Coir: A Thread-Model for Supporting Task- and Data- Parallelism in Object-Oriented Parallel Languages

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- and task-parallelism are two important parallel programming models. Object-oriented paradigm in parallelism provides a good way of abstracting out various aspects of computations and computing resources. Using an object-oriented language like **C++**, one can compose data and control representations into a single active object.

We propose a thread model of parallelism that addresses both data and task parallelism. Computation and communication can be overlapped by suspending a thread of computation which is waiting for an event and running an eligible thread of computation in its place. Threads naturally subsume task-parallelism. Threads are encapsulated into thread objects may be grouped into rope objects [22, 20], that span the parallel machine domain, for collective computation and communication. Thus data-parallelism can be supported. Since rope objects are parallel objects, they can be customized, interestingly, in a serial or a parallel manner. Spatial transparency of objects is achieved by global pointer templates.

We present results from a prototype system running on the SGI Challenge and the Intel Paragon.

keywords: *task-parallelism, data-parallelism, thread, rope, object-oriented paradigm*

1 Introduction

User-level threads provide a good way of abstracting away from physical processors and processes and expressing parallelism at a level closer to the logic of the user algorithm. Threads are to processes as processes are to physical processors. Even on a single processor system, threads provide for coroutine-style programming to express control separation. Since the cost per context-switch for these threads is just a few microseconds, a thread which cannot proceed because it has to wait for some condition to be satisfied(like data arrival, shared data consistency or control consistency) can enable another thread that is ready to execute. Thus maximal processor-utilization can be achieved at a small cost of context-switch.

Pthreads ([12, 16]) is an emerging standard of thread interface at C language level. There are a variety of other thread libraries - those from the vendors include DCE threads, SGI sprocs and SUN light weight processes; and others from the university communities include NewThreads[6](which has a communication model) and Cthreads[4] and its extensions[17]. Those with **C++** interface for sequential and shared memory machines include Awesime[10] and Presto[2]. A number of parallel object-oriented languages address task parallelism; pSather[18](which is based on a monitor-model for NUMA machines and supports Eiffel-like syntax), Mentat[9](which is based on a coarsed grain data-driven model, with extensions to **C++**), Charm++[13](which is based on message-driven model, with **C++** extensions), and CC++[3](which provides parallel control constructs as extensions to **C++**).

We propose an portable object-oriented thread-model for parallelism that addresses shared memory machines and distributed memory machines and hierarchical architectures in general. Thread objects provide abstractions for control parallelism and rope objects provide abstractions for data parallelism. These objects can be customized to specific application target by using object-oriented features like Inheritance, polymorphism, dynamic dispatch and parameterized data types. This model was introduced for shared memory machines in [22] and later generalized in [21, 19]. A detailed discussion on rope objects and their implementation can be found in [20]. Our approach is library-based and addresses both data and task parallel issues. The library provides a target model and runtime system for parallel extensions to **C++** like **pC++**[15]. The relevance of this model to **pC++** is discussed in [1]. The library itself is extensible, with Inheritance

and polymorphism, for an advanced user to be able to write programs in **C++** using the model. It is built in a hierarchical manner, for easily portability and interface with other similar systems.

The paper is organized as follows. In the next section we introduce a portable machine and memory model that subsumes both shared and distributed memory machines. This is followed by a discussion on our model of parallel computation where we define thread and rope objects and features for building a parallel object-oriented language. This is followed by performance results and conclusions.

2 Machine and Memory Model

A parallel machine can be visualized as a 3-level hierarchy consisting of *processor contexts*, *subdomains* and *domains*.

A *processor context* is a processing resource consisting of one or more processors each with a set of registers and a program counter. The processors share an address space context. The processor(s) may be shared across processor contexts - a real or a logical CPU resource supported by the hardware. Mapping multiple physical processors to a processor context facilitates threads within the context to migrate across the processors in the context, thus providing a simple form of load balancing.

A *subdomain* is a set of (one or more) processor contexts that share an address space. It represents a tightly coupled architecture. Each processor context has its own local space within this space, but processor contexts share program globals and may be a heap within the same subdomain. Data sharing and exchange between subdomains may be through message passing or a global pointer abstraction of the data structures.

