Object Template Abstractions for Light-Weight Data-Parallelism [*]

Neelakantan Sundaresan Dennis Gannon

nsundare@cs.indiana.edu gannon@cs.indiana.edu

Computer Science Department

215 Lindley Hall

Indiana University

Bloomington, IN 47405

Abstract

Data-parallelism is a widely used model for parallel programming. Control structures like parallel DO loops, and data structures like collections have been used to express data-parallelism. In typical implementations, these constructs are 'flat' in the sense that only one data-parallel operation is active at any time. To model applications that can exploit overlap of synchronization and computation in data-parallel tasks, or that have independent but limited data-parallelism, or that depict static or hierarchical nested parallelism, a more dynamic model is required. This paper describes how to combine light-weight thread mechanism with object-oriented methodologies to provide light-weight

[*]This research was funded in part by: ARPA DABT63-94-C-0029 and Rome Labs Contract AF 30602-92-C-0135

and dynamic data-parallel constructs. We discuss the necessary abstractions to build objects with data-parallel semantics in the context of the *Coir* system [16, 17], a C++-based system for control and data-parallelism. We also study the three classes of applications: pure data-parallel, static nested data-parallel, and hierarchical algorithms.

1 The *Coir* System

The *Coir* system provides a model for both control and data parallelism. It is implemented as a C++ library. The system is designed to use reusable components like pthread-standard thread libraries [10] and MPI message passing libraries [6]. The architecture model subsumes both shared and distributed memory machines. The following paragraphs provide a brief description. For details on the design and implementation of *Coir* refer to [15, 16, 17].

1.1 Control Parallelism

In *Coir*, control parallelism is modeled in terms of light-weight user-level threads. A thread is basically a sequence of instructions in execution in a program and has its own stack and program counter. Multiple threads can execute on a process, and threads can migrate across processes sharing the same memory arena. Described in C++, the control mechanism is combined with inheritance, dynamic dispatch, and parameterized types to support control objects that are specific to user-applications or to a target model of parallelism. In addition, thread objects can also synchronize over shared memory using light-weight mutexes and condition variables. Thread objects within or across address domains may communicate using thread-to-thread synchronous or asynchronous communication. This communication mechanism can be used to support actor-style or message-driven semantics. The communication mechanism has a one-to-one mapping to the MPI communicator mechanism, so if the MPI implementation avoids buffer copies, no additional copies are introduced.

1.2 Machine Model

The *Coir* system models a hierarchy of shared and distributed memory machines. At the lowest level is a symmetric shared memory component. At the next level is the hierarchical shared memory component. At the highest level is a machine with distributed address spaces. The machine description is abstracted as C++ objects. The machine over which the user application runs is called a *domain*. The largest shared-memory component is called a *subdomain*. The largest component over which a thread object can migrate is called a *processor-context*. A *processor-context* represents the space for thread locality and load-balancing. Coarser level load-balancing is achieved by varying the distribution of threads. Active data-parallel objects are distributed over subsets of *domain* objects.

This paper concentrates on abstractions for data-parallelism and the following sections will discuss the same in detail.

2 Dynamic Data-Parallel Objects

In the *Coir* system, we define a *rope* as a group of threads distributed over a subset of a *domain*, where the threads in the rope are grouped for the purpose of collective data-parallel computation and communication. A rope provides a uniform naming mechanism for data-parallel constructs. *Ropes* are C++ objects encapsulating multiple controls to define data-parallel semantics. Data-parallelism in *Coir* is light-weight in the sense that data-parallel tasks are assigned and executed at the level of light-weight threads. Barrier synchronization, reduction, and broadcast operations are also defined to be rope-specific.

2.1 Rope Objects in C++

Rope objects are implemented as constrained container (template) classes [14]. The template parameters are constrained to be instantiated by arguments that are derived thread classes. The 'pseudo'-C++ syntax for rope templates is as given below:

```
template <class T : Thread> class RopeTemplate {
        ...
};
```

Figure 1 shows the class structure for the Rope base class from which the above RopeTemplate container class is inherited.

The constructor takes 4 parameters, the last three being optional. The first parameter is the size of (number of threads) in the rope. The next parameter is the object representing the distribution scheme of the rope. The third parameter is the domain subset over which the rope is distributed, and the last parameter is the domain subset from which the rope constructor is invoked. The third and the fourth parameters are defaulted to the domain of all processors. The figure also shows the important member functions. The data-parallel function invocation methods (*Execute*), the function for returning values from such a parallel invocation (*Return*), and function for waiting on a data-parallel function to terminate (*Wait*) are some of the important ones. There are methods for synchronization. These are invoked by each of the participating thread in a data-parallel function. These functions are discussed in detail in a later section.

```
class RopeBase {
  public:
     DistRopeTemplate(const int count,  // number of threads
                      DistScheme& dist_scheme,  // scheme for distribution
                      Domain& dist_domain    // domain of distribution
                              = Scheduler::DefaultDomain(),
                      Domain& current_domain // domain of creation
                              = Scheduler::DefaultDomain());
     int Size() const; // Number of threads in the rope

     // return the object that shows how the threads are distributed
     DistributionObj DistObj() const;

     // domain over which the threads are distributed
     Domain DistDomain() const;

     // parallel function invocation
     TaskId Execute(ParFuncType func, ArgType arg, // func + arg
                    const Domain& invocation_domain
                       = Scheduler::DefaultDomain()); // invocation domain
     TaskId Execute(ParFuncType func, ArgType arg,  // func + arg
                    PackableObj& ret_ref,  // object for returning values
                    const Domain& invocation_domain
                       = Scheduler::DefaultDomain()); // invocation domain
     void Wait(const TaskId task_id);

     // returning a value

     // this function has to be performed from the parallel function
     // environment and executed by a member thread
     static void Return(PackableObj& obj_in,
                        BufferToObjCopierType obj_copier);

     ThreadId operator[](const int thread_index) const; // indexing
     RopeId   GetRopeId() const;  // get the corresponding rope identifier

     // intra−rope operations : synchronization
     static void Barrier();
     void Reduce(ReducerBase& rb);
     void Broadcast(BroadcasterBase& bb);
     ...
  };
```

