Experimental Evaluation of *Coir* : A System for Control and 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

Though data-parallelism addresses a wide variety of problems in scientific computing, this model in inadequate for general adaptive applications. *Coir* is our object-oriented thread-based model for data and task parallelism [25, 22, 24]. In this paper we briefly discuss the *Coir* objects - thread objects for control-parallelism and groups of thread objects, called ropes, for data-parallelism. Using an example from the CMU-task-parallel suite[5], we discuss the effectiveness of our model in terms of expressibility and efficiency.

keywords: Software, Object-Oriented Programming, Task and Data Parallelism, Evaluation, Scheduling

1 Introduction

User-level light-weight non-preemptible threads provide a good way of expressing asynchronous parallelism at a level closer to the logic of the user algorithm. Threads are to processes as processes are to physical processors. The advantages of threads over processes is that they can be created and sched-

uled more dynamically and in much larger numbers than processes because no system intervention is required and no system information is saved on context-switch. Even on a single processor system, threads provide for coroutine-style programming to express control separation. 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 ([13, 16]) is an emerging standard of thread interface at C language level. There are a variety of vendor-supplied thread libraries(SGI sprocs, SUN lwps, DCE threads etc.) and others from the university communities[6, 4, 17, 11, 2].

In *Coir*, threads objects depict control parallelism and rope objects, which are groups of thread objects in cooperative computation and communication, abstract data parallelism. Since *Coir* is object-oriented, these objects can be customized to specific application target by using object-oriented features like inheritance, polymorphism, dynamic dispatch and parameterized data types. In this paper we discuss how this library can be used to map a task and data-parallel application effectively on a shared-memory machine. For more details on the design and interface and portability across shared and distributed-memory machines, refer to [25, 22, 24]. The library provides a target model and runtime system for parallel extensions to C++ like pC++[1].

The paper is organized as follows. In the following two sections we briefly discuss control and data parallelism in terms of thread and rope objects. Then we introduce the narrow-band tracking radar problem from the CMU-task-suite library and discuss how to apply our model to this problem. This is followed by experimental results and discussion on the performance. Then we mention related work and then conclude with proposals for future work.

2 Threads for Control-Parallelism

Light-weight non-preemptible user-level threads can be used to express control separation within an otherwise sequential program. A thread which cannot continue computation because it needs some data to arrive or some condition to be satisfied may suspend itself and let another eligible thread of computation proceed. Thus computation and communication can be overlapped. This is feasible because the context-switch cost is low. Threads communicate with each other using shared memory locks - either directly or using a message-passing paradigm.

3 Threads and Data-Parallelism

On MIMD computers data-parallelism is typically implemented at the processor(process)-level. During a data-parallel operation a subset of the set of all available processors participate in it and processors which do not participate idle. Also processors which arrive at a synchronization point idle waiting for others to arrive. By providing data-parallelism at the thread-level, these idle cycles can be used to run other eligible threads. Since the context-switch time is low, this is feasible. Further, abstracting data-parallelism at a level higher than that of physical processors provides for separation and definition of control based on data structures. Using object-oriented paradigms active data-parallel objects can be defined where each object encapsulates data associated with its computation and provides abstractions for control in the computation. We call these data-parallel active objects as *ropes*.

A rope is a set of threads which execute in a cooperative manner over a set of processors. Ropes provide light-weight abstractions for data-parallelism. A rope defines a unique naming or scoping mechanism for the threads in it. Threads in a rope enter a data-parallel operation together and barrier-synchronize at the end of it. Threads may synchronize during the data-parallel operation.

The synchronization may be just a barrier, broadcast or reduction which may be specific to a particular data-type [23]. Further, the threads in the rope may communicate between themselves or with independent threads or with threads in other ropes. Communication between threads in two ropes is abstracted by a rope→rope communication.

4 Machine and Memory Model

A parallel machine can be abstracted based on memory and processor hierarchy. The degree of freedom of a thread is defined by a *processor-context*(PC), which defines the set of physical processors with a set of registers and a program counter and sharing an address space over which the threads migrate. PCs share program globals and a heap within the same shared-memory machine.

A machine may be partitioned into subsets consisting of one or more PC called *domain-subsets* which define the processor-topology over which a rope(ie. threads in a rope) are distributed. The set of all the processors in the machine that are available to the application define the *default domain.* Two ropes may be distributed over the same domain-subset, or overlapping domain-subsets or completely disjoint domain-subsets. A data-parallel operation on a rope over a domain-subset can execute independently of another data-parallel operation on another rope over another or the same domain-subset. Thus both task and data-parallelism are supported.