A *domain* is a set of subdomains for which collective communications and synchronizations are supported. It can be thought of as the set of address subspaces, processors and shared control hardware that is available to a user process when it is loaded.

In this processor-hierarchy, there is a hierarchy of memory: memory private to a thread; memory private to a processor context shared between threads in the processor context and memory private to a subdomain shared between threads in the subdomain;

Resource Objects

Typically, the resource objects are created and handled by the system hence their constructors are hidden from the user. What the user sees is methods that create user-copies of these objects and manipulate those copies and query these objects. Also, the user usually operates through the domain object which represents a domain and is a composition of subdomain objects (representing subdomains) each of which, in turn, is a composition of processor context objects (representing processor contexts). A domain object denotes the space over which threads are distributed and computation is done. Operations on a domain object lets the user define or compose subsets. These operations are useful in defining *rope contexts* and implementing nested parallelism[20].

3 Threads

A *thread*[1] is a unit of control for parallel execution. A thread may be created whenever the program sees a separate control of execution independent of the current thread. Thus, threads can create other threads and the parent and child threads can run simultaneously depending on availability of processing resources. If sufficient processing resources are not available then those threads of computation that cannot be run, but are ready to run, are kept in a *ready* queue. A running thread can yield the processor to another thread and move to a *ready* state or go to sleep and give up the processors by moving to a *blocked* state. The difference between yielding and blocking is that, the yielding thread is still ready to run and will be run by the scheduler if a processor is free. A blocked thread will have to be woken up(unblocked) by another thread to be able to run again. Thus a yielding thread, unlike a blocked thread, may take up CPU cycles and delay running of other eligible threads. When a thread has finished computation it moves to a *dead* state, which is a point of no return. In this state the thread control structure is lost and the parent thread can look at the left-over passive data structure for any results.

3.1 Thread Objects

[1] In this article threads stand for user-level threads.

Figure 1: Thread Class

```
class Thread
{
  // constructor to create a transient thread!
  Thread(FunctionType function, ArgType argument, const int immediate = 1);
  // constructor to create persistent thread!
  Thread(const int immediate = 1);
  // Assign a task to a persistent thread
  TaskId Run(Function function, ArgType argument);
  // Wait for the task to be done
  void Wait(TaskId tid);
  // An executing thread may call the following functions
  static void Block(); // Block myself(deschedule)
  static void Yield(); // Yield(not block) so that another thread may run
  static void Exit(void* result); // Exit. kill myself
  static Thread& Self(); // Query. who am I ?
  // Join: wait for a thread to finish executing
  void Join(void** ret_val);
  // Unblock a blocked thread
  void Unblock();
  // Cancel a thread (try to kill it)
  void Cancel();
   ...
};
```

The fundamental unit of control for parallel execution is a *thread* object. Thread objects are typically created locally and migrate between processors in a processor context. There are two kinds of thread objects: *transient* and *persistent*. Transient threads are those that are created for one task and die at the end of that task. Persistent threads are created to wait around to be assigned tasks. These threads die when the object goes out of scope or is *deleted*. Figure 1 shows the class definition for a thread class. Note that whether a thread is transient or persistent is defined by the constructor that it invokes. When the constructor is given a function and its argument, then the thread is a transient thread and the life of the thread is the life of the function. When the constructor is of the second kind, it is a persistent thread. The constructors take an additional boolean parameter in which the user can specify if the thread activation should immediately follow the thread data initialization. This parameter defaults to *true*. It is useful to set this parameter to *false* when derived threads are created. A persistent thread is assigned tasks through the *Run()* method which returns a task identifier and the parent thread can wait for this task to finish using the *Wait()* method. A running thread may yield or block using the *Yield()* or *Block()* methods respectively. A transient thread may exit using the *Exit()* method. The argument to the *Exit()* method is the value returned to the parent thread. The parent thread can wait for the child thread to finish executing by invoking a *Join()* method on it. The argument to the *Join()* method is the address of the space in which the return value is expected. Since the *Block()*, *Yield()* and the *Exit()* methods are invoked by the thread itself, they are **static** functions

of the *Thread* class. A blocked thread may be unblocked using the *Unblock()* method on it. A *Cancel()* method on a thread is an attempt to kill the thread, or more precisely a hint to the thread that it is no longer required. The thread may take necessary action to die when it deems appropriate.

