hold all the packed `Mobjects`. This may happen , for example, because different objects of the same data type contain different numbers of sub-objects, thus requiring different amounts of storage. If the size of each packed `Mobject` can be determined, and all `Mobjects` involved in the communication are known to have the same size, step 2 could be replaced by a local computation to avoid one round of all-to-all interprocessor communication.

Although the CHAOS++ data exchange routines can be made quite efficient, additional optimizations are still necessary. For complex data types, objects may become quite large, and simply

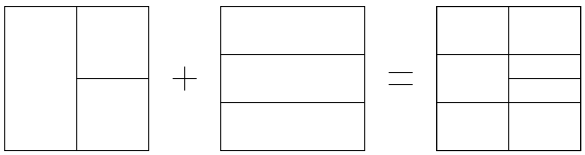

Figure 4: An example of polygon overlay.

polygon overlay problem, which is what was implemented, both input maps consist of only non-empty rectangles, with vertices on a rectangular integer grid.

Polygons of a map may be stored in a sorted order. This can be done by first sorting the polygons by their lower-X coordinates. Then each group of polygons with the same lower-X coordinates is sorted by their high-X coordinates. Polygons with the same lower-X and higher-X coordinates are further sorted by their lower-Y coordinates, breaking ties by comparing their upper-Y coordinates. Since the polygons are assumed to be non-overlapping, this guarantees a unique ordering of the polygons. Several of the variant implementations that will be described in this section rely upon this ordering. In our implementation, the algorithm sorts the output map if both input maps are sorted.

5.1.1 The Sequential Implementations

A naive sequential implementation of the polygon overlay algorithm consists of a nested loop that sweeps through two input maps, comparing each polygon in the first input map to each polygon in the other. Throughout this section, we will refer to the input map that corresponds to the outer loop in the sequential code as the *outer input map*, and the one that corresponds to the inner loop as the *inner input map*.

The naive overlay algorithm can be improved in several ways. The most obvious is to make use of the sorted order of the entries in each input map. If a polygon's high-X coordinate is x, then no polygon whose low-X coordinate is greater than or equal to x can possibly overlap it. This *sorted-order* optimization thus limits the sweep of the inner input map.

A second optimization is to keep track of how much of the area of each polygon has not yet been accounted for. A polygon with all its area "used up" no longer needs to be considered when calculating overlaps with other polygons. This *area-tracking* optimization uses an auxiliary integer vector to keep track of the unaccounted-for area of each inner polygon, and a scalar integer to do the same job for the current outer polygon. If the remaining area of a polygon from the inner map becomes zero, it is never examined in later iterations of the outer loop. When the area of the outer polygon is exhausted, that iteration of the outer loop is terminated.

A further refinement on the area-tracking optimization is to create a linked list of intermediate structures that "shadow" the inner input map. Each element of this list records the area of its corresponding inner polygon. Instead of looping through the whole inner map, the algorithm using this *list-deletion* optimization repeatedly traverses the list. Once an inner polygon's area has been exhausted, its shadow is deleted from the linked list so that later iterations of the outer loop need not consider it at all.

Finally, several combinations of these optimizations are possible. The first set of bars in Figure 5, marked "sequential", give the execution times for processing two sorted input maps of size 60,398 and 60,374, using the naive algorithm, the various optimizations, and the combinations of those optimizations. In Figure 5, execution times for using the naive overlay algorithm are labeled as "naive". Execution times for using only the sorted-order optimization, only the area-tracking optimization, and only the list-deletion optimization, are labeled as "ordered", "area" and "list", respectively. Execution times for using the combinations of these optimizations are labeled accordingly. The experiments show that the sorted-area optimization reduced processing time of the sequential execution by approximately half, which is what one would expect. The area-tracking optimization was much more effective, while the list-deletion optimization delivers the best performance. Surprisingly,