Figure 1: A Rope Template Class Declaration and a Simple Instantiation

Application-specific rope objects can be created by inheriting from RopeTemplate container classes. There are two kinds of inheritance mechanisms used for these objects (see figure 2). Since a rope is defined as a container class constrained by the instantiation of the parameters to thread class types or their derived types, instantiating this parameter with a derived thread class results in *parallel inheritance.* Since rope classes are normal C++ classes, derived rope classes can be defined using normal C++ *serial inheritance.* Typical applications use a combination of serial and parallel inheritance.

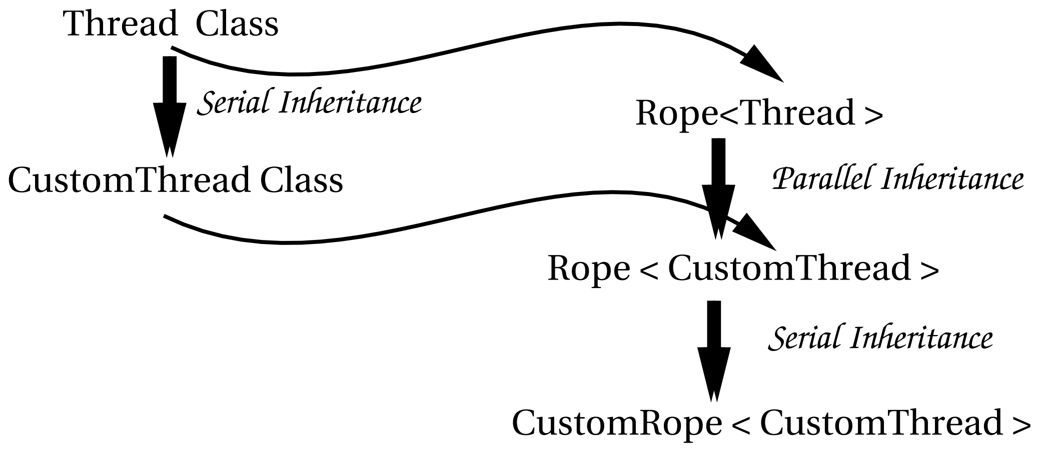

Figure 2: **Rope Inheritance:** Simple Rope class is created from aggregation of Thread class; Serially derived Rope classes are derivations of simple Rope class; Parallelly derived Rope classes are aggregations of derivations of Thread class; Combining derivations of Rope class and aggregation of derived Thread classes, serial-parallel inheritance is achieved.

2.2 Rope Distribution

Distribution of threads in a rope is specified through objects representing the distribution scheme. Distribution schemes are similar to data-distribution in HPF [9] or pC++ [3]. The difference is that the distribution schemes depict both data and control distribution. The distribution scheme objects are passed as additional arguments to the Rope class constructor. The system defines BLOCK and

CYCLIC distribution scheme objects. User-specified distribution scheme classes can be passed. These objects should follow the discipline of describing the distribution scheme by deriving from a base distribution scheme class and instantiating the appropriate virtual functions. The user may choose a distribution scheme that is appropriate to the data-parallel methods to be scheduled on the rope. The threads in a rope may be distributed over any topology of a subset of the set of processors in the machines. The distribution scheme just provides the implementation of an absolute and relative indexing mechanism.

2.3 Scheduling Data-parallel Methods on Ropes

Methods can be scheduled on a rope object by invoking the *Execute* member function. This function expects two parameters, a function and an argument. The function is executed by each thread in the rope. The invocation of this member function schedules a data-parallel task on the rope and returns a data-parallel task-id, which can be waited upon. Thus data-parallel task scheduling is asynchronous.

2.4 The Overlapping Domain Model

Languages like HPF and pC++ have a flat SPMD programming model, ie., the program starts up in a parallel mode and all processors continue to participate in parallel operations. Sequential portions of work is duplicated across all the processors, while for the parallel portion, each processor works on its share. In the C* language, from any parallel context defined by the set of virtual processors participating in the parallel operation, only a subset of these virtual processors participate in a new parallel operation invoked from this context. This can result in 'sparse' contexts [11].

Coir defines the notion of *caller domain* and *callee domain.* The caller domain represents the domain subset containing the *processor-contexts* that participate in a rope creation or a rope-specific data-parallel method invocation. The callee domain represents the domain subset containing the *processor-contexts* over which the threads in the rope are distributed (and hence *processor-contexts* over which they operate). By keeping track of the caller and the callee domains different programming models can be supported. For instance, in a flat SPMD model, the caller and the callee domains are the same. In a master-worker model, the caller and the callee may be disjoint, with the caller being a domain subset with a single *processor-context*. Our general model, termed the *overlapping domain model*, provides a flexible mechanism to create rope objects, schedule tasks on them, and wait on those tasks. Figure 3 shows this dynamic nature of this model. The figure shows four arbitrary domain subsets, $D1, D2, D3$, and $D4$. A rope r is created from $D1$. The threads in the rope are distributed over $D2$. If '$*$' and '$-$' stand for intersection, and difference operation respectively, in the Domain class, *processor-contexts* in $D1 * D2$ obtain a rope object with threads, while *processor-contexts* in $D1 - D2$ obtain a proxy rope object because there are no local threads that are a part of the rope. If r is visible to $D3$, then a data-parallel task can be scheduled from $D3$ on the rope. The task can be waited upon from $D4$ that has a handle on both r and the id of the scheduled task.