3.2 Communication between Threads

To function in a co-operative manner, threads need to communicate with each other to share data, information of control etc. Since our machine model applies to both shared and distributed memory machines, we need to think about interaction between two threads located anywhere in a domain - same or different processor contexts in the same subdomain or in different subdomains.

One can take two extreme views of communication between threads. A "shared-memory" view where all the threads in the domain are directly addressable and a "message passing" view where two threads, in order to communicate, should do so through messages. The shared-memory model is simpler to use. On the other hand, in the message-passing model the user has better control over data movement and granularity, and can combine data transfer and synchronization in a single operation.

Shared-Memory View

The concept of global objects is used to give a view that all objects belong to a global shared memory. A global thread object is a spatially transparent active object. The class interface is the same interface as the *Thread* class except that the constructor has information, if required, to create a thread remotely. The member-function interface is the same as that for *Thread* class except that, for threads which are remote it sends a message to the processor on which the thread resides and the necessary function is locally executed and a result is returned [8].

Message-Passing View

Message passing can take place between two threads in the same processor context or different processor contexts in the same subdomain or different subdomains. Mainly two primitives are used to send and receive messages:

Figure 2: Derived Thread Class for Bitonic Sort

```
class DataThread : public
Thread
{
 public:
  DataThread()
    : Thread(0) {
    vector_ptr = new
Pvector(BlkSize);
    Start(); // trigger the thread
  }
 private:
  Pvector* vector_ptr;
  void merge(int distance);
  void localSort(int flg);
  void grabFrom(int dist);
  void localMerge(int dist, int
flg);
  void localBitonicMerge();
};
```

```
struct Pvector {
 public:
   Pvector(const int size);
   ~Pvector();
   E* part; // BlkSize;
   E* tmp; // BlkSize;
};
void DataThread::merge(int
dist)
{
  int flg = 1;
  while (dist > 0) {
     grabFrom(dist);
     DataRope::Synchronize();
     localMerge(dist, flg);
     dist /= 2;
     flg = 0;
     DataRope::Synchronize();
  }
  localBitonicMerge();
}
```

- Send A thread can send a message to any thread on any context or node.

  Send(int msg_id, int thread_id, int context, int node, void buffer, int buffer_len);*

- Receive A thread can received a message from another thread on any context or node.

  Receive(int msg_id, int thread_id, int context, int node, void buffer, int buffer_len);*

Messages can be *synchronous* and *asynchronous*. In *synchronous* message passing, for every sending thread there is a matching receiving thread and thus sends and receives are matched. In *asynchronous* message passing, the receiving thread is not expecting the message from the sending thread. The message is received and stored up by a server or a watchdog thread, and then forwarded to the thread to which the message was intended. Asynchronous message passing can be effectively used when the receiving thread does not know which thread it is receiving a message from.

3.3 Customizing Threads

Threads can be customized to application-specific objects using the **C++** Inheritance facility. Inheritance can be used to build data- and control- abstraction-rich active objects. Figure 2 gives an example of a customized perstistent thread. Notice that the constructor breaks up the thread object creation and activation so that the derived object initialization can be done before the thread is activated. The *DataThread* class is used in customizing a rope class for a bitonic merge-sort to be discussed later.

Figure 3: A Rope Class

```
class Rope
{
  Rope(const int num_threads, Distribution& dist,
       Domain& rope_domain, Domain& curr_domain);
  // Return the size of the rope
   int Size() const;
  // domain over which the rope is defined
  Domain& RopeDomain() const;
  // Parallel execution
  TaskId Execute(FunctionType func, ArgType arg,Domain& curr_domain);
  void Wait(TaskId);
  // return the index of the current thread
  // in the rope called from a parallel function
  static int SelfIndex() const;
  // synchronization between threads in a rope
  static void Synchronize();
  // reduction
  static void Reduce(ReducerObj&);
  // broadcast
  static void Broadcast(BroadcastObj&, const int bcaster);
};
```

4 Collections of Threads

A rope is a group of threads working together in a co-operative manner. As threads provide an abstraction for task parallelism, a rope, where each thread is assigned the same task with different data provides an abstraction for data-parallelism.