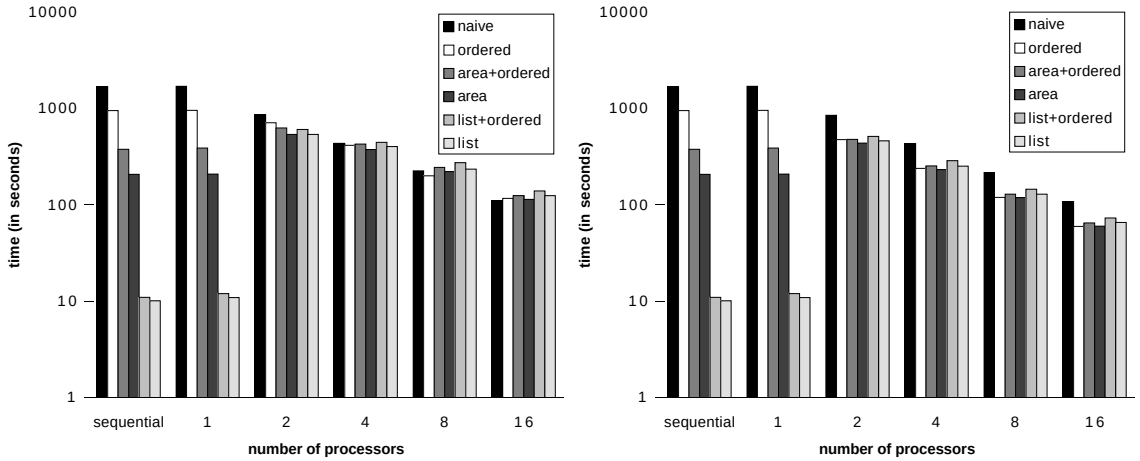

Figure 5: Performance results, in seconds, for the replicated parallel implementation. (a) uses a block distribution, and (b) uses a cyclic distribution.

combining the sorted-order optimization with either area-tracking or list-deletion produces a slower program. This may be attributable to the cost of an extra test in the control section of the inner loop.

5.1.2 The Parallel Implementations

Based on the sequential algorithm, we have implemented two parallel polygon overlay algorithms with the CHAOS++ library. Our basic strategy is to partition the outer input map across the processors so that iterations of the outer loop can be executed in parallel. The two implementations differ in the way the inner input map is handled. The experiments presented in this section use two input maps, both sorted, of size 60,398 and 60,374, respectively, with the output map containing 203,006 polygons. The reported running times do not include the time for either sorting the output or I/O.

The first algorithm is a straightforward parallel implementation with polygons from the inner input map fully replicated across all the processors. After reading all its assigned polygons into local memory, each processor concurrently executes the sequential overlay algorithm over its assigned polygons. Various optimizations that were shown to improve the performance of the sequential algorithm in Section 5.1.1 can also be directly applied with no modifications. Figure 5 gives the performance results for this replicated implementation with the outer input map partitioned with either a block or a cyclic distribution.

The naive algorithm, as expected, shows a linear speedup. The sorted-order optimization reduces the processing time relative to the naive algorithm by approximately half when using a cyclic distribution, but was far less effective for the block distribution, actually increasing the processing time on 16 processors. This is because the sorted-order optimization reduces the workload by telling every outer polygon how early it can terminate the inner loop. For polygons located near the beginning of the sorted outer input array, this is very effective, since very few polygons from the sorted inner input array need to be examined before the sorted-order optimization terminates the inner loop. However, for polygons located near the end of the sorted outer input array, the sorted-order optimization does not terminate the inner loop until most of the polygons from the sorted inner input array are examined. With a block distribution of the outer input map, this results in very imbalanced load on the processors, because the processor that is assigned the last part of the outer array cannot terminate the inner loop early to reduce its computation. That processor thus must perform more computation than the others, and becomes the bottleneck. In contrast, when a cyclic

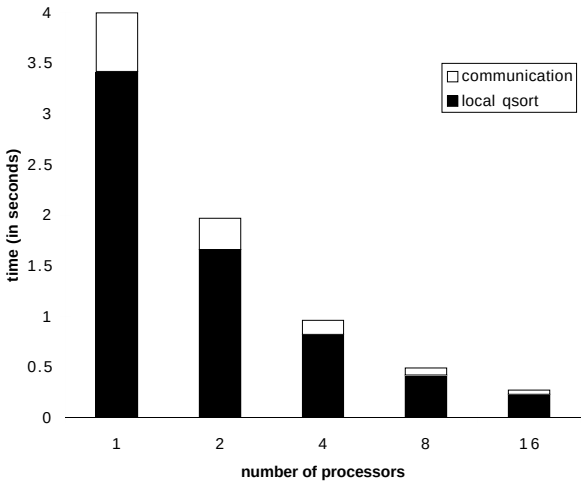

Figure 6: Time (in seconds) spent performing bin sort.

distribution is used the work load is distributed more evenly, providing better performance.

The area-tracking and list-deletion optimizations are less effective in the parallel than in the sequential implementation. In particular, the list-deletion optimizations make the parallel execution much slower than sequential execution with the same optimization. This is because the area-tracking and list-deletion optimizations reduce the computation by skipping all inner polygons that have been exhausted by the preceding outer polygons. With a block distribution of the outer map, each processor is assigned a set of outer polygons that spatially spans a very small area, and is therefore only able to eliminate inner polygons that are wholy contained within that small area. This implies that inner polygons can only be exhausted late in the computation, significantly limiting the effectiveness of the optimizations. With a cyclic distribution, each processor is not confined to polygons in a small area, and is therefore able to exhaust inner polygons much earlier in the computation, providing greater benefits. However, the outer polygons that each processor is assigned under a cyclic distribution do not span a contiguous area, significantly reducing the effectiveness of these two optimizations, so that they are much less beneficial than in the sequential implementation.