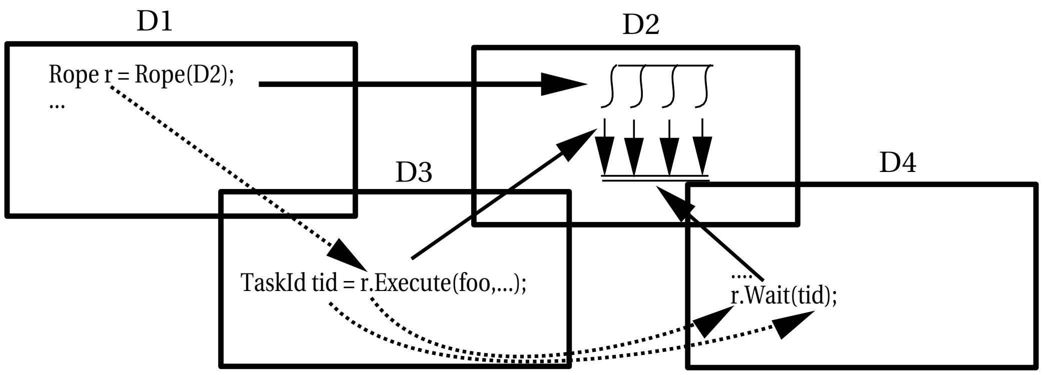

Figure 3: Dynamic Nature of Rope Creation, Data-Parallel Task Scheduling, and Waiting on that task.

2.5 Synchronization Mechanism

Data-parallel synchronization operations are defined at rope-level. Threads participating in a data-parallel task may enter a synchronization phase where they handshake with each other as a measure of check-pointing. This may be a barrier synchronization, a reduction operation, or a broadcast operation. In the traditional process(or)-level synchronization algorithms, when a process enter a barrier phase and waits for other processes to enter the phase, the application process is blocked and no useful computation at the application level can take place. Depending on the implementation of the algorithm, either the process might just busy-wait till all other processes arrive or it may block wherein the operating system might do a 'heavy-weight' context-switch. In either case, no useful computation at the application level can take place. In a multithreaded synchronization, as discussed in our system, when a thread in a rope arrives at the barrier and blocks or yields waiting for the other threads to arrive at the barrier, one of the other threads, if any, gets scheduled. Thus computation related to the newly scheduled thread can proceed. This, combined with the asynchronous data-parallel method scheduling mechanism, provides a way for usefully scheduling multiple data-parallel operations that can overlap computation with synchronization.

Table 1 shows the results of a synthetic application involving only barrier synchronizations. Here one to eight ropes, each of size 64, are distributed over a 64-node partition on the Intel/Paragon. The threads in each rope enter a rope-specific barrier synchronization 10,000 times. It can be seen that time taken for barrier synchronization for two ropes is less than twice the time taken for one rope. This is because, when there is only one rope a thread that arrives at the barrier and waits for other threads in its rope to arrive yields in a loop and since there is no other work to be done, the thread is rescheduled. When there are two ropes, the barrier synchronization operations between the two ropes are intertwined and the number of unnecessary yields are reduced.

Number of ropes active	1	2	4	8
Time taken in secs	37.7	51.9	79.8	150.8

Table 1: Timings in seconds on the Intel/Paragon for 10,000 barriers for ropes of size 64 on a 64-node partition

2.6 Barrier, Reduction, and Broadcast Synchronization

All the three operations - barrier, reduction, and broadcast - involve an arrival phase at the synchronization point, followed by a departure phase. The algorithm used avoids erroneous interference between different synchronization operations related to the same rope. For instance, if a tree algorithm is used to implement synchronization, the synchronization involves a fan-in followed by a fan-out. The three synchronization operations have the same computational structure (fan-in followed by fan-out) but actually perform different computations. The barrier synchronization does not involve any computation, while in the broadcast synchronization, one of the threads that broadcast the value is identified as the root of the broadcast tree, and after the fan-in is achieved, the value is broadcast during the fan-out phase. In the case of reduction operation, during fan-in at every node of the tree a local reduction operation takes place, and during fan-out the final reduced value is actually broadcast to each of the threads.

2.7 Implementation of the Synchronization Operations

The synchronization operations are implemented using a hierarchical tree algorithm, which includes a fan-in followed by a fan-out. The algorithm is hierarchical because it caters to our hierarchical machine model of three levels - *processor-contexts*, *subdomains*, and *domains*. At the lower two levels, the algorithm used is a multithreaded tree barrier algorithm for shared-memory machines. At the *domain*-level, a message-passing tree synchronization algorithm is used.

2.8 Object-Oriented Implementation of the Synchronization Algorithms

The main data structures are the tree of nodes, at the three levels of *processor-context*, *subdomain* and *domain*. There are two types of nodes: a *FanInNode* and a *FanOutNode*, both derived from a base class *NodeBase* (see figure 4).

Barrier Synchronization

For barrier synchronization, a *BarrierNode* is multiple-inherited from *FanInNode* and *FanOutNode*. Corresponding to these node objects, other objects are defined that actually create the nodes, establish the fan-in/fan-out relationships, and perform the barrier operations. *FanInObj*, *FanOutObj* classes are defined to be inherited from a base *SynchronizationObj* class (see figure 5). A *BarrierObj* class is multiple-inherited from *FanInObj* and *FanOutObj* classes. We use virtual inheritance to avoid the ambiguity of multiple base class objects in the inherited class object. There is only one tree with different tree links for fan-in and fan-out information.