Threads in a rope are distributed over a domain object which defines the *distribution context* of the rope. Distribution of the threads is specified at rope creation time. Threads in a rope enter a parallel function together and barrier-synchronize at the end of the function. During the computation of the parallel function, the threads in the rope can enter a rope-specific barrier. A rope object also provides abstractions for global reduction and broadcast. Figure 3 shows a description of the rope class.

Rope Contexts

One of the advantages of executing programs in an SPMD fashion, as opposed to a master-worker or client-server fashion is that the processors do not need to communicate for work assignment. Each processor knows its part of the work and proceeds with it as far as it can. Communication takes place only when the processors need to exchange computation data. This provides an efficient computation model when the computation is regular. For problems where work cannot be distributed evenly across the processors through the life of the program this model results in unnecessary barrier synchronization, communication and processor-idling.

We extend the SPMD model to threads and define a context model.[2] Typically there are two kinds of contexts: an *invocation context* and an *execution context*. The invocation context is the context from which rope creation and execution methods are invoked. The execution context is the context in which the invoked methods are actually run. If the invocation context is different from the execution context, then a representative invoking thread sends messages to one or more representative execution threads in all those processors of the execution context which do not belong to the invocation context. Thus unnecessary messages are avoided in the parts where the two contexts overlap. In an SPMD model, the invocation and execution contexts are the same. while in a master-worker model, the invocation context and execution context are disjoint. Choosing the context-model minimizes unnecessary message passing at the same time avoids the disadvantages of unnecessary synchronization unlike in the SPMD model. The context model also provides a good way to define, represent and compile constructs for nested-parallelism[20].

4.1 Customizing Rope Objects

Rope objects can be customized to application-specific data-parallel active objects using the Inheritance facility in **C++**. Since a rope is a *parallel object*, one can think of customizing it in two ways:

- Serial Inheritance

  Since a rope class is a **C++** class, one should be able to derive from this class and provide further abstractions of the rope methods or define a 'data-rich' derived class. The data-richness is added at the level of the rope class.

- Parallel Inheritance

  Since a thread object can be customized to build data-rich threads, one can think of a rope object built out of customized threads. Here the abstraction and the data-richness is added at the level of the threads that constitute the rope. This qualifies as parallel Inheritance where each component of the parallel class inherits from a corresponding component of a base parallel class. To support parallel Inheritance, the rope class is defined as a template, parametrized by a thread class. The user can

[2] a context is different from a processor context discussed earlier. A context is a subset of a domain whereas a processor context is a member of a subdomain.

Figure 4: Bitonic Sort Using Our Library

```
class DataRope : public
RopeTemplate<DataThread> {
 public:
   DataRope(const int size)
     :
RopeTemplate<DataThread>(size)
{ }
   virtual ~DataRope() { }
   void ParSetKey() {
     TaskId tid =
Execute(DataThread::SetKey);
     Wait(tid);
   }
   void ParSort() {
     TaskId tid = Execute(Sort);
     Wait(tid);
   }
 private:
   static void Sort(void* x);
};
```

```
main() {
  DataRope X(RopeSize);
  X.ParSetKey();
  X.ParSort();
}
void DataRope::Sort() {
  DataThread& this_thr =
Scheduler::Self();
  DataRope* this_rope =
Scheduler::SelfRope();
  int size = this_rope→Size();
  this_thr.localSort(1);
  DataRope::Synchronize();
  for (int i = 1; i < size; i *= 2)
    this_thr.merge(i);
}
```

instantiate it with any other class derived from the Thread class[3]

```
template <class ThreadClass> class RopeTemplate
{
...
};
```

