Evaluating High Level Parallel Programming Support for Irregular Applications in ICC++*

Andrew A. Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang

Department of Computer Science,
University of Illinois at Urbana-Champaign
concert@red-herring.cs.uiuc.edu

Abstract. Object-oriented techniques have been proffered as aids for managing complexity, enhancing reuse, and improving readability of irregular parallel applications. Using a suite of seven challenging irregular applications and the mature Illinois Concert system (a high-level concurrent object-oriented programming model backed by an aggressive implementation), we evaluate what programming efforts are required to achieve high performance that is comparable to the best reported for low-level programming means on large-scale parallel systems.
Our study indicates that procedure and computation granularity, namespace management, and low-level concurrency management can be fully automated for these applications. Decoupling these concerns makes managing the remaining fundamental concerns – data locality and load balance – much easier. These two require programmer input, which our system integrates cleanly into the programming interface. Overall, only small changes to the sequential code (fewer than 5% of the source lines) were required to express concurrency and performance optimizations.

1 Overview

Object-oriented techniques have been proffered as aids for managing complexity, enhancing reuse, and improving readability of parallel applications, and numerous variants have been explored (e.g. [14]). Such techniques appear most promising for irregular applications where modularity and encapsulation help manage complex data and parallelism structures. In this paper, we systematically evaluate a high level concurrent object system's programmability for achieving high performance on irregular applications. Prior research exists for both implementing irregular parallel applications efficiently and concurrent object-oriented programming, but we know of no studies that address both together.

* The research described in this paper was supported in part by DARPA#E313 through AFOSR Contract F30602-96-1-0286, ONR grants N00014-92-J-1961, N00014-93-1-1086, NASA grant NAG 1-613, and T3D resources at the Jet Propulsion Laboratory. Andrew Chien is supported in part by NSF YI Award CCR-94-57809. Vijay Karamcheti was supported in part by an IBM Computer Sciences Fellowship.

Of the research on programming irregular parallel applications, none provides a single programming interface while supporting all three dimensions of irregularity – data, control, and concurrency. Research efforts have largely focused on building libraries which provide runtime support for some dimensions of irregularity [6, 13, 9]. Most concurrent object-oriented programming systems have ignored support for fine-grained parallelism [8, 2, 5], which forces programmers to map irregular concurrency into grains large enough to achieve efficiency.

This study is the first systematic evaluation of programming effort for a significant range of irregular applications. Our application study is done in the context of the Illinois Concert system, a high performance compiler and runtime for parallel computers which has been the vehicle for extensive research on compiler optimization and runtime techniques over the past five years [3]. Using the mature Concert system, we evaluate programming effort for a rich suite of irregular parallel applications.

Our results indicate that a high-level concurrent object-oriented programming model and a sophisticated implementation can eliminate many concerns, easing the programming of irregular parallel applications. First, programmers are freed from explicit namespace management, computation granularity (procedure and data), and low-level memory race management. Second, while in most cases, programmers still needed to express application specific data locality and load balance, the orthogonal framework for functionality and data placement eased this effort by allowing radical changes in data layout with modest code changes. Finally, over the seven applications only modest code changes (less than 5% of the lines) were required, mostly to express the careful data locality and load balance needed to achieve top performance.

2 Background

In evaluating a programming approach, our goal is to identify the concerns which must be addressed by a programmer to achieve good performance and how those concerns are managed. Two high level concerns are of most interest — *concurrency and synchronization specification*, and *data locality and load balance*. The former is required to express the parallelism structure, while the latter ensures efficient execution. We describe interfaces in the Concert system which aid in the management of these two concerns.[1]

Basic Programming Interface Our programming interface is a basic object-oriented programming model augmented with simple extensions for concurrency. The basic model is based on C++ (ICC++ [14]), providing a single namespace, scalars and objects, and single inheritance. Simple extensions for concurrency include annotating standard blocks (i.e. compound statements) and loops with the `conc` keyword. A `conc` block defines a *partial* order amongst its statements, allowing non-binding concurrency while preserving local data dependences. A

[1] More information about Concert is available at `http://www-csag.cs.uiuc.edu/`

simple semantics for object concurrency provides synchronization — method invocations on an object execute as if they had exclusive access.