In the replicated algorithm, no interprocessor communication is required until the output polygons must be sorted across the processors. A simple bin sorting algorithm is used for that step in the application, with the total area covered by the input maps partitioned regularly along the x-axis among processors, and each of the output polygons moved from the processor where it was generated to the processor that contains the x-coordinate of its lower left corner. A local sort is then carried out on each processor over the assigned output polygons. In the parallel implementation, the CHAOS++ communication primitives are used to scatter output polygons among the processors. Figure 6 shows the time required to perform the bin sort. The white area in Figure 6 includes time spent for assigning the output polygons to processors, building a communication schedule, and performing the communication, taking about 14% of the total sorting time.

The second parallel algorithm forms the processors into a ring, and partitions both input maps with a block distribution. During program execution, polygons from the outer input map stay on the processors they were initially assigned to, while polygons from the inner map move between processors. The computation is thus broken down into alternating phases of computation and communication. In the computation phase, all processors concurrently invoke the sequential overlay algorithm, either the naive or improved versions, to process their local polygons from both maps. In the communication phase, a simple scheme is used, with each processor employing the CHAOS++ datamove routines to pack all the inner polygons it currently holds into a single message, then sending the polygons to the next processor in the ring. With NP processors, exactly NP computation phases and $NP - 1$ communication phases are required to ensure that the entire inner input map

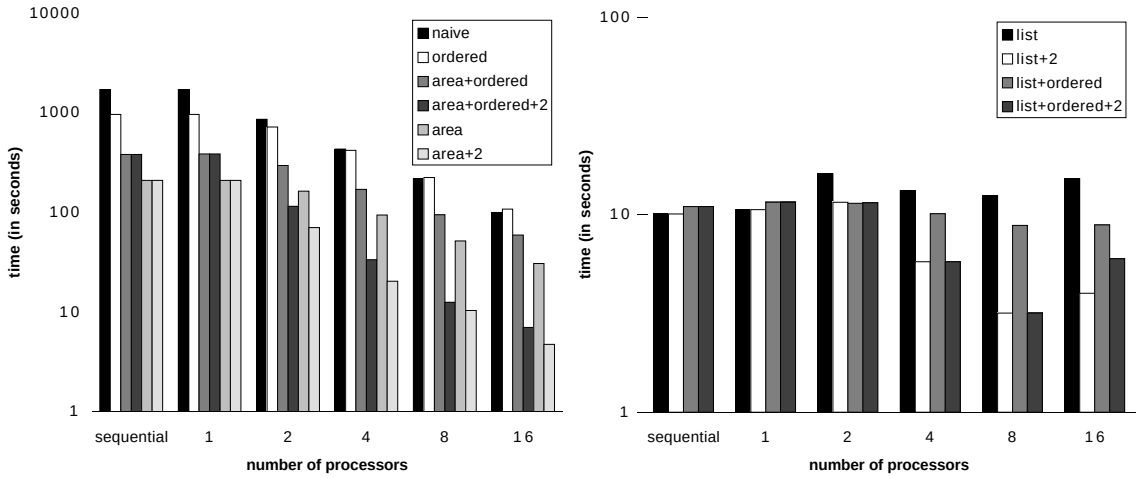

Figure 7: Performance results, in seconds, for ring parallel implementation, using block distributions and a simple communication scheme.

is examined by all processors. Since the communication pattern remains the same throughout the entire execution, a schedule is computed once and used for all the communication phases. If sorting of the output polygons is required, the same bin sort algorithm described earlier is used.

Figure 7 shows the performance of the parallel ring algorithm. The naive ring algorithm provides the same performance as the naive replicated algorithm, showing linear speedup, and the sorted-order optimization, which can be applied without any modification to the parallel ring algorithm, suffers from the same load imbalance problem as does the replicated algorithm. The area-tracking optimization is implemented by transmitting, at each communication phase, an auxiliary area vector, in addition to the inner input polygon vector. Area-tracking is more effective than in the replicated algorithm, since processors in the ring algorithm are able to skip inner polygons that have been exhausted by preceding processors in the ring, effectively reducing the computational load, which did not happen with the replicated algorithm.

The list-deletion optimization, as can be seen from the sequential performance, provides the greatest performance improvement from the naive parallel executions, when applied to the ring algorithm. In the parallel implementation, the linked list that shadows the inner input map is declared as a CHAOS++ `Mobject`, and by using a customized `pack` function the runtime system is able to traverse the list during each communication phase and transmit only the inner polygons that have not yet been exhausted. This not only reduces the volume of communication, but also greatly reduces the processing time in later computation phases, resulting in improved performance. The effectiveness of the list-deletion optimization is further amplified by the fact that the two input maps were partitioned using the same distribution function. Partitioning both maps with a block distribution assigns submaps that are almost spatially aligned with each other to the same processor, which enables the parallel program to eliminate most of the inner polygons during the first computation phase. Table 1 shows, in the first column, the total number of inner polygons sent over all the communication phases, using the simple communication scheme and the list-deletion optimization. The second column shows the number of polygons sent during the very first communication phase, for the simple communication scheme. It can be seen that only a few inner polygons survive after the first computation phase and need to be transferred through the processor ring for further processing. For comparison, the naive algorithm (with no optimizations) transfers the whole inner map (60,374 polygons) during *every* communication phase. The third column in Table 1 gives the total number of polygons that need to be communicated by the ring algorithm, when using a different communication scheme that will be described shortly.