Reduction Operation

Let S_τ be set of all elements of type τ. Let $\wp(S_\tau)$ denote the set of all finite subsets of S_τ. Let $S_\otimes^\tau$ be the set of all commutative and associative binary operators/functions defined over elements of type τ. Then a reduction operator, ρ, is defined from $S_\otimes^\tau \times \wp(S_\tau) \longrightarrow S_\tau$. A reduction operation is the application of an operator $\otimes \in S_\otimes^\tau$ to a set of elements of type τ and returns a result of type τ.

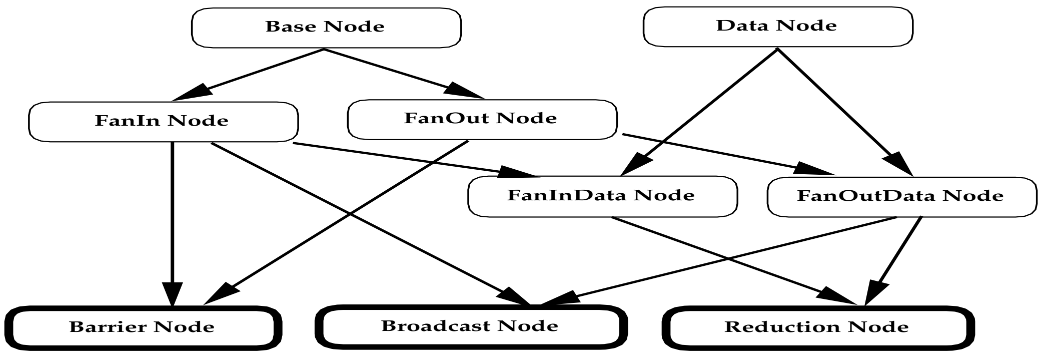

Figure 4: Node Object Inheritance Hierarchy for the Different Data-Parallel Synchronization Operations

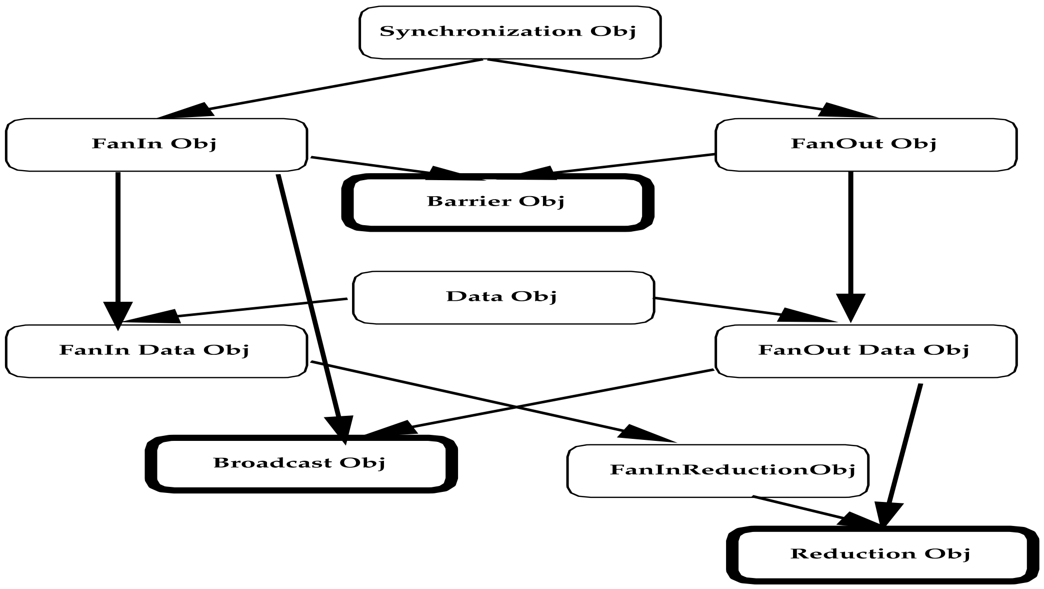

Figure 5: Synchronizer Object Inheritance Hierarchy for the Different Data-Parallel Synchronization Operations

Operationally, if S_τ is the set $\{a_1, a_2, a_3, \ldots, a_n\}$, then

$$\rho(\otimes, S_\tau) = \otimes(a_1, \otimes(a_2, \otimes(a_3, \ldots\ldots \otimes (a_{n-1}, a_n))))$$

is one way of applying the operator.

In the implementation of the reduction operation, local reductions are performed at each subtree during the fan-in. The final reduced value is broadcast to all the threads during the fan-out phase. Thus,

if $\xi_{\nu,\tau}$ is the value stored in the current node ν with k children

$\forall\ C^i_\nu$, where C^i_ν is ith child of ν, $i = 1, \ldots k$

$\xi_{\nu,\tau} = \xi_{\nu,\tau} \otimes \xi_{C^1_\nu,\tau} \otimes \xi_{C^2_\nu,\tau} \cdots \otimes \xi_{C^k_\nu,\tau}$

Coir uses object-oriented paradigm to provide a type-safe synchronization mechanism. Given any commutative or associate binary operator that operates on some user-defined data-type, rope-specific reduction operation can be performed. The user defines an appropriate functor (function object) that is equivalent to the binary operator. To allow the fan-in and fan-out trees to perform reduction and broadcast respectively, the *FanInNode* and *FanOutNode* classes are customized to be data-rich (see figure 4). So a *DataNode* class is defined and *FanInDataNode* and *FanOutDataNode* classes are defined which multiply inherit from *FanInNode* and *FanOutNode* respectively and from *DataNode.* The *ReductionNode* multiply inherits from *FanInDataNode* and *FanOutDataNode.* The *FanInObj* and *FanOutObj* synchronization objects are customized too, to define *FanInDataObj* and *FanOutDataObj* classes, which also inherit from a *DataObj* class (see figure 5). In the fan-in stage, in addition to fan-in synchronization, reduction operation is performed, and hence the customization *FanInReductionObj.* In the fan-out stage, a thread, in addition to notifying the thread corresponding to its children nodes, also copies the data to those nodes.