5 Example: Narrow-band Tracking Radar

We study the application of our model of parallelism to the narrow-band tracking radar problem in the task-parallel suite available from CMU [5, 21]. The suite program was in Fortran 77 and we translated it to C++ and to our programming model.

The narrow-band tracking radar benchmark is used to measure the effectiveness of various multi-computers for radar applications. The problem is interesting for studying combinations of task and data parallelism because of its hard real-time requirements, and because the input data-set is limited by the physical properties of the radar sensor.

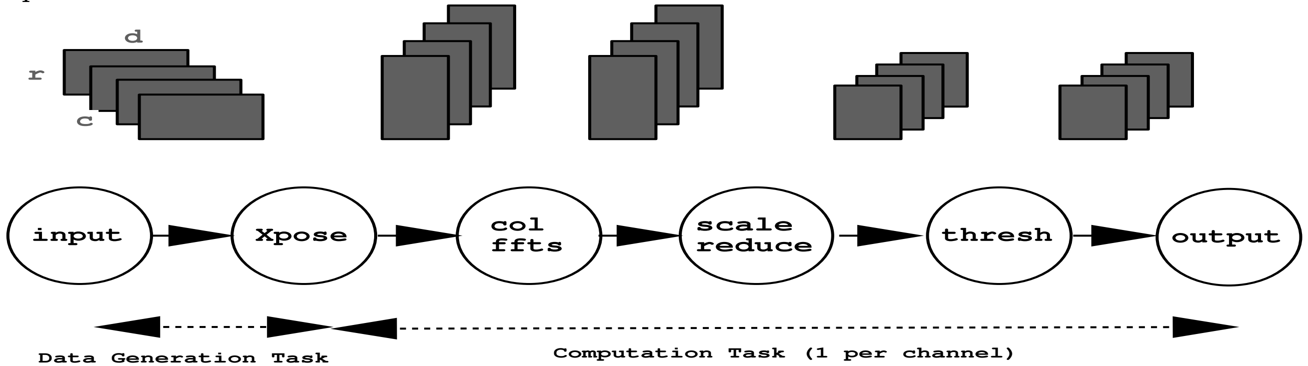
The program inputs data from a single sensor along $C = 4$ independent *channels*. Every 5 milliseconds, for each channel, the program receives $d = 512$ complex vectors of length $r = 10$, one after the other in the form of an $r \times d$ complex array A. At a high-level, each input array A is processed in the following way.

1. Transpose the $r \times d$ input array to form a $d \times r$ array.
2. Perform r independent d-point fast-fourier transforms(FFTs).
3. Convert the resulting complex $d \times r$ array into a real $w \times r$ subarray, $w = 40$, by replacing each element $a + ib$ in the $w \times r$ subarray with its scaled magnitude $\sqrt{(\frac{a^2+b^2}{d})}$.
4. Threshold each element a_{jk} of the subarray using a cutoff that is a function of a_{jk} and the sum of the subarray elements. Elements that are above the threshold are set to unity; elements below the threshold are set to zero.

5.1 Available Parallelism

Figure 1 shows the different phases in the radar algorithm. The second phase(*transpose*) basically transposes the data and is communication-intensive. The third, fourth and fifth phases, ie. *fft*, *scale-reduce* and *threshold* are purely data-parallel. The challenging part of parallelizing this application is that, the values *r, d, c* and *w* are not up to the user and cannot be increased. Data-parallelism is limited and hence one has to exploit the higher-level task parallelism. The data sets are independent so the algorithm can execute for multiple data sets through replication.

Figure 1: **Different stages between data generation and computation** on each of the c channels. Each channel has work for on data of size $r \times d$. Each of the computation stages are data-parallel. Scale-reduce is the only phase after transpose where the processors need to communicate for the reduction operations.



5.2 Applying our Model

The sensor is attached to only one processor, and data-generation task runs on that processor. For every set of data, there are two kinds of tasks - the data-generation task and the computation task for each of the channels which includes the functions for transposing, fft, scaling, reduction and thresholding. Based on the number of processors available and the granularity of the parallelism in the execution, one or more ropes, distributed over a subset of the set of available processors, may be defined where each rope is responsible for one channel [1] Operations on each channel are data-parallel and the rope-sizes are based on the data-sizes.