Since every processor in the ring parallel algorithm performs multiple scans over its assigned outer

number of processors	simple communication scheme	first communication phase	ping-pong communication scheme
2	393	393	393
4	2394	1026	1710
8	11690	2492	4025
16	50533	5244	8479

Table 1: Number of polygons sent by the ring parallel implementation with the list-deletion optimization, under two different communication schemes. The inner map contains 60,374 polygons.

input submap, it should be useful to also apply the area-tracking and list-deletion optimizations to the outer input map. Applying these optimizations results in a significant performance improvement, as shown by the bars marked with "+2" in Figure 7.

However, Figure 7 also reveals that, with the list-deletion optimization applied to the inner input map, the parallel execution never outperforms the sequential implementation. Even with the list-deletion optimization applied to both input maps, there is a slowdown when increasing the number of processors from 8 to 16. This is because the simple rotation communication scheme does not take full advantage of the spatial locality between the two input maps. As was described earlier, the two input maps were partitioned with the same block distribution, and with that distribution every outer polygon only overlaps with inner polygons on the same processor, and possibly ones on the neighboring processors in the ring. Therefore, the ring algorithm starts with a partitioning scheme that promotes polygon exhaustion during the very first computation phase. However, the simple communication scheme consistently rotates the inner input map in one direction along the processor ring, gradually increasing the spatial mismatch between the two input maps, thereby preventing early exhaustion of inner map polygons for later computation phases. This can easily be solved by adopting a *ping-pong* communication scheme. If we view the processors as forming a ring, and use mod NP arithmetic, an inner input submap that is initially assigned to processor i under the ping-pong communication scheme would visit processor $i+1$, processor $i-1$, processor $i+2$, processor $i-2$, processor $i+3$, etc., as illustrated by Figure 8. The last column in Table 1 shows the total number of polygons transferred over all the communication phases with the ping-pong communication scheme. It can be seen that, compared to the simple communication scheme shown in the first column, many fewer polygons were transferred between processors with the ping-pong communication scheme. Note that the same number of polygons are communicated during the first communication phase under both communication schemes (the second column in the table), since both schemes exhibit the same communication pattern in the first communication phase. The only required modification to the parallel program is to generate a new communication schedule for every communication phase. Figure 9 shows the performance of the parallel implementation using the ping-pong communication scheme. We see that the new algorithm scales quite well up to 16 processors.

5.2 Scientific Computation - Direct Simulation Monte Carlo Method

The Direct Simulation Monte Carlo (DSMC) method 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 3-dimensional Cartesian mesh [33]. Depending upon its current spatial location, each particle is associated with a mesh cell, which typically contains a few particles, and moves from cell to cell as it participates in collisions and various boundary interactions in the simulated physical space. What distinguishes the DSMC method from other Particle-in-Cell (PIC) methods is that the movement and collision processes of particles are

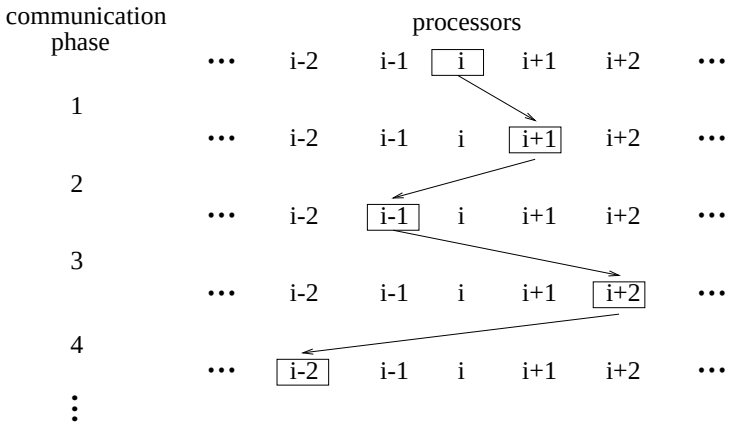

Figure 8: Communication pattern of an inner input submap initially assigned to processor i, using a ping-pong communication scheme.