The simplest instantiation of this template is the rope class which is

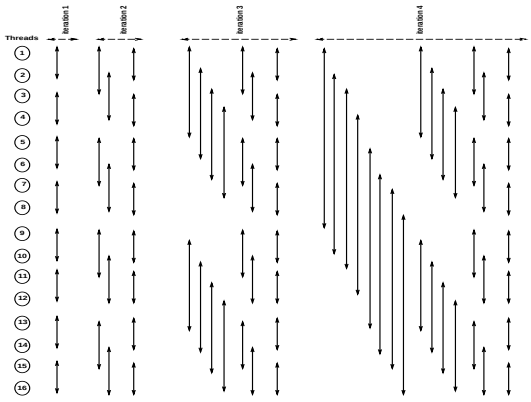
```
RopeTemplate<Thread>
```

5 An Example: Bitonic Sort

We describe a bitonic merge-sort program[14] written in our model(refer figure 4). The program was adapted from a parallel version bitonic sort for coarse-grain object parallelism, written in **pC++**[14]. The data, of size N, is divided among the threads of a rope of size R. Each thread in the rope is a customized rope is responsible for $\frac{N}{R}$ elements. We customize the thread class to a data-rich *DataThread* class(see figure 2) which holds the data in a *struct Pvector*. There are two fields in this struct - *part* which is the data and *tmp* which is an auxiliary buffer for data exchange with other threads. The *DataThread* class defines the *CreateThread()* and *DeleteThread()* methods for customized thread creation and deletion. The *DataRope*

[3] **C++** template syntax does not provide an obvious way to ensure that the arguments to a template can be only classes derived from a particular base class. We ensure that only classes derived from the Thread classes form the thread components of the rope by making assumptions about the Thread class in the rope template body code. Thus a program which tries to define a rope of non-thread objects will just not compile.

Figure 5: Pattern of data exchange for the bitonic merge sort when there are 16 threads



class is a *serial-parallel* inherited *Rope* class, which defines two parallel methods, *ParSetKey()*,which initializes the data to random numbers, and *ParSort()*, which sorts it. In the parallel sort, each thread invokes the method *DataRope::Sort()*. In this routine, a thread locally sorts its data and then synchronizes with the other threads in the rope. Then, in a loop of size $\log(R)$(whose index increments by multiples of 2), it calls *merge()*. The argument passed to this routine is the initial distance between the current thread and the thread with which it exchanges data. Every thread taking part in the *merge()* routine, decrements - in a loop - the distance *dist* between the current thread and its exchange partner by factors of 2. The thread acquires the data from the *dist*-neighbor in the *grabFrom()* routine, and then performs a *localMerge()*, which it takes the lower or the upper half - as the case maybe - by merging the local and the grabbed bitonic sequence. Outside this loop is *bitonicMerge()* which merge-sorts the bitonic sequence of the local data. Figure 5 shows the pattern of communication between threads for a 16-thread rope. Let us assume a data size of N, rope size of R whose threads are distributed over a domain with P processors. For sake of simplicity we assume that the domain consists of P processors and ignore the domain - subdomain - processor-context hierarchy. Each thread holds $\frac{N}{R}$ data elements. Let $R = kP$, where $k \geq 1$.

Every thread, in the parallel sort function, starts with a local sort whose average complexity is of the order $\Theta(\frac{N}{R}\log(\frac{N}{R}))$. For each thread, there are $\Theta(\log R)$ calls to the body of the *merge* routine ie. $\Theta((\log R)^2)$ calls to *grabFrom()* and *localMerge()*. *grabFrom()* involves exchanging data(communication) of size $\frac{N}{R}$ and we can assume that to be of $\Theta(\frac{N}{R})$. *localMerge()* is a linear algorithm and is again of the order $\Theta(\frac{N}{R})$.

bitonicMerge() is called once for every call to *merge()* ie. $\Theta(\log R)$ times and is a linear algorithm of the order $\Theta(\frac{N}{R})$.