2.9 Broadcast Operation

In *Coir* the broadcast operation is combined with synchronization. In a broadcast operation, one thread contributes a value and all the other threads receive the value. All the threads have the knowledge of who the contributing thread is. This operation can also be written as a fan-in followed by a fan-out. In the fan-in phase, all the threads just hand-shake to notify that they have arrived at the operator and the fan-out phase consists of the actual data broadcast.

The fan-in data structure for the broadcast operation is same as that for barrier and the fan-out is the same as that for reduction.

2.10 Callback Mechanisms and Synchronization Operations

The reduction and the broadcast operation involve computation and communication of user-defined objects. Providing a type-safe implementation for the same involves issues similar to providing type-safe callback mechanisms used in object-oriented implementations of user-interface libraries. *Coir* defines a *Synchronizer data* class which satisfies the conditions for a good callback mechanism. *Synchronizer data* objects are user-defined objects that are required for a synchronization operation. By requiring these objects to be defined and compiled at the application-level, the process of checking for type correctness and issues of type safety is delegated to the user code. The library code for these operations just keeps a pointer to the data type as a *void** pointer. Creating, accessing, deleting, and updating the actual data is done through user-provided functions. These functions are wrapped into functor objects. The synchronization library understands only the functor base classes. It only cares about the fact that a functor of a particular type supports the appropriate '()' operator. Each of the functor classes support the '()' operators as virtual functions and any inherited concrete class of the abstract class should provide an appropriate implementation of the same. The synchronization implementation which traverses the synchronization tree does not do any data-specific operation on its side. It just has access to the functors and to the data pointers provided by the functors on the application side. It passes it around as a generic pointer and the data value represented by the pointer is used or modified only in the application-specific routines corresponding to the functors. The rest of the type checking is done at the application code level. On the application side there is a cast from a generic pointer type to the appropriate pointer data type. This is a *safe* cast because the same entity that removed the type reinstates it. By using templates for the functors the system lets the compiler generate the appropriate functors for different types in a correct manner, thus providing genericness. Figure 6 shows a ReducerFunctorBase class from which a templatized class - ReducerFunctorTemplate - is derived. This class enables templatizing the binary reduction operation based on the type DataType. A user may inherit from this class (like the AddReducer class shown in the figure). Such an inherited class just needs an instantiation of the datatype-specific binary operation which is achieved by defining the virtual operator '()' [1]. If DataType has an overloaded '+' operator, the '$x + y$' above uses the overloaded operator and if '+' is inlined then it is inlined in the body of '()'.

2.11 Communication between Ropes

Coir allows thread-to-thread message-passing, and rope-to-rope message-passing. In the current implementation, rope-to-rope message-passing is implemented in terms of thread-to-thread message pass-

[1]It might be possible to avoid the cast from *void** to the appropriate data templatizing the entire reduction algorithm. This would mean that the barrier and the broadcast algorithms also should be templatized. Also since reductions are rope-specific, this would mean templatizing part of the interface to synchronization operations of the rope class. This will increase the object file size and is unfavorable. At the same time, templatizing the functors permits inlining of the lowest level binary operators (the implementation of '()' in AddReducer in the figure.)

```
class ReducerFunctorBase {
   public:
     ReducerFunctorBase() { }
     virtual void operator()(void* res, void* lef, void* rt) = 0;
};

// User will used instantiations or derived classes of the following template for
// instantiating reducer functors
template <class T> class ReducerFunctorTemplate : public ReducerFunctorBase {
  public:
    ReducerFunctorTemplate<T>() : ReducerFunctorBase() { }
    T operator()(T& x, T& y);
    void operator()(void* result, void* left, void* right) {
        *(T*)result = (*this)(*(T*)left, *(T*)right);
      }
};

// Instantiation of a Sum Reduction Operation
class AddReducer : public ReducerFunctorTemplate<DataType> {
  public:
   AddReducer() {}
   DataType operator()(DataType& x, DataType& y)  { return x+y; }
};
```

Figure 6: A ReducerFunctor class and an Instance of its Derivation

ing, by identifying a 'leader' thread in each rope and directing the messages to/from it. Our goal is to have a direct mapping to the MPI communicator mechanism. We cannot use the MPI inter-communicator mechanism directly because rope distribution domains can overlap and the process groups in inter-communicators cannot. Rope-to-rope communication can be implemented by creating a process group composed of all the contexts in the two ropes taking part in communication and using intra-communicators. Another way is to use a combination of intra- and inter-communicators.

3 Applications

Light-weight data-parallelism can be used in data-parallel applications that involve regular SPMD-style data-parallelism, or nested data-parallelism corresponding to statically nested for-loops, or hierarchical data-parallelism where the levels of nesting is dynamic in nature. Since the data-parallel objects are dynamic in nature, a rope can be created for each set of data-parallel operations in the application and rope operations can be scheduled on different ropes simultaneously. Here we give three examples, one each for the three types of data-parallel applications.

