Innovative Language-Based & Object-Oriented Structured AMR using Fortran 90 and OpenMP

Dinshaw S. Balsara (NCSA) and Charles D. Norton (NASA-JPL)

New Trends in High Performance Computing (Award Submission)

Parallel adaptive mesh refinement (AMR) is an important numerical technique that leads to the efficient solution of many physical and engineering problems. While some AMR libraries have been designed, there are many advantages to considering alternative approaches based on language paradigms and standards. In this paper, we describe how AMR programming can be performed in an object-oriented way using the modern aspects of Fortran 90 combined with the parallelization features of OpenMP. This unique approach combines efficiency, portability, and maintainability for the application scientist that requires programming flexibility beyond the features a static library may provide.

Introduction

Adaptive methods are extremely useful in the solution of large scientific problems with complex geometry, but sophisticated programming and powerful computational resources are required. Since AMR is complicated, involving the manipulation of abstract structures like hierarchical distributed mesh components of varying resolution, scientists currently rely on libraries to hide the complexity of message passing on large distributed memory parallel systems. Designing an all-encompassing library, however, that is suitable for any kind of AMR application is extremely difficult—probably impossible. Some researchers are pursuing structured AMR library approaches using the C++ programming language [5, 6], but these have not yet demonstrated high performance, scalability, or popularity for a large class of AMR applications.

We introduce an exciting solution using language-based development that excels with the strengths of an SMP/ccNUMA environment and matches MPI performance in a message passing environment. Our approach, based entirely on well-defined standards, reduces programming complexity, preserves the investment in existing Fortran-based solvers, and benefits from years of compiler optimization techniques. Indeed, this is the first work that demonstrates a scalable, efficient, and complete approach to AMR that integrates emerging trends in high performance computing while returning control of software development to the user, rather than relying on the static features of a library. We illustrate this approach by applying it to Balsara's RIEMANN framework, see [1] and references therein.

Language-Based Design for Parallel AMR

Our approach combines the parallelizing directives of OpenMP with the Fortran 90 standard for structured AMR. We have developed efficient, parallel, and scalable methods for performing all of the tasks required. This includes creation and deletion of AMR hierarchies, processing of inter-grid transfers across/within levels, and the solution of these grids anywhere in the hierarchy in a load balanced, and parallel, way.

Object-Oriented AMR with Fortran 90

Introducing object-oriented programming techniques with the new features of Fortran 90 [3, 4] makes it possible build intricate AMR structures that are efficient. While the array-syntax

and dynamic memory management features are most familiar, new features including modules, derived-types (user defined types), use-association, generic interfaces, and (safe) pointers, simplify AMR data structure design.

Fortran 90 allows us to create, manage, and delete grid types that are used in the solution process. These grids can overlap and support parent, child, and sibling relationships across AMR levels along with the interpolation of boundary conditions. Collections of grids at a given level in the AMR hierarchy totally cover the regions that need refinement. Fortran 90 modules allow one to define specific features that can be applied to the grids, either as a collection or individually. Additional features useful for the solution process can be included in the module as well, and when used in main programs that allows objects to be created. State changes in the objects are limited to the routines that the module makes public. This object-oriented design allows all grid operations to be completely parallelized including the regridding strategies [2].

Using OpenMP for AMR Parallelization

The features of OpenMP that complete our approach are the directives that support data distribution, generation of threads for independent loops, and the *affinity* clause that allows one to support the "owner-computes" rule for efficient processing. We have also implemented a very efficient load-balancer to ensure that grid objects are created and processed in the hierarchy in a balanced, and parallel way. Figure 1 briefly shows the use of Fortran 90 object abstractions and the directives. The code segment illustrates how a series of dynamically defined grids is

```
type (single_grid), pointer :: this
integer, dimension(max_single_grids) :: array_for_affinity
!$SGI DISTRIBUTE array_for_affinity(cyclic(1))
!$OMP PARALLEL DO PRIVATE(igrid, this)
!$OMP& SHARED(level, grid_is_active, pointers_to_grids)
!$SGI+AFFINITY (igrid)=DATA (array_for_affinity(igrid))
    do igrid = 1, max_single_grids
         if (grid_is_active(level, igrid) == 1) then
            this => pointers_to_grids(level, igrid)%sgp
            call wrapper_solver_single_grid (this,...)
         end if
    end do
```