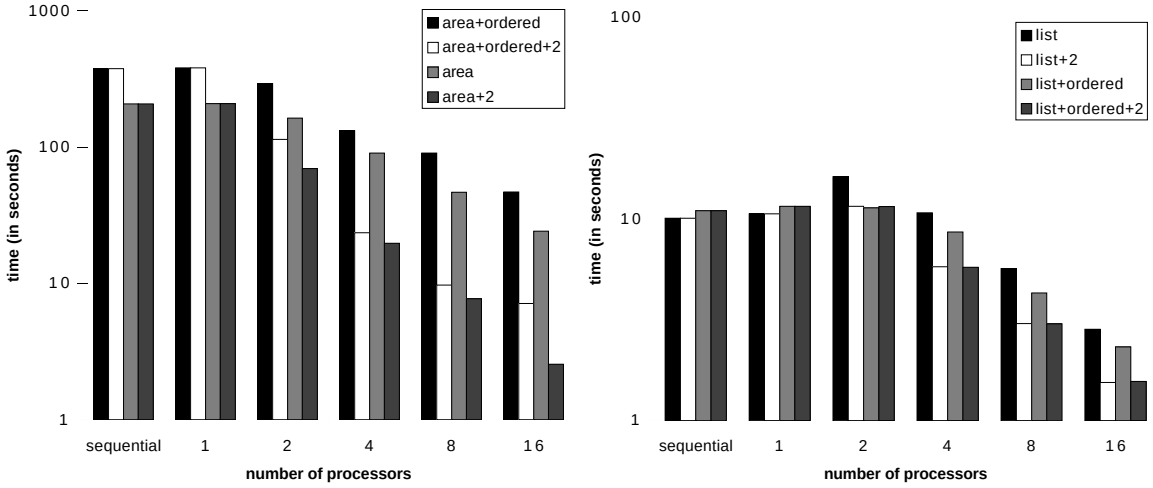

Figure 9: Performance results, in seconds, for ring parallel implementation, using block distribution and a ping-pong communication scheme.

completely uncoupled over a time step [34].

In the parallel implementation, mesh cells are distributed *irregularly* among the processors to achieve a good load balance. A CHAOS++ translation table with an entry for each mesh cell is constructed to describe the distribution. Particles, which only interact with other particles in the same mesh cell, are assigned to the owner processors of the mesh cells that the particles reside in. Since particles move between mesh cells, the cells, and thus the particles, are redistributed across the processors occasionally (once every few time steps) to maintain a good load balance.

Moon [28] 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, such as velocity components, rotational energy and position coordinates, are stored in separate data arrays, and the association between the Cartesian mesh cells and the particles is represented by indirection arrays. As mesh cells are redistributed, these data arrays are also redistributed accordingly using the CHAOS data transportation primitives. Since the number of mesh cells is relatively small, the translation table for mesh cells is replicated across all the processors so that dereference requests can be resolved locally without any communication. The replicated translation table is modified only when the mesh cells are redistributed.

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 data member in the Cell class. That is, the Cell class serves as a container class, and the Particle objects are sub-objects of the Cell class. Both the Cell and Particle classes are derived from the CHAOS++ `Mobject` base class, and when redistributing the cells, the `pack` and `unpack` functions of the Cell class that support a deep copy are used to move all the particles associated with a cell between processors, along with other data for the cell.

Figure 10 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 48,600 particles. 400 time steps are performed, and a chain partitioner [28] is used to dynamically partition the mesh cells at runtime when load imbalance is detected. The Fortran code has been shown to be a good implementation, and the C++ version is at most 15% slower. As the number of processors increases, communication cost becomes a significant factor and accounts for the non-linear speedup of the Fortran code. In the C++ version, invoking the virtual functions `pack` and `unpack` when moving particles and cells is somewhat expensive, and is the main reason why the C++ version performs worse than the Fortran code when very few processors are used. With a small number of processors, each processor is assigned a large number of particles and cells and therefore invokes the virtual functions `pack` and `unpack` many times. As more processors are employed, fewer cells and particles are assigned to each processor, which requires fewer invocations of these virtual functions. With these smaller overheads, the performance of the C++ code comes closer to that of the Fortran version.

5.3 Spatial Database Systems - Vegetation Index Measurement

Vegetation Index Measurement (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 [31]. The overall project involves developing scalable and portable parallel programs for a variety of image and map data processing applications, to be integrated with new methods 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 over the past three decades. The VIM application is an example of such analysis, and exhibits a data access pattern that is often observed in these types of applications.

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

Figure 10: Performance for DSMC application on Intel iPSC/860.

over a period of time, or images from multiple sensors taken at the same time. The query region is overlaid with a 2-dimensional mesh, whose resolution is 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 [40] (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 column data vectors, with each column represented by a class derived from `Mobject`. This was done primarily for efficient implementation of the spatial operators provided by the existing class library. For every specified sensor image, the algorithm first computes a sub-image that spatially contains the whole query region, and regularly partitions the sub-image across the processors by blocks of columns of data vectors to obtain good load balance.