Since the ropes can have overlapping domain-subsets, two different ropes working on different channel data can be mapped onto the same set of processors. The motivation behind defining such overlapping domain-subsets is that when some of the threads in a rope are waiting for others to arrive at the synchronization step for, say, reduction before the *threshold* stage, threads from other ropes can be run in their place.

[1] we could have 1 rope working on more than 1 channel but for these set of experiments we fixed one rope to a channel.

Figure 2: **Timings in milli-seconds per iteration for 1-channel case**, for varying number of processors, with one processor-per-PC and rope distribution of 1 thread-per-processor. **Observation:** The size of the problem is such that data-parallelism can be exploited reasonably only up to 4 processors

# of processors	1	2	4	8
time(msecs)	18.6	10.05	9.2	14.0

Experiments

Each experiment consists of 10 iterations(10 sets of data for each of the 4 channels). The processors in the machine can be divided up in different ways by assigning one or more processors per PC. For instance, an 8-processor partition on a shared memory machine can be divided up as, (a) 1 PC with 8 processors or (b) 2 PC s with 4 processors per PC or (c) 4 PC s with 2 processors in each PC or (d) 8 PC s with 1 processor per PC and so on.

Our experiments were conducted on 8 processor partitions(with an additional processor for data-generation). Figure 2 shows the timing for only one channel with number of threads per rope = number of processors, and 1 processor per PC. The results show that there is not much data-parallelism that can be exploited beyond four processors.

Figure 3 shows the performance for varying the rope sizes(1, 2, 4, 8) with 1 PC with different number of processors(1, 2, 4, 8) in it. All the four channel ropes are mapped onto the same domain-subset. It can be seen that when there is only one processor, increasing the rope sizes marginally increases the time taken. Increasing the number of processors improves performance for up to 4 processors.

On comparing row 3(#processors = 4), column 3(rope size = 4) of figure 3 with column 3(#processors = 4) on figure 2, we see that the former is less than 4 times the latter, showing the advantage of scheduling multiple independent task on the same set of processors and importance of grouping processors appropriately. This, we believe, is because of the fact that when a thread in one of the

Figure 3: **Performance(time in millisecond for 1 iteration) for 4 channels, for case of 1 processor-context in the machine**, varying the number of processors and the rope sizes. All ropes are mapped onto the same set of processors. **Observation:** beyond 4 processors, the unrelated threads interfere affecting performance.

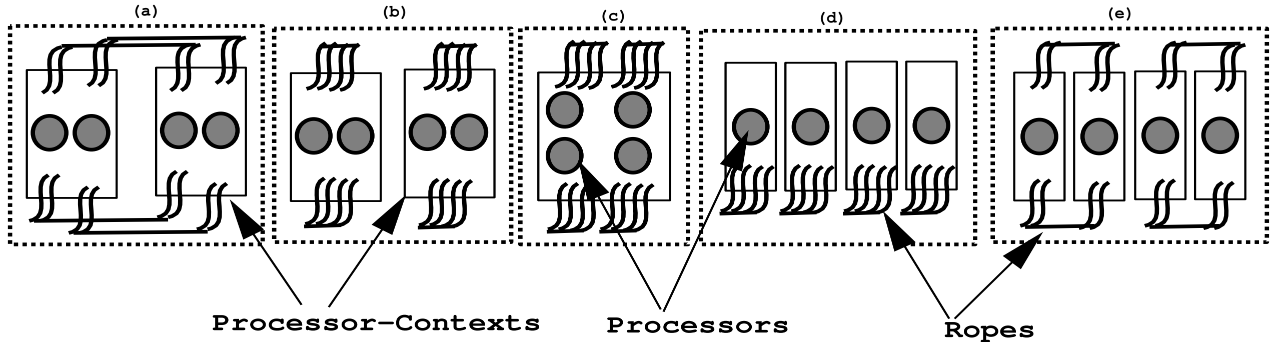
# of processors	rope sizes			
	1	2	4	8
1	52.8	53.7	54.07	57.6
2	36.2	31.6	34.07	36.1
4	23.1	22.3	21.3	25.6
8	39.6	36.7	39.8	50.4

ropes enters a synchronization(in this example, reduction), a thread in the other rope can execute in its place. Also notice that when the rope is of size 1(row 1 in the table), there is no synchronization and hence overlapping the ropes onto the same domain-subset has no advantage.