Figure 1: OpenMP/Fortran 90 object-oriented multi-grid parallel structured AMR.

processed at an AMR level, and how a Fortran 90 wrapper is used to call an existing Fortran 77 solver. The affinity clause, and the parallel do, ensure that processors work on the grids that they themselves own[1]. While not all the details are given here, this demonstrates that all steps associated with constructing hierarchical grids, managing their solution across various AMR levels, and supporting their load balance, can be accomplished based entirely on a parallel compiler language-based approach. A secondary benefit is that the code can run sequentially by simply ignoring the directives.

[1]Most of the arrays could be replaced with lists, but arrays are demonstrated for simplicity. Module and object definitions have also been omitted in this abstract.

# grids	round-robin binning	1st load imbalance	2nd load imbalance	3rd load imbalance	4th load imbalance	5th load imbalance	6th load imbalance
125	186.09	64.15	64.15	64.15	64.15	64.15	64.15
150	180.38	39.91	27.78	26.77	26.77	26.77	26.77
175	153.88	26.62	24.19	23.54	23.54	23.54	23.54
200	113.76	5.41	4.97	4.97	4.97	4.97	4.97
250	128.61	22.20	3.46	2.55	2.55	2.55	2.55
330	118.60	8.04	0.74	0.40	0.40	0.40	0.40
375	105.73	9.00	1.05	0.24	0.20	0.20	0.20
450	94.15	13.40	2.47	0.33	0.16	0.12	0.12
525	90.50	6.44	0.73	0.11	0.05	0.05	0.05
625	78.33	4.88	0.63	0.12	0.04	0.04	0.04
740	81.13	5.67	1.28	0.10	0.03	0.03	0.03

Table 1: Percentage load imbalance for successive iterations of the load balancer.

Language-Based, Dynamically Load Balanced, Parallel Performance

In AMR, where changes in computational work can only be estimated at run-time, applications require dynamic load balancing over each level in the AMR hierarchy. We have designed a specialized load balancer that is uniquely well-suited for AMR applications. The load balancer is iterative, and improves in quality with successive iterations. It utilizes a pairwise exchange of load assigned to available processors such that an exchange causes a maximal reduction in load imbalance between pairs of processors. The computational cost of the algorithm is low, and it can be parallelized easily.

Table 1 compares round-robin binning of tasks to our load balancer where 125 to 740 tasks are applied to 100 processors. There is a 300% difference between the minimum and the maximum load, which is assigned randomly. Note that the round-robin approach has a large percentage load imbalance, while our approach quickly reduces load imbalance to less than 1% in a small number of iterations. The load associated with updating a single grid is proportional to the number of computational zones on the grid. Table 2 shows the cumulative speedup on an Origin 2000 for processing an AMR level. The full paper will explore the language-driven structured AMR design and implementation along with performance enhancing schemes in substantially greater detail. We will also present applications to several interesting elliptic and hyperbolic systems.

# of Processors	1	2	4	8	16	32	64
Time (seconds)	27.52	10.32	5.27	2.74	1.40	0.77	0.42
Cumulative Speedup	1	2.66	5.21	10.03	19.60	35.60	64.75

Table 2: Performance results for scalability on processing an AMR level.

References

[1] D. Balsara *J. Quant. Spectroscopy and Rad. Transfer*, 62:167–180, 1999.

[2] M. Berger and I. Rigoutsos. *IEEE Trans. on System, Man, and Cybernetics*, 21:61–75, 1991.

[3] V. K. Decyk, C. D. Norton, and B. K. Szymanski. *Sci. Prog.*, 6(4):363–390, Winter 1997. IOS Press.

[4] C. D. Norton. In J. Schaeffer, ed., *High Perf. Comp. Sys. and Applications*, pg. 47–58. Kluwer, 1998.

[5] M. Parashar, et. al. In *Proc. SC'97*. IEEE Computer Society, Nov. 1997.

[6] D. Quinlan. In *Proc. IMA Workshop on Structured AMR*, Minneapolis, MN, March 1997.