Jovian I/O library [3] routines are then invoked to read the computed sub-image from the disks and distribute the data vectors as specified. The Jovian library has been designed to optimize the I/O performance of multiprocessor architectures with multiple disks or disk arrays. On each processor, the contribution of the assigned portion of the sub-image to the mesh cells of the query window is then computed, and the CHAOS++ library is used to generate the necessary communication to combine the results across all processors. The communication phase is necessary, since each sub-image is distributed to obtain good load balance, and consequently, computing the vegetation index of a single mesh cell may require satellite image data points assigned to different processors. Furthermore, the mesh cells may not be spatially aligned with the input satellite data, and therefore may be distributed differently from the satellite data. The communication phase is needed to align the computed results with the distribution of the mesh cells.

The results of running VIM on both the Intel iPSC/860 and the IBM SP-1 over two different sets of data are illustrated in Figure 11. In the first experiment, three satellite images, each of which consists of 750×100 data points, are used to compute the vegetation index for a query region consisting of 150×20 mesh points. In the second experiment, three satellite images, each of which consists of 1200×150 data points, are used to compute the vegetation index for a query region consisting of 240×30 mesh points. For comparison, the execution times for the sequential code are

(a) (b)

Figure 11: Execution time for the Vegetation Index Measurement (VIM) on the (a) Intel iPSC/860 and the (b) IBM SP-1.

also given when available. Due to insufficient memory, we could not conduct the second experiment on a single node of the iPSC/860. Since we are mainly concerned with the computation time for characterizing the behavior of the CHAOS++ library, the query regions in both experiments wholly contain all the input satellite images, and the I/O time is not included in these measurements. A detailed description of the I/O performance of the VIM application using the Jovian library is given in Bennett *et al.* [4]. The results from Figure 11 show that good speedup is obtained.

5.4 Image Processing - Image Segmentation

Another application under development is image segmentation. This application segments a given image into a hierarchy of components based on the border contrast between adjacent components. The segmentation serves as a preprocessing phase for an appearance-based object recognition system developed at the University of Maryland [35]. The hierarchy is 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 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 contrast criterion between adjacent pixels given by the user. The algorithm then collapses adjacent components into larger components, based on a series of weaker and weaker criteria on the border contrast between adjacent components. The history of component collapses is kept as a hierarchy of image segmentations for later use.

In the parallel implementation, the initial image is represented as an array regularly distributed by blocks of columns of pixels across the processors. On each processor, CHAOS++ routines are used to fetch non-local neighboring pixels into a local buffer, so that the computation for clustering pixels into components can be carried out locally. A component is then created for each identified cluster of pixels. Once formed, the components are stored in a local undirected weighted graph on each processor, 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, which are the graph nodes in the undirected graph.

The component class keeps a list of border objects, each of which contains, along with other related information, a pointer to one of the adjacent components. These are the weighted edges in

(a) (b)

Figure 12: Performance results for the image segmentation application on the (a) Intel iPSC/860 and the (b) IBM SP-1.

the undirected component graph. After the local component graphs on all the processors are built, they are used to form a distributed graph. A simple scheme is used to merge and assign overlapping components to processors to achieve better load balance. and CHAOS++ routines are invoked to compose local component graphs into a global distributed graph, as described in Section 3.3.2. The CHAOS++ mapping structure is then constructed on each processor for the distributed graph, and used to generate a communication schedule. The communication schedule is used for efficient exchange of component data between real `Gobjects` and ghost `Gobjects` when performing graph contraction. As the graph changes during program execution, CHAOS++ primitives are invoked repeatedly to modify the mapping structure, and generate new communication schedules.

Figure 12 shows the performance results for segmenting two different images on both the Intel iPSC/860 and the IBM SP-1. The parallel implementation is based on sequential code that has not been highly optimized, so only the execution times for the parallel code are shown. The image segmentation problem has been chosen to challenge the capabilities of the CHAOS++ library, so the performance of the sequential algorithm is not crucial. In the first experiment, the pixels of a 400×400 image are initially clustered into 7,935 components, connected by 17,816 (undirected) edges, and the components are used to build a segmentation hierarchy of height five. In the second experiment, a 512×513 image initially generates 22,100 components, connected by 49,549 (undirected) edges, and the components are used to build a segmentation hierarchy of height nine. Due to insufficient memory, there is no result for conducting the second experiment on a single node of the iPSC/860.

In the current implementation, newly formed components are allocated dynamically and inserted into the component graph. On the iPSC/860, however, allocating memory dynamically turns out to be a very expensive operation, and is responsible for almost 80% of the total running time of the application. That is the main reason why the SP-1 greatly outperforms the iPSC/860 in both experiments, especially when the number of components generated from the image is large. The cost of dynamic memory allocation on the iPSC/860 is also non-linear, meaning that as more components are allocated, the longer it takes on average to allocate one component. Since using more processors causes each processor to allocate fewer nodes in the component graph, the super-linear speed-ups shown in Figure 12 for the iPSC/860 are not surprising.