Thus the cost, per thread, is of the order

$$\Theta(\frac{N}{R}\log\frac{N}{R}) + (\log R)^2\Theta(\frac{N}{R}) + \log R\Theta(\frac{N}{R})$$

which reduces to

$$\Theta(\frac{N}{R}(\log N + (\log R)^2)$$

Ignoring the cost for context switching and barrier synchronization, the parallel slack is of the order of $\Theta(k)$. Thus the total cost is

$$k\Theta(\frac{N}{R}(\log N + (\log R)^2)$$

which is

$$\Theta(\frac{N}{P}(\log N + (\log kP)^2)$$

This formula is very approximate as we do not take into account cost of context switches and thread management. This is offset by the effect of overlapping communication with computation[5, 7, 11].

For fixed P and N, the cost is $\Theta((\log K)^2)$; for fixed N and k, the cost is $\Theta((\log P)^2/P)$; and for fixed k and P, it is $\Theta(N \log N)$.

Assuming a sequential average cost of $N \log N$, the speed up is

$$S(N,P,k) = \frac{N\log N}{\frac{N}{P}(\log N + (\log kP)^2)} = \frac{P}{1 + \frac{(\log kP)^2}{\log N}}$$

When the size of the data is very large compared to the number of threads, $N \gg kP$, we get a speed up of P. The effect of thread distribution on communication is studied in [20].

5.1 Performance

We studied the performance of our library on the bitonic sort program. We concentrated on issues like independence of rope sizes to processors, interference of threads on the same processor within ropes and across ropes. We ran tests on a 20-processor SGI-challenge symmetric multiprocessor machine and on the Intel Paragon distributed memory machine.

Graph 6 shows the performance of a rope of size 16 on a 16-processor partition on both the machines for data sizes varying from 100,000 to 10 million. The SGI challenge the partition constists of 1 processor per processor context, 16 processor contexts per subdomain and 1 subdomain in the domain[4]. On the paragon we have 1 processor per processor context and 1 processor contexts per subdomain and number of subdomains in the domain as number of processors used for the computation, which in this case is 16. The performance on the SGI Challenge is much better than on the Paragon because on the SGI the messages are just pointer in the shared memory and hence are much cheaper than on the paragon. It can be seen that on both machines the program scales well for increasing data sizes.

Graph 7 shows the performance for different number of processors(with rope size = number of processors) for data of size 1 million on the two machines. Here again, the number of processors per processor context is 1. On the paragon each processor corresponds to a subdomain whereas on the SGI each processor corresponds to a processor-context. It can be seen that the speed-up is linear for increasing number of processors.

Computation and communication in the bitonic sort algorithm is quite regular. Also the threads synchronize after every step of data exchange and every step of local merge computation. Also the computation and communication times are comparable. In this situation, it does not make sense to have multiple threads per processor; if there are two threads on a processor, one precedes the other on a communication or a computation step and waits for the other to arrive at the synchronization step. Further, the communications are synchronous and if we assume similar processor speeds, the threads possibly arrive at the data exchange point at the 'same time'. Thus, having multiple threads per processor will cause some overheads of context-switch as the graph 8 shows for a 8-processor case for the two machines. This graph shows the performance for a data size of 1 million for an 8- processor partition - 8 subdomains in the case of the Paragon, and 8

[4] A shared-memory machine typically has 1 subdomain per domain

processor-contexts(each with one processor) in a subdomain in the case of the SGI challenge.

Load balancing can be studied by dividing up a set of processors into different combinations of sizes of processor- contexts and subdomains. For instance,an 8-processor partition on the SGI challenge may be partitioned in at least 4 ways:

- 1 processor per processor-context and 8 processor-contexts per subdomain
- 2 processors per processor-context and 4 processor-contexts per subdomain
- 4 processors per processor-context and 2 processor-contexts per subdomain
- 8 processors per processor-context and 1 processor-context per subdomain

Having multiple processors per processor-context provides for load balancing(by giving the threads more freedom for migration) but then thread scheduling is more expensive because heavier-weight locks are required as multiple threads might be active on the context at the same time. In order to study load balancing we scheduled multiple sorts on different ropes at the same time. With more sorts to perform, they perform better. Figure 9 shows three cases of partitioning for data-size of 100,000 elements :

1. the one labeled 'loadbalance.sgi1' corresponds to the case where the 8-processor partition is divided into a domain with 1 processor per processor-context and 8 processor-contexts per subdomain and a rope with 8 threads used per sort.