Performance Optimization Interfaces Data placement and colocation is managed through *collections* — distributed arrays of objects for which the user can specify standard layouts (e.g. block, cyclic) or full custom distributions. Between individual objects, colocation control can be expressed at object allocation time.

Three additional interfaces support control over data consistency, thread placement, and task ordering. Whereas the data placement interfaces were found useful in all applications, these additional interfaces are only required by a few of the applications. The data consistency interface allows the Concert system to use relaxed consistency models for higher performance. As shown in the following code fragment, the **with_local** annotation specifies a set of objects that need to be localized and provides information about read-only access semantics:

```
function( obj1, obj2, ... ) {
   with_local( obj1, READ_ONLY, obj2, READ_ONLY )
     { ... = func( obj1, obj2, ... ); }
}
```

The thread placement interface provides support for controlling data locality and load balance from the perspective of threads and is specified by annotating the thread creation using the **!target** keyword with an integer argument. In the following code fragment, the thread corresponding to `thread_function` is executed on a random processor.

```
( thread_function( arg1, arg2, ... ) !target rand() );
```

Task ordering is useful in applications based on search which prioritize promising tasks for higher efficiency. Task ordering is specified by annotating the thread creation site using the **!priority** keyword which takes the task priority as an argument. Additionally, the interface permits the programmer to specify a custom scheduler for thread execution. In the following code fragment, the thread corresponding to `thread_function` executes with priority `arg1` and is managed by a global priority scheduler:

```
( thread_function( arg1, ... ) !priority arg1 !scheduler PRIORITY );
```

Implementation Technology The Concert system contains a wide range of aggressive optimizations and has been used to demonstrate high performance in absolute terms on a wide range of applications [3]. It automatically addresses several concerns which programmers must manage in lower level programming models:

- Procedure Granularity and Virtual Function Call Overhead
- Compound Object Structuring
- Thread Granularity
- Distributed Name Management
- Initial data placement, Data consistency and sharing (sometimes)

3 Application Case Studies

We illustrate the benefits of high-level programming features and support using a suite of seven applications (see Table 1). Each problem is challenging to express in parallel, exhibiting irregularity in one or more of data structure, parallel control structure, and concurrency.

Applications	Description
Polyover [14]	Overlay of two polygon maps (computer graphics)
Barnes-hut [15]	Barnes-Hut hierarchical N-body method (computational cosmology)
FMM [15]	Fast multipole method (computational cosmology)
SAMR	Structured adaptive mesh refinement (computational fluid dynamics)
Gröbner [1]	Gröbner basis (symbolic algebra)
Phylogeny [7]	Evolutionary history of species (molecular biology)
Radiosity [15]	Hierarchical radiosity method (computer graphics)

Table 1. The Irregular Applications Suite

Although we have examined all seven applications, for space reasons we restrict our discussion here to three of them — FMM, Gröbner, and Radiosity.[2] For each application, we describe programmer effort associated with concurrency specification and ensuring data locality and load balance.

3.1 FMM

The FMM code computes the interactions of a 2-D system of particles using the fast multipole hierarchical N-body method. Our parallel algorithm is derived from the FMM application in the SPLASH-2 suite [15].

Concurrency and Synchronization Specification There are three levels of parallelism in the dominant force computation phase: across interactions corresponding to separate boxes, across computations with different interaction lists for a single box (each box has four lists), and across interactions with boxes within the same interaction list. They are naturally expressed with the `conc` annotations, as illustrated below. The global namespace and object consistency of ICC++ obviate additional code to access potentially remote boxes. In all, there are 126 lines (out of 3951 lines) in the code corresponding to concurrency specifications.

```
 conc for (i = 0; i < NumBoxes; i++)
  boxes[i].computeInteractions();
...
computeInteractions() {
  if (type == CHILDLESS) conc {
    computeSelfInteraction();
```

[2] See a longer version of this paper at `http://www-csag.cs.uiuc.edu/` for the rest

```
    conc for (i = 0; i < num_u_list; i++) UListInteraction(u_list[i]);
    conc for (i = 0; i < num_w_list; i++) WListInteraction(w_list[i]);
  }
  conc for (i = 0; i < num_x_list; i++) XListInteraction(x_list[i]);
  conc for (i = 0; i < num_v_list; i++) VListInteraction(v_list[i]);
}
```

Data Locality and Load Balance FMM exhibits both data-dependent irregular data access patterns and computation granularity. The interaction lists of a box depend on its spatial coordinates, producing irregular data accesses. In addition, the lengths of interaction lists are highly irregular across boxes, making both data locality and load balancing essential for good performance.

