Appears in the Proceedings of IPPS'96.

Optimizing COOP Languages: Study of a Protein Dynamics Program

Xingbin Zhang, Vijay Karamcheti, Tony Ng, and Andrew A. Chien
Department of Computer Science, University of Illinois at Urbana-Champaign
{*zhang,vijayk,achien*}*@cs.uiuc.edu, ng@lcs.mit.edu*

Abstract

Fine-grained concurrent object-oriented programming (COOP) models can simplify the programming of irregular parallel applications but are often perceived as inefficient. In this paper, we study implementation techniques to obtain efficient parallel execution of fine-grained COOP languages using a medium-sized protein dynamics program. We found that even with high data locality and good sequential code efficiency, an implementation using only thread-oriented optimizations and software multithreading fails to achieve parallel efficiency. The two major sources of overhead are the lack of processor-level data reuse and fine-grained threads for remote object accesses. Two processor-oriented optimizations, processor-level caching and communication grouping, overcome these inefficiencies and achieve performance comparable to a highly-tuned SPMD program.

1. Introduction

Irregular applications are challenging to program on distributed memory machines. They often exhibit irregular parallelism and require sophisticated distributed data structures. Fine-grained concurrent object-oriented languages that provide a shared namespace, object-level concurrency and implicit dynamic thread creation can simplify the programming of irregular applications. Irregular parallelism can be naturally expressed as implicit, concurrent threads operating on objects. The shared namespace and object-level concurrency control free the programmer from explicit name management and simplify synchronization. Unfortunately, fine-grained COOP languages have not gained wide-spread use because they are often perceived as inefficient.

In this paper, we study the implementation techniques required to obtain efficient parallel execution of a protein molecular dynamics code, IC-CEDAR, in a fine-grained COOP language on distributed memory machines. IC-CEDAR is a medium-sized application implemented using the Illinois Concert system.[1] IC-CEDAR is illustrative because although it is computationally intensive and has ample concurrency, the major phases exhibit irregular computation and data access patterns. Our goal is to achieve performance comparable to other highly-tuned implementations, such as Fortran or C with message passing, while preserving the natural expression of computation in Concert's high-level programming model.

We base our study on an implementation which

- embodies selected computation algorithms that give the best overall concurrency, locality and scaling properties for computationally intensive phases of the application,
- incorporates custom spatial partitioners of the problem input sets to obtain good locality, and
- is expressed in a style natural to the computation.

We found that even given high data locality and achieving good sequential code efficiency, an implementation that relies only on thread-oriented compiler and runtime optimizations and software multithreading fails to achieve parallel efficiency. Our initial implementation of IC-CEDAR is an order of magnitude slower than comparable Fortran implementations on parallel machines. This is discouraging since the Concert system already embodies most of the state-of-the-art optimizations that have traditionally been considered sufficient to obtain efficiency. Using three versions of IC-CEDAR representing different implementation techniques, we show that two major sources of overhead still remain. First, the lack of processor-level data reuse causes excessive processor communication and synchronization overhead. Second, fine-grained threads created for remote object accesses interleave with local computation, increasing both processor synchronization overhead and cache interference.

[1] Software distribution and a more detailed version of this paper are available from `http://www-csag.cs.uiuc.edu/`.

We demonstrate that two processor-oriented optimizations can overcome these inefficiencies and enable IC-CEDAR to achieve parallel performance comparable to a highly-tuned Fortran implementation. We believe these optimizations can be automated in a compiler and permit similar irregular applications to be expressed in a natural COOP style and achieve good performance.

2. COOP Model and Implementation

The Concert system supports a fine-grained COOP model based on Actors [1]. Computation is expressed as method invocations on objects or collections of objects. Method invocations conceptually operate within dynamically created threads that are inherently concurrent. In addition to common features of sequential object-oriented languages, three features of the programming model contribute fundamentally to programming irregular parallel applications:

- a shared name space,
- implicit dynamic thread creation, and
- object-level concurrency control.

A shared name space allows programmers to build sophisticated distributed data structures without explicit name management as well as decouples the algorithmic specification from data distribution. *Implicit dynamic thread creation* frees programmers from explicit thread management and synchronization and allows the irregular concurrency in the application to be expressed in a non-binding manner, leaving the implementation free to adapt the concurrency to the hardware parallelism. *Object-level concurrency control* maintains sequential consistency of the object state in the global namespace, enabling the safe composition of concurrent operations and freeing the programmer from managing explicit locking.