   The other 2 curves correspond to the cases with 2 processors per processor-context and 4 processor-contexts in the subdomain.

2. The curve labeled 'loadbalance.sgi2' is for 8 threads per rope per sort
3. The one labeled 'loadbalance.sgi3' is for 4 threads per rope per sort

It can be seen that when the number of sorts is 1 or 2, case 1 shows better performance than the other two cases. For fewer sorts, the multi-processor processor-contexts are starved for work, but at the same time pay a penalty for higher context-switch times. As the number of sorts increases, the total number of threads increase too, thus processor-starvation is reduced in cases 2 and 3. Also better load balancing is achieved. For instance, in case 1, when a thread is is waiting at a barrier, it can be rescheduled only on the same processor(as there is only 1 processor per context) even though a processor in some other context may be

Figure 6: 1 processor per proc.context, 16 proc.contexts per subdomain, rope size 16

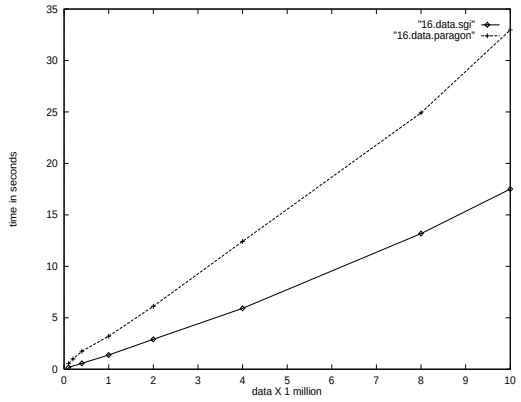
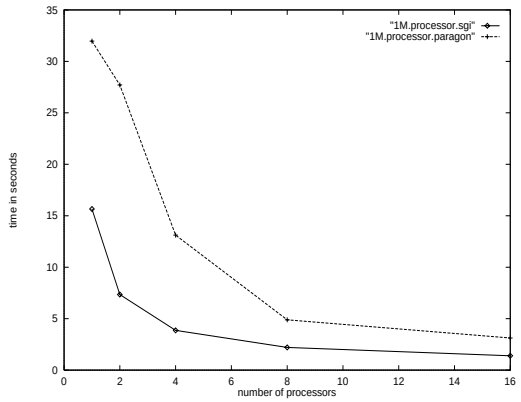


Figure 7: data size = 1 million, varying number of processors, rope size=# of processors



idle, whereas in cases 2 and 3 it can be rescheduled on the other processor within the processor-context if it is idle and looking for work. Since threads in the same rope are constrained by synchronizations, having fewer threads per rope and more independent ropes(more sorts) gives better performance. Hence case 3 posts better results than case 2.

6 Conclusions

In this paper we proposed an object-oriented thread-model for parallelism which addresses both shared-memory and distributed-memory machines. We addressed both task and data parallelism issues through thread objects and aggregate thread objects called rope objects respectively. We showed how to customize

Figure 8: For data size 1 million, #processors = 8, varying rope size

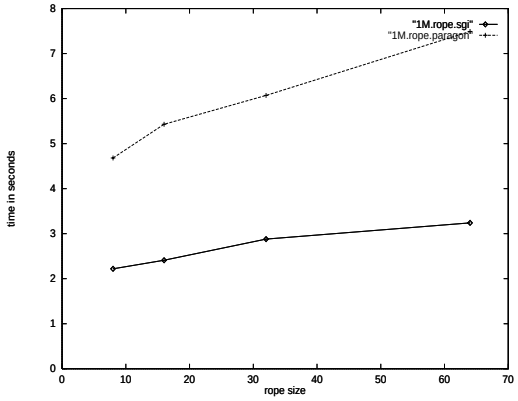
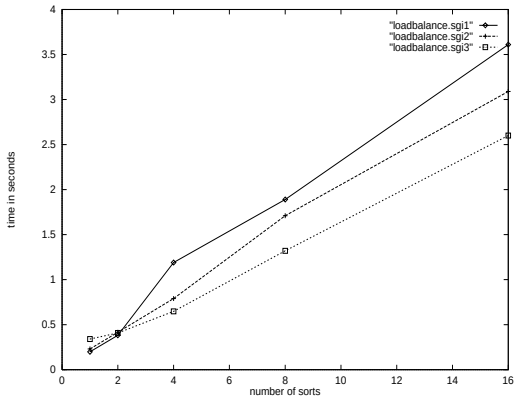