On the SP-1, since the datasets are relatively small, communication cost becomes a factor as the number of processors increases and the amount of work assigned to each processor decreases, and accounts for the non-linear speedup. One other significant feature of the current implementation of the segmentation algorithm is that no remapping of the distributed component graph is performed,

once it is formed. Currently the input image is initially partitioned, and that partition decides how the components are distributed among processors. This results in a rather unbalanced work load among the processors, and also contributes to the non-linear speedups as the number of processors increases.

6 Related Work

In this section, we briefly discuss other systems from the area of concurrent object-oriented programming, as well as related work on support for distributed pointer-based data structures. Roughly speaking, there are two types of systems that are relevant.

The first type of system augments an existing language with parallel constructs. Parallelism is exploited by both a compiler and an associated runtime system. Examples of this type of system, based on C++, include Mentat [17], CC++ [7], ICC++ [10], C** [24], CHARM [21], and pC++ [26]. Mentat, CC++ and CHARM consider program execution as completely unstructured interactions among a set of objects and support only asynchronous communication, and this approach suffers from overhead incurred from either polling or interrupts. ICC++, pC++ and C** provide array-based collections of objects, and parallel constructs to operate on them. However, the runtime system of C** assumes a shared-memory or distributed shared-memory architecture, and data fetch is performed by page faulting through the underlying memory consistency protocol. ICC++ and pC++ are able to run on distributed memory architectures, but still rely solely on asynchronous communication. CHAOS++, in contrast, is a user-level class library, and does not assume any language or compiler support. After analyzing and optimizing communication patterns, CHAOS++ generates communication schedules to enable collective communication, which ensures that neither polling nor interrupts are needed at the receiving processors. Communication can thus be performed efficiently for loosely synchronous applications.

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++ [30] and LPARX [23]. These libraries both provide efficient management of array-based parallel constructs distributed across processors. CHAOS++, however, performs optimization through preprocessing techniques, and provides efficient support for dynamic distributed data structures, including pointer-based data structures.

Some recent work focuses on support for distributed pointer-based data structures. Gupta [18] suggests an approach that assigns to every element of a distributed pointer-based data structure a name known to all processors. The name is based on the position of the element in the data structure, which is registered with all processors as the element is inserted into the data structure. This naming convention seriously restricts the way the data structure can be used. For example, new elements can only be added to the data structure one at a time, not in parallel. This is a serious restriction for efficient parallel implementations, because it could require many synchronization operations, which are usually expensive on distributed memory machines.

Carroll *et al.* [6] propose a composite tree model, which describes a data-parallel computation as a tree structure. The computation starts at the root of the structure, which, usually in parallel, invokes methods from its child nodes. The child nodes, in turn, invoke methods from their child nodes, and so on. Parent nodes then wait for their children to complete, and combine the results returned by their children nodes to form their results. However, it is not clear how accesses through pointers of an element of a distributed pointer-based data structure can be implemented under the composite tree model, other than with fine-grain blocking sends and receives.

Olden [36] is a C-based system that supports recursively defined pointer-based data structures. In Olden, an application allocates nodes of its pointer-based data structures on the heaps of different processors, and the runtime system migrates threads of computation to processors that own the heap-allocated nodes being processed. Parallelism is exploited through concurrent execution of migrating threads and their continuations. A major difference between Olden and CHAOS++ is the way work is allocated. In Olden, work follows the data, while in CHAOS++, the required data is copied so that computation can be carried out locally.

7 Critique and Conclusions

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, 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, other than an underlying message passing system. Integration with the CHAOS runtime library, for array-based adaptive irregular applications has already been accomplished, and integration with the Multiblock PARTI runtime library, for array-based structured grid applications, is currently in progress.

We are currently working to create a more robust version of the CHAOS++ runtime library, so that it can be more effectively used by application developers. One of the major difficulties in using the current version of the library is the complexity of the user interface. A user is asked to derive classes from the `Mobject` base class, and provide implementations for the pack and unpack functions to support deep copies. Some of this could be automated by a compiler, perhaps with the help of annotations provided by the user. On the other hand, building a distributed graph requires some understanding of the way the runtime library works, and extra work from the user (for example, laying out the `Gobjects` on the boundaries of the subgraph owned by a processor in a consistent order, as described in Section 3.3.2). At this point in time, we have yet to find a more general interface for building distributed graphs. Furthermore, CHAOS++ relies heavily on C++ virtual function invocations, which are usually somewhat more expensive than normal function calls. Compiler analysis and optimization that reduces the cost of virtual function invocations could significantly improve the performance of the CHAOS++ runtime library.

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 also in the process of integrating CHAOS++ into the runtime software being developed by the Parallel Compiler Runtime Consortium. The goal of this consortium is to provide common runtime support for compilers of data parallel languages, through specification of interfaces for data structures and for routines for deriving and optimizing data movement among processors. Runtime support, such as that provided by CHAOS++, could then be used by any compiler that understands these interfaces, allowing the use of multiple runtime support packages (e.g. for coping with different array distributions) by a single compiler.