The Concert system [2] is a state-of-the-art implementation of a COOP programming model. On distributed memory machines, the execution model synthesizes a global namespace and hides latency of remote invocations via software multithreading on commercial single-threaded microprocessors. The Concert compiler [11, 13] implements a suite of interprocedural optimizations and can achieve sequential performance equal to that of C for a large number of programs including the Livermore kernels [10], a demanding numerical benchmark. The Concert runtime implements a flexible hybrid execution model [12], which creates threads lazily and exposes a hierarchy of high performance COOP primitives [8] to allow compile-time specialization.

3. IC-CEDAR

IC-CEDAR is a parallel version of CEDAR [6], a sequential protein molecular dynamics program. Computation proceeds in discrete time steps with four computation phases, shown in Figure 1. During each time step, forces exerted on each atom are calculated and used to update the atom's position and velocity. Although having ample concurrency, the application is challenging to parallelize because of irregular computation and communication patterns. We briefly describe the two major computation phases and challenging aspects of their implementation.

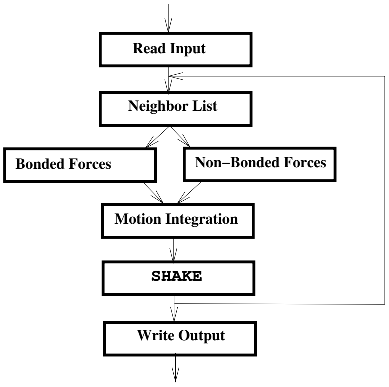


Figure 1. The basic algorithm of IC-CEDAR.

The neighbor list phase maintains a list of interacting (neighbor) atoms for each atom using a group-based cutoff to reduce the computational complexity of the all-pairs force computation. Although the multicell algorithm used relies on complex distributed data structures, it is naturally expressed in a COOP model as shown in Figure 2.[2] Because of the non-uniform spatial distribution of atoms and the large spatial volume examined for each atom, this phase exhibits irregular long-range data accesses and work distribution. Optimizing communication is especially important because the computation granularity is small.

The force calculation phase includes both short-range bonded forces and long-range Columbic and Lennard-Jones forces as shown in Figure 3. Again, work distribution across atoms is highly irregular, determined by the size of the neighbor list as well as the number of geometric constraints for each atom. The force calculation requires reading the position and updating the force of interacting atoms, also inducing irregular data accesses.

[2] All sample codes are shown using ICC++ [5], a parallel C++ dialect supported by the Concert system.

```
cells.doall(neighbor_list, cutoff); // fanout to all cells
...
void Cells::neighbor_list(float cutoff) {
  conc for (i = 0; i < num_neighbors; i++) {
    neighbor = neighbor_cells[i];
    conc for (j = 0; j < num_groups; j++)
      conc for (k = 0; k < neighbor.num_groups; k++) {
        group1 = overlapping_groups[j];
        group2 = neighbor.overlapping_groups[j];
        if( group1.within_cutoff(group2, cutoff) )
          group1.insert(group2);
      } } }
```

Figure 2. Sample neighbor list code.

```
conc {                // expressing task parallelism
  bondangles[0].doall(bonded); // bonded forces
  atoms[0].doall(nonbonded);
}
void Atom::nonbonded() {
  conc for ( j = 0; j < num_neighbors; j++) {
   // getxyz() returns multiple values.
   (x1, y1, z1) = neighbor_list[j].getxyz();
   ... compute using (x,y,z) and (x1,y1,z1), etc...
}
```

Figure 3. Sample force computation code.

4. Performance Optimizations

In this section, we discuss the implementation techniques required to achieve efficient execution of IC-CEDAR, assuming high data locality and preserving the natural expression of computation in the program.

4.1. Base Implementation

Our base implementation is representative of common approaches for optimizing fine-grained COOP languages. It consists of two components to obtain efficiency: a custom spatial partitioner that distributes objects to maximize spatial locality and a suite of state-of-the-art compiler and runtime optimizations to minimize overhead in the COOP model.

Data placement that exploits the spatial locality in both the neighbor list and force computation phases is mandatory for parallel efficiency. We apply space-filling curves, used in astrophysics applications [4], to distribute objects spatially as well as for dynamic load-balancing. In addition, a suite of locality-sensitive optimizations in the Concert system is applied to the base IC-CEDAR code. Speculative inlining and hybrid execution enable code optimizations for local execution. The majority of sequential COOP overhead is removed; the sequential CEDAR in C/Fortran is only 60% faster than IC-CEDAR on a single node.