Figure 5 is divided into 4 sub-tables for different number of processors in the partition. Each sub-table shows the performance for various combinations of grouping the available processors into PCs; various ways of overlapping the rope domain-subsets; and different rope-sizes 1, 2, 4 and 8. Column 3 in the table indicates the *overlap* of the ropes. N ropes mapped onto the same set of processors cause an overlap of size N. An entry in the column is **none** when all the 4 ropes are mapped to disjoint set of processors; **all** means that all the 4 ropes are mapped to the same set of processors and an entry **2** means 2 rope are mapped onto one half of the processors available and the remaining 2 ropes onto the other half. Columns 1, 2 and 3 together determine the domain-subset over which a rope is distributed. Figure 4 shows some examples of this.

From figure 5 it can be seen that for any configuration of the processors and a fixed rope-size, performance improves as we increase the number of processors. Looking at any subtable(fixed number of processors) it is clear that just partitioning the set of processors and distributing the threads on them differently, different performances are achieved. For instance, in the last subtable (# processors = 4), the time taken goes from the best(13.36 msec: line 4) to three times as bad(46.3 msec: line 9)

Figure 4: **Examples of Rope Overlap:** A machine with 4 processors is divided up in different ways and 4 ropes each of size 4 are distributed over these processors. (a) 2 PC s with 2 processors each; ropes distributed over both the PC s(overlap = **all**) (b) Same as (a) except that there are 2 ropes per PC (overlap = **2**) (c) 1 PC with 4 processors; all ropes on this PC (overlap = **all**) (e) 4 PCs, 1 processor per PC, 1 rope per PC (overlap = **none**) (e) 4 PC s, 1 processor per PC, two ropes for every 2 PC s (overlap =**2**)



for the same rope-sizes. Comparing line 1 of subtable for 8 processors with line 2 of subtable for 4 processors case, we can see that overlapping 2 ropes on a domain-subset, takes less than twice the amount of time taken to execute a single rope on that domain-subset.

If the rope size does not exactly divide the number of rows some threads work on more rows than others. Thus, some threads may arrive at the reduction-barrier 'earlier' than others. Such situations are prime candidates for overlapping ropes. We noticed that, overlapping ropes in such a way that the overall work done on all the PCs are comparable gives better performance than arbitrarily distributing the ropes.

6 Related Work

A number of research efforts have been undertaken for supporting task and data parallelism paradigms in higher-level languages. Fortran-90 and HPF[7] are purely data-parallel. The Fx Fortran system[20], adds annotations to HPF to support task parallelism. Here, task parallelism is encapsulated by *begin*

Figure 5: **Performance with different ways of grouping subsets of an 8-processor partition** on the SGI/power-challenge. Four sub-tables, one each for partitions of size 1, 2, 4 and 8 processors are shown. Column 2 and 3 indicate number of PC s and number of processors per PC(note:col 2 $\times$ col 3 = col 1). Column 4 shows overlaps(see figure 4 for examples). Note that column 5 depends on column 4 and columns 2 and 3. **Observation:** Overlapping domain-subsets of the ropes takes less than twice the time taken for just one rope running on the same domain-subset; the best performance is obtained when there are 2 processors per PC, rope-size of 4 and no rope domain-subset overlaps.

# of procs	# of PC s	# procs per PC	overlap	rope distribution (#PC s, #procs per PC)	time(msec per iter) for rope sizes 1	2	4	8
1	1	1	all	(1,1)	52.9	55.2		
2	2	1	2	(1,1)	28.6	29.2	29.1	31.2
	2	1	all	(2,1)		33.6	35.4	36.6
	1	2	all	(1,2)	33.9	32.5	34.1	35.5
4	4	1	none	(1,1)	17.6	18.9	18.9	19.5
	4	1	2	(2, 1)		19.6	21.3	22.4
	4	1	all	(4, 1)			29.9	32.9
	2	2	2	(1,2)	18.8	19.5	20.8	22.1
	2	2	all	(2,2)		24.9	22.2	25.8
	1	4	all	(1,4)	23.3	22.2	23.5	26.6
8	8	1	none	(2,1)		14.6	15.1	16.4
	8	1	2	(4,1)			21.4	24.1
	8	1	all	(8,1)				40.2
	4	2	none	(1,2)	17.4	13.8	13.36	15.6
	4	2	2	(2,2)		19.07	19.4	19.3
	4	2	all	(1,2)			25.0	26.9
	2	4	2	(1,4)	28.3	25.5	20.9	38.5
	2	4	all	(2,4)		26.3	30.4	34.1
	1	8	all	(1,8)	42.1	34.8	46.3	57.1