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 Samuel 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 and Argonne National Laboratory for providing access to their SP-1s.

References

[1] G. Agrawal, A. Sussman, and J. Saltz. Efficient Runtime Support for Parallelizing Block Structured Applications. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 158–67. IEEE Computer Society Press, May 1994.

[2] G. Agrawal, A. Sussman, and J. Saltz. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747–54, July 1995.

[3] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[4] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Collective I/O: Models and Implementation. Technical Report CS-TR-3429 and UMIACS-TR-95-29, University of Maryland, Department of Computer Science and University of Maryland Institute for Advanced Computer Studies, February 1995.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation and Performance Results. In *Proceedings Supercomputing '93*, pages 351–60. IEEE Computer Society Press, November 1993.

[6] M.C. Carroll and L. Pollock. Composites: Trees for Data Parallel Programming. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 43–54. IEEE Computer Society Press, May 1994.

[7] 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.

[8] C. Chang, A. Sussman, and J. Saltz. Object-oriented runtime support for complex distributed data structures. Technical Report CS-TR-3438 and UMIACS-TR-95-35, University of Maryland, Department of Computer Science and University of Maryland Institute for Advanced Computer Studies, March 1995.

[9] C. Chang, A. Sussman, and J. Saltz. Chaos++: A runtime library for supporting distributed dynamic data structures. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[10] A.A. Chien and Julian Dolby. The Illinois Concert System: A Problem-solving Environment for Irregular Applications. In *Proceedings of DAGS'94, The Symposium on Parallel Computation and Problem Solving Environments*, 1994.

[11] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing '93*, 1993.

[12] 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):489–96, March 1994.

[13] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pages 185–220. Elsevier, 1992.

[14] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–79, September 1994.

[15] G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works*. Morgan Kaufman, 1994.

[16] M. Gerndt. Updating Distributed Variables in Local Computations. *Concurrency: Practice and Experience*, 2(3):171–93, September 1990.

[17] A.S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[18] R. Gupta. SPMD Execution of Programs with Pointer-Based Data Structures on Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, 16:92–107, 1992.

[19] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–60, July 1991.

[20] Y.-S. Hwang, R. Das, J.H. Saltz, M. Hodoscek, and B.R. Brooks. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.

[21] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Andreas Paepcke, editor, *Proceedings of OOPSLA'93*, volume 28, pages 91–108. ACM Press, 1993.

[22] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[23] 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.

[24] J.R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–41. Springer-Verlag, August 1993.

[25] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[26] 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.

[27] R. Mirchandaney, J.H. Saltz, R.M. Smith, K. Crowley, and D.M. Nicol. Principles of Runtime Support for Parallel Processors. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 140–52. ACM Press, July 1988.

[28] 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 176–83. IEEE Computer Society Press, May 1994.

[29] W. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[30] R. Parsons and D. Quinlan. Run-time Recognition of Task Parallelism Within the P++ Class Library. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 77–86. IEEE Computer Society Press, October 1993.

[31] R. Parulekar, L. Davis, R. Chellappa, J. Saltz, A. Sussman, and J. Towhshend. High Performance Computing for Land Cover Dynamics. In *Proceedings of the International Joint Conference on Pattern Recognition*, September 1994.

[32] R. Ponnusamy, Y.-S. Hwang, R. Das, J.H. Saltz, A. Choudhary, and G. Fox. Supporting Irregular Distributions using Data-Parallel Languages. *IEEE Parallel & Distributed Technology*, 3(1):12–24, Spring 1995.

[33] 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. American Institute of Aeronautics and Astronautics.

[34] P.J. Roache. *Computational Fluid Dynamics*, volume 1. Hermosa Publishers, Albuquerque, New Mexico, 1972.

[35] C. Rodríguez. An Appearance-Based Approach to Object Recognition in Aerial Images. Master's thesis, Computer Science Department, University of Maryland, College Park, 1994.

[36] A. Rogers, M.C. Carlisle, J.H. Reppy, and L.J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–63, March 1995.

[37] J. Saltz, R. Ponnusamy, S.D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das. A Manual for the CHAOS Runtime Library. Technical Report CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and University of Maryland Institute for Advanced Computer Studies, March 1995.

[38] 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.

[39] S.D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proceedings of Supercomputing '94*, pages 97–106, November 1994.

[40] C.T. Shock, C. Chang, L. Davis, S. Goward, J. Saltz, and A. Sussman. A high performance image database system for remote sensing. In *24th AIPR Workshop on Tools and Techniques for Modeling and Simulation*, Washington, D.C., October 1995. Society for Photogrammetric and Industrial Engineers. To appear.

[41] 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 214–25. IEEE Computer Society Press, November 1993.

[42] A. Sussman, G. Agrawal, and J. 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 University of Maryland Institute for Advanced Computer Studies, December 1993.