We ensure data locality and load balance using a spatial partitioning of particles and boxes and a history-based load balancing algorithm, adopted from [10]. The spatial partitioning orders the particles by their position, and this ordering is blocked across the processors. Because the FMM method operates on boxes, particles within the same box are assigned to the same node, and boxes are assigned to the processor that owns a majority of its sub-boxes with ties broken arbitrarily. The history-based load-balancing counts the number of interactions for each particle, and balances the number of interactions across the processors. We use the data placement interface in the Concert system to achieve the assignment of particles to processors. Given the default owner-computes policy of Concert, separate code to distribute the tree handles both load-balancing and initial placement. The computation is decoupled from data placement, enabling easy experimentation with different policies. The data placement and load balancing code consists of 37 line changes (out of 3951 lines).

3.2 Gröbner

The Gröbner application computes the Gröbner basis of a set of polynomials. Our algorithm is derived from a parallel implementation by Chakrabarti [1].

Concurrency and Synchronization Specification Parallelism in the application arises from parallel evaluation of polynomial pairs, and is naturally specified using a `conc` loop around the pair generation code in the sequential program. Pair evaluation tasks that produce irreducible terms invoke a method on the basis object to augment it. Support of a global namespace in Concert ensures that potentially remote objects (either the polynomial or the basis) can be accessed with code identical to the sequential methods. The only synchronization specification required is that the basis object be kept consistent against concurrent access, trivially ensured in Concert due to the object-level concurrency control semantics. The natural expression of concurrency and synchronization specifications only require changes to 5 lines in the program (out of 1919 ICC++ lines).

Data Locality and Load Balance Pair evaluation work is data-dependent and varies with evaluation order. Consequently, careful management of data locality and load balance is required for efficiency.

Data locality ensures that the pair tasks have efficient access to the basis and its polynomials. Since load balancing requires pairs to be evaluated on arbitrary processors, both polynomial and basis objects are cached on demand, utilizing additional information about relaxed data consistency semantics. Programmer input is required for this information which conveys the fact that polynomials are accessed in a read-only fashion, and that the computation can proceed with an inconsistent view of the basis. The latter permits overlap of pair evaluations with basis augmentations, and is expressed as in the following code segment. Over the entire program, the programmer needs to specify access information about 2 kinds of views which requires changes to 4 lines in the program.

```
processPair( pOne, pTwo ) {
  spol = spol( pOne, pTwo );
  basisp = &(Basis);   /* can use an inconsistent Basis copy */
  with_local( basisp, READ_ASYNC )
    { ... = basisp->setReduce( spol ); }
}
```

Load balance is needed for pair evaluation tasks which also require prioritized execution to avoid redundant work. Our implementation attaches the pair evaluation task to a priority-based work-stealing scheduler (provided as part of the Concert system) as shown in the code fragment below. Programmer input is required to identify the task granularity (the `processPair` function), supply an integer priority using the `!priority` keyword, and specify a scheduler for enqueuing the task (the `GLOBAL_PRIORITY` scheduler). Similar custom load-balancing annotations were required to 2 lines in the program.

```
/* add pNew to Basis: generate work by iterating over Basis */
 conc for ( pOld = ... ) {
  if ( (pOld != pNew) && test(pOld, pNew) )
    ( processPair( pOld, pNew )  !priority calculate_priority(pOld,pNew)
        !scheduler GLOBAL_PRIORITY );
}
```

3.3 Radiosity

The radiosity method computes the global illumination in a scene. Our parallel algorithm is derived from the radiosity application in the SPLASH-2 suite [15].

Concurrency and Synchronization Specification There are three levels of parallelism in each iteration — across all input patches, across children of a subdivided patch, and across neighbor patches in the interaction list — which are all naturally expressed using `conc` annotations. Concert's global namespace enables access to remote objects with code identical to sequential methods. Two synchronization specifications ensure correct execution, simply expressed using the object-level concurrency control semantics. The first specifies that a child patch radiosity calculation must be nested within that of its parent. The second specifies exclusive access to the patch object. Over the entire program, these specifications are only required in 7 places (out of 3144 lines).