parallel and *end parallel* annotations. The body of this encapsulation can contain only loops and subroutine calls. Assigning processors to the task is done using *processor* and *origin* directives. Since the model of parallelism is processor-based, different co-executing tasks cannot be scheduled on the same set of processors. Fortran-M[8] adds task-parallel extensions to Fortran-77. Here, a process models a task and concurrent processes communicate through message-passing. The runtime system of Fortran-M is based on a thread system called Nexus[9]. Message-passing is done over channels rather than between threads. A process do-loop is analogous to ropes in *Coir*. There are constructs to partition a machine using the *submachine* construct and distribute the processes. Since their

runtime system is also thread-based they do support overlapping *submachines*. CC++[3] is a task and data parallel syntax extension to C++, which is based on the same runtime as Fortran-M. The Opus [15] runtime system is another similar effort to support task parallelism in HPF and is based on a communicating pthread-based library called *Chant*[12] for distributed memory machines. A number of other parallel languages based on C or object-oriented languages like C++ or Eiffel address task parallelism; pSather[18](which is based on a monitor-model for NUMA machines and supports Eiffel-like syntax), Mentat[10](which is based on a coarse grain data-driven model and heavy-weight threads, with extensions to C++), Jade with shared-object abstractions[19]. Charm++[14](which is based on message-driven model, with C++ extensions) are a few of them.

7 Conclusions

We have proposed and implemented a model for supporting task and data parallelism in object-oriented parallel languages using light-weight threads. In this paper, we showed how, using threads, data-parallel constructs can be designed independent of the underlying processor structure. Our C++ library-based model, *Coir*, forms the substrate for building parallel object-oriented languages. Our machine model subsumes parallel machines with memory and processor hierarchy and shared and message-passing paradigms of communication. We showed how a parallel machine may be partitioned for (data-parallel) tasks and how independent tasks can proceed on the same processor in independent threads of control. Through an example we saw how the system can be effectively used to exploit task and data-parallelism. Our initial results are still not good enough to solve the actual radar application(which expects a performance of under 5 msec per iteration and our best is 13.36 msec). The program spends most of its time in the fft computation, which can be improved significantly. Next we plan to use this paradigm to extend pC++, which is currently a purely data-parallel collection-

based language, for task parallelism and for collection-parallelism which is detached from processor-level parallelism. Since the processing elements in *Coir* are dynamic, it will allow the compiling system make smart decisions on how to distribute the ropes and whether or not to overlap the domain-subsets and how to group and regroup processors into processor-contexts.

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] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, James Stichnoth, Jaspal Subhlok, John Webb, and Yand Bwolen. The CMU Task-Parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie-Mellon University, Pittsburg, PA., March 1994.

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

[7] High Performance Fortran Forum. High Performance Fotran Language Specification, Version 1.0. May 1993.

[8] Ian Foster and Mani Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992, Argonne National Laboratory, June 1992.

[9] Ian Foster, John Garnett, and Steven Tuecke. Nexus User's Guide, Version 2.0. August 1994.

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

[11] 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.

[12] 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.

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

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

[15] Piyush Mehrotra and Matthew Haines. An Overview of the Opus Language and Runtime System. In *Proceedings of 7th LCPC Workshop, Ithaca, NY*, August 1994.

[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] M Rinard, D Scales, and M Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.

[20] Jaspal Subhlok, David O'Hallaron, and Thomas Gross. Task Parallel Programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science, Carnegie-Mellon University, Pittsburg, PA., August 1994.

[21] Jaspal Subhlok, David O'Hallaron, Thomas Gross, Peter Dinda, and John Webb. Communication and Memory Requirements as the Basis for Mapping Task and Data-Parallel Programs. In *Proceedings of Supercomputing '94, Washington D.C.*, November 1994.

[22] 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*, August 1995. to appear.

[23] Neelakantan Sundaresan and Dennis Gannon. *Aggregate Thread Synchronization Operations on Shared and Distributed Memory Machines.* Technical Report 430, Indiana University, Computer Science Department, Bloomington, IN, 1995.

[24] Neelakantan Sundaresan and Dennis Gannon. *Coir: A Thread Model for Supporting Task and Data Parallelism in Object-Oriented Parallel Languages.* Technical Report 429, Indiana University, Computer Science Department, Bloomington, IN, 1995.

[25] 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.