Unfortunately, as a result of relying only on thread-oriented optimizations and software multithreading, the parallel performance of the base code is poor, achieving less than 5% parallel efficiency on 64 nodes. This is because most COOP implementations generally follow the logical thread structure imposed by the program and optimize for the performance of method invocations (logical threads). Although such thread-oriented optimizations achieve sequential efficiency, processor-oriented optimizations are required to achieve efficient parallel execution on distributed memory machines based on commodity microprocessors.

4.2. Processor-Oriented Optimizations

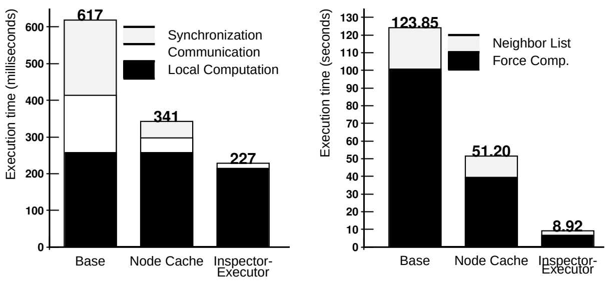
To study sources of remaining inefficiency in detail, we simplified the nonbonded force calculation into a representative computation kernel and use it as an example throughout our discussion. We instrumented the force kernel on 16 nodes of the CRAY T3D to obtain an execution time breakdown among local computation, processor communication, and synchronization overhead. Communication overhead includes processor involvement in sending and receiving messages, and synchronization overhead includes the time for thread creation and blocking (context switching), as shown by the *Base* bar in Figure 4(a).

4.2.1 Exploiting Processor-Level Reuse

Lack of processor-level data reuse is the first major source of overhead in the base implementation. In the force kernel even with high spatial locality, 40% of interactions still involve remote atoms because each processor is assigned only a small fraction of the space. However, there are ample opportunities for node-level reuse: more than 85% of remote accesses are to objects previously accessed in the same time step.

Although multithreading effectively hides the communication and remote invocation latency, the large number of remote accesses incurs significant processor overhead for communication and synchronization. A remote object access involves sending and receiving two messages and a handler execution, costing nearly 12 microseconds of total processor overhead on the T3D even with a low overhead communication layer [9]. In addition, a remote invocation causes processor synchronization overhead when the current computation blocks, requiring a thread context switch. Without effective node-level reuse, the overhead of communication and synchronization limits the parallel efficiency of the force kernel to 40% on 16 nodes.

We first investigate a runtime-based technique to exploit processor-level reuse. Because efficient object caching which maintains object sequential consistency for fine-grained COOP languages is an open problem, we implemented a demand-driven user-level cache. Remote object accesses first lookup a per-node cache. On a cache miss, a request is issued, and the current computation blocks to allow other threads to proceed. The computation restarts after the data arrives. The *Node Cache* bar in Figure 4(a) shows the execution time breakdown of the force kernel using demand driven caching. Figure 4(b) compares the base IC-CEDAR version to the one with node-level caching on a myoglobin simulation on 64 nodes of the CRAY T3D. The figure shows that node-level caches significantly improved the performance in both the neighbor list and force calculation phases, resulting in a speedup of 2.4.



(a) Force kernel on 16 nodes.

(b) Neighbor and force phases on 64 nodes.

Figure 4. Execution time breakdown of three versions of the force kernel (left) and IC-CEDAR (right) on the CRAY T3D. The total times are shown on the top of each bar.

4.2.2 Separation of Communication and Computation

The remaining major overhead is the fine-grained threads created for remote object accesses. These threads not only incur processor synchronization overhead at creation and context switches but also interfere with the ongoing local computation in both the processor data and instruction cache. Figure 4(a) shows that processor synchronization overhead still constitutes around 13% of the execution time in the node-cache version of the force kernel.

We used the inspector/executor model [15], in which an inspector determines all remote data accesses and performs the communication prior to local computation, to completely separate communication from computation. For the force kernel, processor synchronization overhead is dramatically reduced by grouping the synchronization of all outstanding requests, as shown in Figure 4(a). The figure also shows significantly improved processor cache performance for both communication and local computation. Communication overhead is reduced by 50% compared with the node-cache version with identical number of messages and communication volume. Local computation time is also improved by 30%.

The inspector/executor style execution is applicable to IC-CEDAR because in all phases, communication can be decoupled from ongoing computation. For example, communication required in the force phase depends only on the neighbor lists. Furthermore, the underlying data structure is non-recursive, allowing an inspector with a fixed number of stages to be constructed. Figure 4(b) shows the performance of IC-CEDAR after manually applying the transformation. The separation of communication and computation has a dramatic impact on performance, achieving a speedup of 5.8 over the node-cache version. On the T3D, cache interference is exacerbated by the smallness of the L1 cache and the absence of an L2 cache, but in the future, we expect the effect to be more pronounced due to technology trends.