3.1 PDE Solvers

In this example, two 2-dimensional partial differential equation solvers are scheduled simultaneously by scheduling two ropes, say r_1 and r_2, each of size equal to the number of available processors. Thus, in effect, there are two threads on each processor performing operations related to different solvers. In both the ropes, r_1 and r_2, each thread participating the PDE data-parallel operation executes a loop.

```
Loop N times {
    C: Update the local-grid using 5-point averaging.
    M: Exchange data with the 4 threads responsible for 4 neighbor-grids;
}
```

Each iteration of this loop can be considered to be a communication/computation phase - it has a communication component (M) where a thread communicates and exchanges data with four threads, holding the data for the neighboring grids of its data; and a computation component (C) where the thread updates its local grid-data through 5-point averaging. In the experiment, we introduce a delay of the order of C in each phase (iteration) given above, in one of the threads in r_2. The delay is introduced in a round-robin fashion. Thus during phase i, thread $i\% \mid r_2 \mid$ [2] causes the delay. Because of this delay, the remaining $i\% \mid r_2 \mid$ threads are blocked at the synchronization point C for a period of time related to the delay. During this period, the threads that are waiting get descheduled, and the threads from the other rope r_1 are run. Since the delay caused by the delaying thread is the same as the time taken to compute in a phase, the threads from the other rope can compute during this time. Basically, this algorithm ensures that all the threads do not arrive at the synchronization point at the 'same time'. If the computation takes C time and the communication takes M time, a program with no multithreading would take $N(3C + 2M)$ units of time while the multi-threaded program described here would take about $N((9C + 1)/4 + 2M)$ units of time if N is large enough to amortize the cost of context-switching due to multithreading. Figure 7 shows a fraction of the results from running the overlapping solvers. The figure has 5 horizontal lines. Each line has solid and dashed segments. The solid segments indicate computation, while the dashed segments indicate blocking on communication. The figure depicts two threads chosen, t_{r_1} and t_{r_2} on the same processor from r_1 and r_2, respectively. The thread t_{r_2} is known to have phases of longer computation because of the delay introduced. The

[2] $\mid r \mid$ denotes the number of threads in the rope r.

line labeled 'A' shows the behavior of t_{r_1} in the absence of t_{r_2} (ie. when its rope is run by itself without the rope of t_{r_2} run). It can be seen that there are very short breaks of communication between computations. This is because all the threads do the same amount of computation and arrive at the synchronization point at about the "same time". The line labeled 'B' shows the behavior of the thread t_{r_2} when its rope is run by itself. The line shows significant breaks of communication because of the delays caused by one of the threads in its rope. The line labeled 'C' shows the behavior of thread t_{r_1} when both the ropes are run together. Communication segment breaks of about the same length (or twice the length) of the computation segments can be seen. The line labeled 'D' shows the behavior of t_{r_2} when both the ropes are run together. The communication and the computation segments are of the same length except when the thread itself causes the computation delay when the segment is twice the length. The line labeled 'E', shows the combined utilization of the CPU on the processor on which both of these threads reside. The line consists of mainly solid segments indicating that the communication and computation overlap has successfully taken place.

3.2 Radiosity Through Ray-tracing

Ray-tracing is used to compute the radiosity in the preprocessing stage of graphic rendering systems [13]. In this example, we assume a system consisting of a fixed set of polygons (triangles) representing parts of objects in a scene. For the sake of simplicity, one of the patches (the 0th triangle), is counted as the light source. Energy (light) emits from this triangle and falls upon the remaining polygons of the scene. Based upon the reflective properties of an object, part of the energy is reflected back into the system and falls on other objects of the system. A ray-tracing approach shoots rays from the light source and checks which other triangle the ray hits. If it hits a triangle, none or part of the energy in the ray is reflected back. Each ray that is emitted from the triangle representing the light

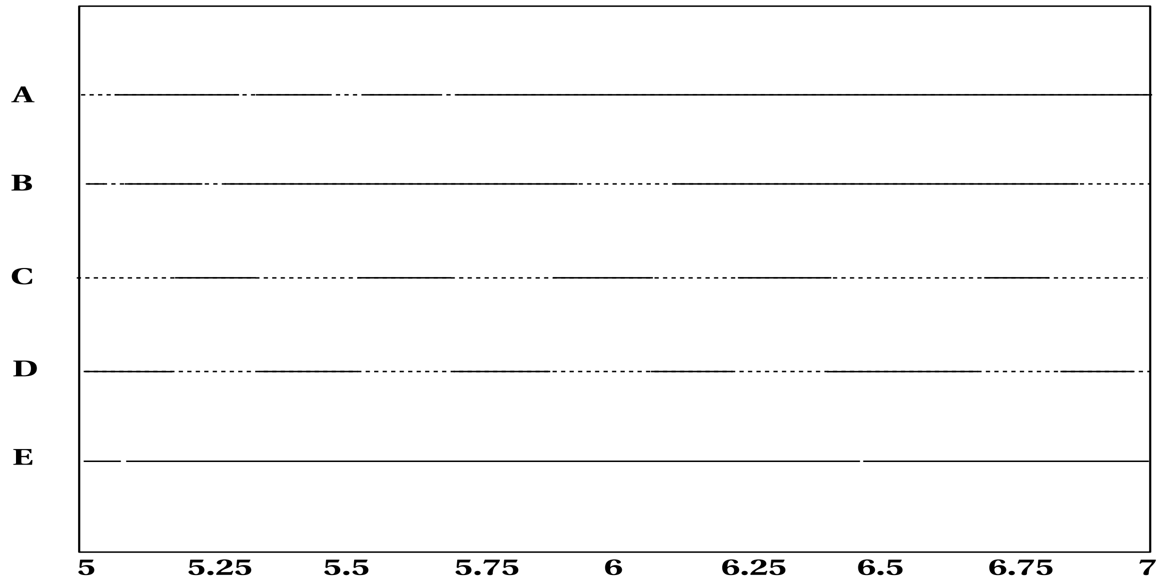

Figure 7: Overlap of computation and communication between threads of two different ropes; Two threads are chosen from each of the ropes and a snapshot of their behavior is shown here. See text for description of the figure.

source is traced through the system until the ray is no longer reflected back. The sequential algorithm consists of three nested loops - the outer loop indexes over the rays; the inner two loops trace the ray through all its reflections until it is no longer reflected. The outer one of these is a 'while loop' which iterates over the different reflected instances of the ray and terminates when the ray is no longer reflected. The innermost loop iterates over the set of triangles to see if the ray hits a triangle. If so, it accumulates the radiosity of the triangle and, depending on the reflectance of the triangle, generates a new ray in some direction with some energy. The outermost and the innermost loops are parallel, and the middle 'while' loop is sequential.