Data Locality and Load Balance This application exhibits data-dependent irregularity at each level of parallelism as well as across iterations. Consequently, careful management of data locality and load balance is required for efficiency.

Data locality is required for radiosity and visibility tasks. The former accesses the target patch, and for each interaction, the interaction object and the neighbor patch. The visibility task not only accesses a pair of patches, it also traverses the BSP tree in a data-dependent fashion. Locality of interaction objects and the BSP tree is ensured by data alignment: the former with the source patch and the latter by explicit replication. In contrast, patch locality requires demand-driven caching which is made efficient by utilizing relaxed consistency information. As the simplified visibility calculation code below shows, the programmer uses the `with_local` annotation to specify the read-only access semantics (a view) of both polygonal input patches. The programmer needs to specify similar access information about 3 kinds of views. Over the entire program, ensuring data locality increases the source code length by 33 lines (as compared to the original 3144 lines): of these the bulk (25) replicates the BSP structure, the rest describes alignment and data consistency specifications.

```
computeVisibility(spatch, dpatch, cur) {
  with_local( spatch, READ_ONLY, dpatch, READ_ONLY )
    { vis = vis_func( spatch->pos, dpatch->pos, ... ); }
  cur->vis = vis;
}
```

Load balance is primarily required to balance the radiosity and visibility tasks across the machine. Allocating dynamically created patches on random nodes at creation time (the default policy in Concert), and executing radiosity tasks local to them ensures that the latter is load balanced. However, load balancing visibility tasks requires a sender-initiated scheme. As the following code fragment shows, the programmer had to identify the task which must be balanced (`computeVisibility`) and its target destination (a random node), both of which are specified in Concert using the `!target` keyword. Similar annotations were required at 3 places in the program.

```
 conc for (cur=start_interaction_list; cur; cur = cur->next ) {
  if ( cur->vis == VISIBILITY_UNDEF )
    ( computeVisibility( this, cur->dest, cur )  !target rand() );
}
```

3.4 Summary

All of the applications use globally-shared pointer-based data structures and most exhibit irregular concurrency structures, which are expressed naturally using the global namespace, object consistency model and `conc` annotation of ICC++.

The application suite falls into two main classes. Polyover, Barnes-Hut, FMM, and SAMR exhibit irregular concurrency which requires either semi-static or random load balancing schemes. Programmer input is only required in the form of a small amount of code for load balancing and data placement, typically using collection maps. As a result, fewer than 5% of the lines typically require changes: 81 out of 1695 for Polyover, 107 out of 2631 lines for Barnes-Hut, 163 out of 3951 lines for FMM and 163 out of 6303 lines for SAMR.

On the other hand, Radiosity, Gröbner, and Phylogeny additionally require fully dynamic load balancing, application-specific scheduling and relaxed consistency models. Thus, additional effort is required from the programmer who must have intimate knowledge of the application to specify load balancing policies, thread priorities, and data-access semantics. However, Concert provides high-level annotations to allow these kinds of information to be incorporated into the computation with very little code change: 11 out of 1919 lines for Gröbner, 16 out of 2305 lines for Phylogeny, and 43 out of 3144 lines for Radiosity.

4 Parallel Performance

We discuss performance based on the Cray T3D implementation of the Concert system.[3] All programs achieve good sequential performance (within a factor of 2 of corresponding C programs), benefiting from implementation techniques that eliminate programmer management of concerns such as procedure and thread granularity, and object structuring, otherwise required for performance.

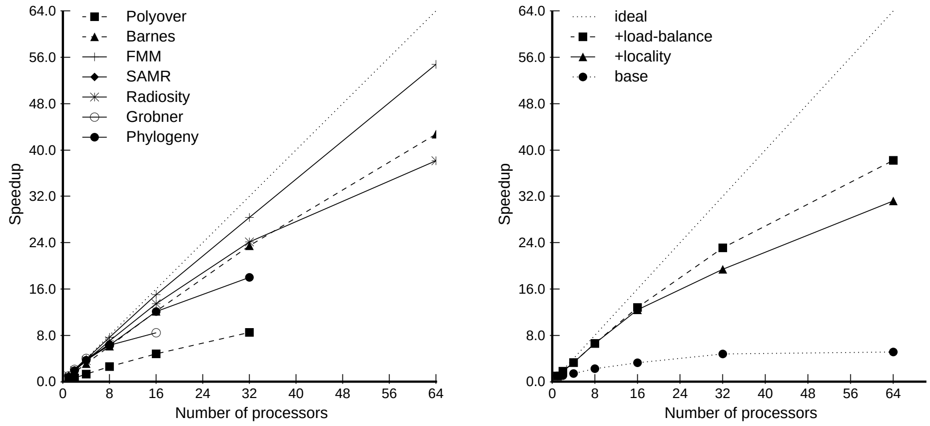

Fig. 1. Overall speedup of application suite (left) and incremental performance benefits from program transformations for the Radiosity application (right).