Figure 9: An 8 processor partition on the SGI divided up different ways



these objects in an application-specific way. Our performance study shows that context-switching in the thread environment does not adversely affect the performance of regular data-parallel algorithms and multiple data-parallel tasks. In the future, we will study the advantages of this model over a pure data-parallel model by studying irregular and adaptive algorithms with regular components. We are using this model and this library to build parallel language extensions in **pC++**-2.0 [1] and an interface for global active objects.

References

[1] Peter Beckman, Dennis Gannon, and Neelakantan Sundaresan. pC++ Meets Multi-Threaded Computation. In Jack Dongarra and Bernard Tourancheau, editors, *Proceedings of the second workshop on Environments and Tools for Parallel Scientific Computing*, Philadelphia, May 1994. SIAM.

[2] Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A system for object-oriented parallel programming. *Software-Practice and Experience*, 18(8):713–732, August 1988.

[3] Mani Chandy and Carl Kesselman. *Compositional CC++: A Declarative Concurrent Object-Oriented Programming Notation*. MIT Press, 1993.

[4] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[5] Edward Felten and Dylan McNamee. Improving the Performance of Message-Passing Applications by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference(SHPCC-92)*, 1992. Williamsburg VA.

[6] Edward Felten and Dylan McNamee. *NewThreads2.0 User's Guide*, August 1992. Williamsburg VA.

[7] 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.

[8] Dennis Gannon. Private Communication.

[9] Andrew Grimshaw. Easy-to-use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, May 1993.

[10] Dirk Grunwald. A Users Guide to AWESIME: An Object-Oriented Parallel Programming and Simulation System. Technical Report CU-CS-552-91, University of Colorado, Boulder, Colorado, November 1991.

[11] 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.

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

[13] Laxmikant Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object-Oriented System Based on C++. Technical report, University of Illinois, Urbana-Champaign, March 1993.

[14] Allen Malony, Bernd Mohr, Peter Beckman, and Dennis Gannon. Program Analysis and Tuning Tools for a Parallel Object Oriented Language: An Experiment with the TAU System. To appear.

[15] Allen Malony, Bernd Mohr, Peter Beckman, Dennis Gannon, Shelby Yang, François Bodin, and S Kesavan. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. *Proceedings, Supercomputing '93*, pages 588–597, November 1993. ACM Sigarch and IEEE Computer Society Technical Committees on Supercomputing Applications and Computer Architecture.

[16] Frank Mueller. Pthreads Library Interface. Technical report, Florida State University, July 1993.

[17] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A Machine Independent Interface for LightWeight Threads. Technical Report GIT-CC-93/53, College of Computing, Georgia Institute of Technology, June 1993.

[18] Stephen Murer, Jerome Feldman, and Chu-Cheow Lim. pSather monitors: Design, Tutorial, Rationale and Implementation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA., June 1993.

[19] Neelakantan Sundaresan. A thread model for supporting task and data parallelism in object-oriented parallel languages. Presentation at the Parallel Object-Oriented Machines and Applications Workshop, POOMA 94, 1994.

[20] Neelakantan Sundaresan and Dennis Gannon. Implementation of Ropes: An Aggregate Thread Class Library for Parallelism. Indiana University, Computer Science Department, Technical Report. In preparation, 1995.

[21] Neelakantan Sundaresan and Linda Lee. A Thread-based Object-Oriented Expansible Library Model for Parallelism:Abstract. Poster presentation at the Scalable High-Performance Computing Conference, Knoxville, Tennessee, May 1994.

[22] 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.