5. Final Performance

We present performance results of the final version of IC-CEDAR using a myoglobin data set consisting of 10,502 atoms (2946 water molecules). The program transformations discussed in Section 4.2 are applied manually at the source level, and the Concert system performs all remaining optimizations. Unless otherwise stated, all results are for a 25-step simulation, with one neighbor list construction (10Å cutoff and 1,845,523 nonbonded pairs).

5.1. Sequential and Parallel Performance

Table 1 shows the execution time breakdown of the myoglobin simulation with identical parameters for the sequential CEDAR in C/Fortran and a one-processor run of IC-CEDAR (parallel version) on a 50 MHz Sparcstation 20 with 128 MB of memory.[3] CEDAR is 75% faster than IC-CEDAR on one node, which we believe is mainly due to the differences in code quality between the Fortran and C++ compilers.

Tables 2 and 3 show the execution time breakdown for IC-CEDAR on the CRAY T3D and the TMC CM-5, respectively. Because IC-CEDAR requires too much

[3] The C portion of CEDAR is compiled with cc -O2, and the force kernels in Fortran are compiled with f77 -O2. The generated code of IC-CEDAR, which is in C++, is compiled using g++ -O2.

Computation Phase	CEDAR	IC-CEDAR
Neighbor list	21.65	39.03
Force calculation	195.84	351.48
Motion Integration Velocity Rescaling		4.59
SHAKE	17.00	14.68
Total	234.49	409.78

Table 1. CEDAR *vs.* IC-CEDAR, in seconds.

memory to run on a single node, and CEDAR is unavailable on the T3D, an accurate speedup measurement is difficult. We use the execution times of IC-CEDAR minus the communication time on the smallest possible machine configuration, 8 nodes on the T3D and 16 nodes on the CM5, for estimating scaled processor efficiency. Tables 2 and 3 show that the performance is good up to 64 processors. At 128 nodes, the speedup of IC-CEDAR levels off, mainly due to the moderate input size and communication overhead. We are currently pursuing the use of bulk data transfer to improve the parallel efficiency at large machine configurations.

Computation Phase	Processors				
	8	16	32	64	128
Neighbor List	8.55	5.47	3.81	2.53	2.19
Force Calculation	33.89	20.03	10.13	6.39	5.09
Motion Integration Velocity Rescaling	0.91	0.59	0.33	0.30	0.47
SHAKE	3.43	1.83	1.38	1.32	1.37
Total	46.78	27.92	15.65	10.54	9.12
Estimated Efficiency	91%	76%	68%	50%	29%

Table 2. T3D execution times in seconds.

Computation Phase	Processors			
	16	32	64	128
Neighbor List	8.76	6.12	4.59	3.51
Force Calculation	122.84	64.64	38.02	33.71
Motion Integration Velocity Rescaling	1.26	0.81	0.51	0.59
SHAKE	4.74	3.65	3.00	2.94
Total	137.60	75.22	46.12	40.75
Estimated Efficiency	73%	67%	54%	31%

Table 3. CM-5 execution times in seconds.

5.2. Comparison to Other Implementations

Finally, to place our performance results in context, Table 4 compares the overall execution times of IC-CEDAR and CHARMM, a similar protein MD package [7], on 64 nodes of the CRAY T3D. CHARMM[4] is a highly-tuned SPMD implementation in Fortran with calls to the CHAOS [15] runtime library for communication and load-balancing. To reduce the differences in the data sets, we increased the nonbonded cutoff for the IC-CEDAR input. Because the two codes use different simulation models, it is difficult to make a direct comparison. To the first order approximation, however, both codes exhibit similar computation and communication behavior, and at 64 nodes on the T3D, IC-CEDAR achieves comparable absolute performance, within a factor of 2 of CHARMM.

	IC-CEDAR	CHARMM
Overall Execution Time	27.28 sec	18.46 sec
Number of Atoms	10,502	14,026
Nonbonded Cutoff	15 Å	14 Å
Nonbonded Pairs	6.1 million	6.7 million

Table 4. IC-CEDAR *vs.* CHARMM on the T3D.