Figure 1(left) shows application speedups with respect to the non-overhead portion of the single node execution time. These speedups compare favorably with the best reported elsewhere for hand-optimized codes. Polyover's speedup

[3] Except for the SAMR code, whose performance results are on the SGI Origin 2000.

levels off because of insufficient work in the parallel algorithm but is comparable to other reported numbers [14]. Both the Barnes speedup of 42 and FMM speedup of 54 on 64 nodes are competitive with shared-memory versions [15] and hand-optimized versions [12]. The Radiosity speedup of 23 on 32 T3D processors compares well with a reported speedup of 26 on 32 processors of the DASH machine [10]. Speedups for both Gröbner and Phylogeny compare favorably with implementations that explicitly manage data locality [7, 1].

The above competitive speedups were made possible by transformations that enhance data locality and load balance. To illustrate their performance advantages, Figure 1(right) shows the speedup for three Radiosity versions: base, locality, and load-balance. As described in Section 3.3, both locality and load-balance versions are realizable with changes to less than 2.5% of the source code, yet improve speedup significantly.

5 Discussion and Related Work

Despite the wealth of parallel programming research, we know of few application-based studies that systematically evaluate the "programmability" delivered by a parallel language or system. To our knowledge this the the only study which looks at a collection of irregular applications on a general purpose parallel programming system to evaluate its support for high level expression and performance optimization. We find the outcome encouraging. In the Concert system, much of the difficulty of managing the irregular concurrency and modifying a program's data distribution has been eliminated. If commercial parallel programming environments can incorporate similar technologies the prospects for high performance through high level parallel object-oriented programming are encouraging.

Programming techniques for irregular applications is an important body of related work. All of these techniques pursue a library-based approach that considers irregularity in a single dimension and typically only support particular classes of algorithms (e.g., on sparse matrices or adaptive meshes [9]). While the best known parallel programming approaches — message passing [11] and data parallel [4] — can be used for programming irregular applications, neither provides adequate tools to manage the inherent complexity.

6 Summary

Based on our application studies, we must conclude that while the development of irregular parallel applications remains challenging, the Concert System enables such programs to be written with a high level approach. Although the complexity of data locality and load balance is significant (often comparable to the algorithmic complexity), the high level program expression enables their orthogonal expression improving program flexibility. Finally, the optimized parallel versions of the irregular applications had identical basic program structure to their sequential counterparts. Concurrency was added with small program perturbation, and data locality and load management was achieved with minor program changes (less than 5% of original source lines).

References

1. S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the Fourth ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 169–179, May 1993.
2. K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.
3. Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, April 1997.
4. High Performance Fortran Forum. High performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Rice University, January 1993.
5. A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.
6. J.H. Saltz, *et al.* A manual for the CHAOS runtime library. Technical Report CS-TK-3437, Department of Computer Science, University of Maryland, 1995.
7. Jeff A. Jones. Parallelizing the phylogeny problem. Master's thesis, Computer Science Division, University of California, Berkeley, December 1994.
8. J. Lee and D. Gannon. Object oriented parallel programming. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1991.
9. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, Sunriver, Oregon, April 1994.
10. Jaswinder Pal Singh. *Parallel Hierarchical N-Body Methods and Their Implications For Multiprocessors*. PhD thesis, Stanford University Department of Computer Science, Stanford, CA, February 1993.
11. Snir, et. al. *MPI: The Complete Reference*. MIT Press, 1995.
12. M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing Conference*, pages 12–21, 1993.
13. Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Katherine Yelick. Run-time support for portable distributed data structures. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 111–120, Boston, May 1995. Kluwer Academic Publishers.
14. Gregory V. Wilson and Paul Lu, editors. *Parallel Programming Using C++*. MIT Press, 1995.
15. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.