In the *Coir* model, a customized rope for rays and a customized rope for triangles is created. Since, for each iteration of the outermost *foreach*, there is an instance of the inner *foreach* loop, multiple instance of the triangle ropes may be created. Also, the inner *foreach* loop contains a reduction operation, which involves finding the triangle, if any, that a ray hits. We conducted experiments to find out how many triangle rope instances are required for each ray rope and how to distribute the ropes based on the amount of parallelism in each of them. Figure 8 shows the results on the SGI/Power-Challenge.

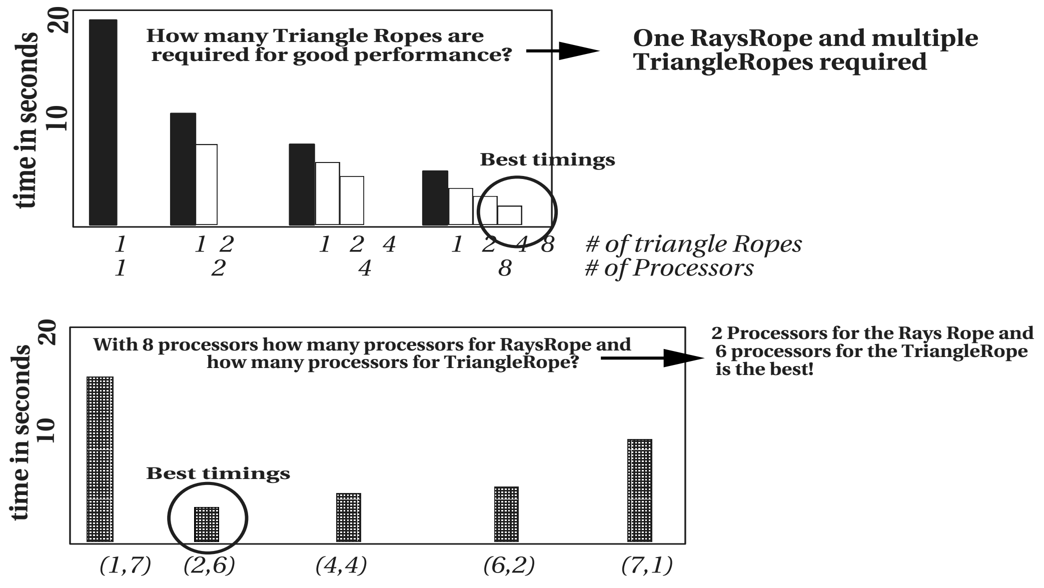

Figure 8: Performance of the radiosity algorithm

3.3 Adaptive Volume Integration

Adaptive volume integration is a multi-level numerical integration technique, where refinement of the grid points are done only in sections where the integrand is badly behaved or has singularity. Figure 9 shows the pseudo code for an adaptive volume integration algorithm [2] on the SGI/Power-challenge. In this example the threads in the ropes were allowed to migrate freely. The levels of nesting are not known at static time and the decision about whether or not a new rope should be created is made at run time based on the system state. Figure 10 shows the performance. It can be seen that the non-threaded version runs slightly faster than the *Coir* version on a single processor. This is because of the overheads of the *Coir* runtime system. Corresponding to a rope size of r, the sequential non-threaded version runs a loop of size r. The results shown here are preliminary in the sense that ropes of fixed sizes are created at every level of refinement, and no attempt is made to do intelligent load-balancing. The *Coir* system provides sufficient state information and hooks to make dynamic decisions, so we believe that it should be possible to achieve better performance by creating ropes of appropriate sizes at different levels of refinement and choosing between sequential and parallel refinements at any level.

```
class GridRope : public Rope<GridThread> {
  ...
  Solve(...)  {
   SolveLocalGrid();
   if (NeedToRefine(...)) {
      if (SufficientGranularity(...)) {
         GridRope newRope(...);
         newRope.Solve();
       }
      else
        SequentialSolve();
    }
  }
}
```

Figure 9: Pseudo-code for Adaptive Volume Integration

4 Related Work

There are other systems that provide light-weight mechanism for data-parallelism. Iterates [7], Presto [12] and Coir++ [18] model data-parallelism for shared-memory machines. ActorSpaces [1] and CA [4] in actor systems, provide aggregate computations on top of message-driven control models. Gang-scheduling [19] in multi-threaded operating systems also involves a group mechanism. The scheduling