6. Discussion and Related Work

Using IC-CEDAR, we demonstrated that although thread-oriented optimizations for COOP languages can produce good sequential efficiency, they are insufficient to achieve parallel efficiency. On distributed memory machines based on commodity microprocessors, software multithreading can incur large processor overheads for communication and synchronization. We showed that aggressive processor-oriented optimizations, node-level caching and communication grouping, are required for parallel performance.

While these results are based on IC-CEDAR, we believe they are applicable to other particle-based applications, such as molecular dynamics or particle-in-cell applications, which all exhibit the following key characteristics:

- major computation phases exhibit irregular long-range data access patterns, which are independent of the ongoing computation,
- underlying data structures are not recursive; the reference depths of objects are fixed and shallow, and
- dynamic load balancing can happen infrequently, enabling a quasi-static work distribution.

We believe for these applications, compile-time global program transformations can achieve efficient node-level reuse and communication grouping for COOP languages. The inspector/executor execution model has been extensively studied for SPMD programs [15, 7] and has been shown to give good performance for irregular applications whose data access pattern is independent of the ongoing computation. Ongoing efforts to

[4]The CHARMM execution time, which we scaled down appropriately to get an identical number of time steps and neighbor list updates as used in IC-CEDAR, is from `http://www.ki.si/parallel.html`.

compile such applications in parallel Fortran using this model share many issues with the required transformations here. Compiler-directed node-level reuse and communication grouping for COOP programs, in the presence of a shared namespace and inherently concurrent threads, requires additional global data access and concurrency information. We are currently developing a compiler framework using interprocedural analysis coupled with user annotations to automate the optimizations described here.

Our work is related to several dynamic programming language systems [16, 14, 3] whose objective is to achieve high performance on distributed memory systems. Implementations of these systems have traditionally focused on communication overhead reductions and thread-oriented optimizations, relying on user-specified data distribution to reduce communication volume. As our results show, these techniques are insufficient to achieve good parallel performance for applications similar to IC-CEDAR, when the code is expressed in a natural style. The processor-oriented optimizations we have described in this paper are likely to benefit these systems as well.

7. Conclusion and Future Work

In this paper, we studied the implementation techniques required to obtain efficient parallel execution of an irregular application, IC-CEDAR, written in a natural COOP style. Even with good data placement, we showed that current thread-oriented optimizations for COOP languages coupled with software multithreading are insufficient to obtain parallel efficiency on commercial distributed memory machines. Processor-oriented techniques to effectively exploit node-level reuse and group communication are required and sufficient for IC-CEDAR to achieve parallel performance comparable to a highly-tuned SPMD program. We are currently developing a compiler framework to automate these optimizations and studying runtime object caching and function migration techniques for more dynamic applications where the data access pattern is intrinsically coupled with ongoing computation.

Acknowledgments

We thank Julian Dolby, Scott Pakin, and John Plevyak for their careful reviews of the paper and the anonymous reviewers for their useful comments.

The research described here was supported in part by NSF grants CCR-92-09336, MIP-92-23732 and CDA-94-01124, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and a special-purpose grant from the AT&T Foundation. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Xingbin Zhang is supported by the CSE program of the University of Illinois. The experiments were conducted on the CM-5 at the National Center for Supercomputing Applications and the T3D at the Pittsburgh Supercomputing Center.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] A. A. Chien and J. Dolby. The Illinois Concert system: A problem-solving environment for irregular applications. In *the Symposium on Parallel Computation and Problem Solving Environments.*, 1994.

[3] D. Culler, et. al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *ASPLOS-IV*, pages 164–75, 1991.

[4] A. Y. Grama, et. al. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. In *Supercomputing'94*, pages 439–448, Nov. 1994.

[5] CSAG. The ICC++ reference manual. Concurrent Systems Architecture Group Memo, June 1995.

[6] J. Hermans and M. Carson. Cedar documentation. Unpublished manual for CEDAR, 1985.

[7] Y.-S. Hwang, et. al. Parallelizing molecular dynamics programs for distributed memory machines. *IEEE Comp. Sci. and Engr.*, pages 18–29, Summer 1995.

[8] V. Karamcheti and A. Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, 1993.

[9] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *ISCA '95*, 1995.

[10] F. H. McMahon. The Livermore Fortran kernels: a computer test of the numerical performance range. UCRL-53745, LLNL, Livermore, California, 1986.

[11] J. Plevyak. *Optimization of Object-Oriented and Concurrent Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois. In Preparation.

[12] J. Plevyak, et. al. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing'95*, 1995.

[13] J. Plevyak, et. al. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of PoPL*, pages 311–321, January 1995.

[14] A. Rogers, et. al. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 1995.

[15] J. Saltz, et. al. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[16] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.