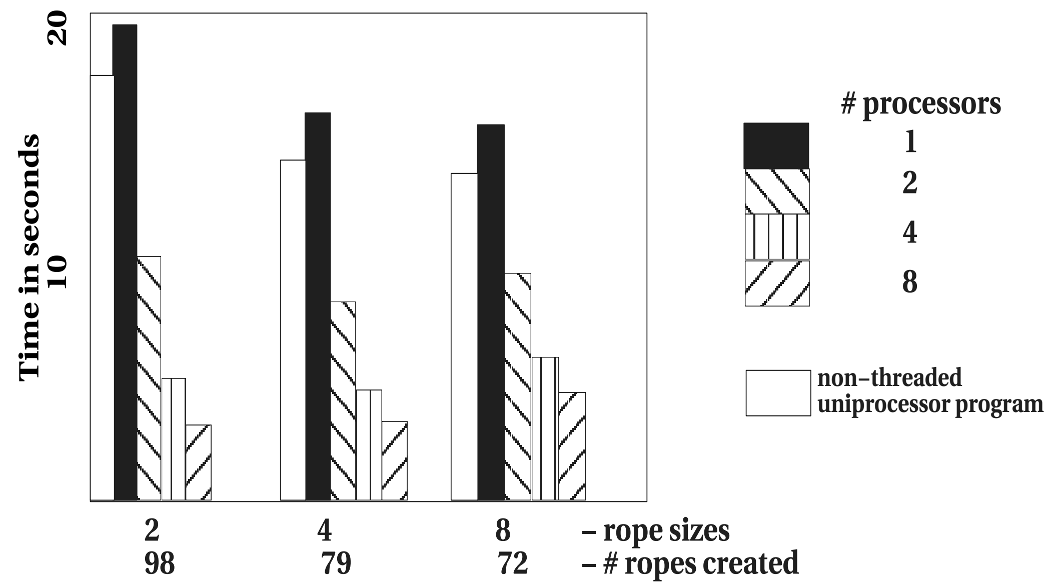

Figure 10: Performance of the adaptive volume integration

mechanism there involves scheduling and descheduling all the threads in a group at the same time, unlike *Coir* where threads in a rope are scheduled independent of each other. Other C language-based multithreaded runtime systems like Chant [8] and Nexus [5] are also adding data-parallel layers on top. Though there are similarities between our system and all of these systems, we differ by the fact that, in *Coir*, data-parallel objects appear like normal C++ objects, and at the same time have a direct mapping to the underlying control mechanism of DO loops or collection methods. Further, synchronization operations like barrier, reduction, and broadcast are defined at the level of these objects, and reduction operations can be on user-specified data-types. Our model subsumes a variety of today's shared-memory and distributed machines, and can be adapted to SPMD or master-worker programming model.

5 Conclusions

In this paper we discussed how *Coir* supports light-weight data-parallel objects. These objects support asynchronous data-parallel operations, and synchronization operations. These objects appear like normal C++ objects, but at the same time follow data-parallel semantics. The model of execution of these objects is more powerful than the HPF or C* [20] models.

We gave examples of how multithreading can be used to effectively use data-parallelism in three classes of applications - flat data-parallel, static nested data-parallel, and hierarchical nested data-parallel. As the next step, we plan to incorporate optimization mechanisms, based on the system state information provided by *Coir*, to make smart decisions on creation and scheduling of these data-parallel objects.

References

[1] Gul Agha and Christian Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *ACM Sigplan Notices*, pages 23–32, 1993. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), San Diego, California, May 19-22, 1993.

[2] Kendall Atkinson. *An Introduction to Numerical Analysis.* John Wiley and Sons, 1978.

[3] François Bodin, Peter Beckman, Dennis Gannon, Shelby Yang, Allen Malony, and Bernd Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. *Proceedings, Supercomputing '93*, November 1993. ACM Sigarch and IEEE Computer Society Technical Committees on Supercomputing Applications and Computer Architecture.

[4] Andrew Chien and William Dally. Concurrent Aggregrates (CA). In *Second ACM Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices*, volume 25(3), pages 187–196, March 1990.

[5] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Systems. Technical report, California Institute of Technology, Computer Science Department, Pasadena, CA., August 1994.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994. The example programs from the book are available from ftp://info.mcs.anl.gov/pub/mpi/using.

[7] Dirk Grunwald and Suvas Vajracharya. The Design of an Object-Oriented Runtime System for Integrated Task and Object Parallelism. Technical report, Department of Computer Science, University of Colorado, Boulder, CO 80309-0430, July 1994.

[8] Matthew Haines, David Cronk, and Piyush Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing 94, Washington D.C.*, November 1994.

[9] High Performance Fortran Forum. Draft: High Performance Fortran Language Specification, Version 1.0. also available as technical report CRPC-TR92225 from Center for Research on Parallel Computation, Rice University, January 1993.

[10] IEEE. *Thread Extensions for Portable Operating Systems (Draft 6)*, February 1992. P1003.4a/6.

[11] Alexander Klaiber and James Frankel. Comparing Data-Parallel and Message-Passing Paradigms. In *Proceedings of the 1991 International Conference on Parallel Processing, Vol. II (Software)*, pages 11–20. CRC, Inc., August 1993.

[12] Kendall Square Research Parallel Programming Guide, July 1993. Kendall Square Research Corporation, Technical Documentation.

[13] Peter Shirley. Radiosity via Ray-tracing. In James Arvo, editor, *Graphics Gems II*, chapter VI-4, pages 306–310. Academic Press, 1991.

[14] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley Publishing Company, 1994.

[15] Neelakantan Sundaresan. *Modeling Control and Dynamic Data-Parallelism in Object-Oriented Languages*. PhD thesis, Computer Science, Indiana University, 1995. upcoming.

[16] Neelakantan Sundaresan and Dennis Gannon. A Thread-Model for Supporting Task and Data Parallelism in Object-Oriented Parallel Languages. In *International Conference on Parallel Processing*, pages 45–49, August 1995.

[17] Neelakantan Sundaresan and Dennis Gannon. Experimental Evaluation of *Coir*: A System for Control and Data Parallelism. In *Seventh International Conference on Parallel and Distributed Computing and Systems*, October 1995. to appear.

[18] Neelakantan Sundaresan and Linda Lee. An Object-Oriented Thread Model for Parallel Numerical Applications. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994. Sunriver, Oregon.

[19] A Tevanian Jr., R Rashid, D Golub, D Black, E Cooper, and M Young. Mach Threads and the UNIX Kernel: The Battle for Control. In *Proceedings of the USENIX Summer Conference*, pages 185–198, June 1987.

[20] Thinking Machines Corporation. *C* Release Notes*. Thinking Machines Corporation, Cambridge, MA, December 1990. Version 6.0 and 6.1.