Automatic Thread-Level Parallelization in the Chombo AMR Library

Matthias Christen[2], Noel Keen[1], Terry J. Ligocki[1],
Leonid Oliker[1], John Shalf[1], Brian Van Straalen[1], Samuel W. Williams[1]

[1]Computational Research Division (CRD)
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

[2]University of Basel, Switzerland

August 16, 2011

Abstract

The increasing on-chip parallelism has some substantial implications for HPC applications. Currently, hybrid programming models (typically MPI+OpenMP) are employed for mapping software to the hardware in order to leverage the hardware's architectural features. In this paper, we present an approach that automatically introduces thread level parallelism into Chombo, a parallel adaptive mesh refinement framework for finite difference type PDE solvers. In Chombo, core algorithms are specified in the `ChomboFortran`, a macro language extension to F77 that is part of the Chombo framework. This domain-specific language forms an already used target language for an automatic migration of the large number of existing algorithms into a hybrid MPI+OpenMP implementation. It also provides access to the auto-tuning methodology that enables tuning certain aspects of an algorithm to hardware characteristics. Performance measurements are presented for a few of the most relevant kernels with respect to a specific application benchmark using this technique as well as benchmark results for the entire application. The kernel benchmarks show that, using auto-tuning, up to a factor of 11 in performance was gained with 4 threads with respect to the serial reference implementation.

1 Introduction and Motivation

As supercomputers move into the Petascale and Exascale regime, application codes are needing to exploit several levels of explicit parallelism to make use of these highly concurrent architectures. Chombo is a highly successful MPI-based parallel PDE simulation package. This work describes one part of the effort to create a hybrid parallel Chombo package that uses MPI for the high level existing parallelism, and OpenMP for the fine-grained loop-level parallelism existing in the code already. These low-level multidimensional loops are spread across hundreds of lines of existing legacy F77 code. Fortunately, the code is written in a special form of F77 called `ChomboFortran`, a domain-specific language used by the Chombo development team to write portable, maintainable, dimension-independent Fortran code and to automate the process of linking the main C++ library to the F77 kernels. This project is an attempt to explore using this already existing language as a target for writing fine-grained loop parallel code automatically, and provide sufficient context for sophisticated compute kernel extraction and auto-tuning optimization work.

Experiments using a Godunov gas dynamics simulation and an ice sheet modeling calculations demonstrate that our methodology can be integrated into advanced numerical methods with minimum effort. Results on large-scale Opteron- and Nehalem-based supercomputers show that significant performance achievements can be attained using our hybrid and auto-tuned blocking schemes, if sufficient computational intensity as available in the underlying kernel. Overall, our approach highlights the potential of automating improved parallel performance, reducing memory requirements, and enhancing load balancing efficiency for an important class of Chombo-based adaptive mesh refinement calculations.

1.1 Chombo

Block-structured adaptive mesh refinement (AMR), developed by Berger and Oliger [2, 3, 6] for computational gas dynamics, is a multiscale algorithm that achieves high spatial and temporal resolution in localized regions of dynamic multidimensional numerical simulations. A broad range of physical phenomena modelled by PDEs exhibit multiscale behavior where variations in the solution occur over scales that are much smaller than the overall problem domain. Examples include flame fronts arising in the burning of hydrocarbon fuels, nuclear burning in supernovae, effects of localized features in orography or bathymetry on ocean currents, tracking tropical cyclones, localized kinetic effects for plasma physics problems, and, in general, small scale effects due to nonlinear instabilities. In each of these problems, the fundamental mathematical description is given in terms of various combinations of PDEs of classical type (elliptic, parabolic, hyperbolic).

1.2 Related Work

As Chombo is a package based on finite difference type PDE solvers, many of the F77 kernels do stencil computations on structured grids. Within this domain, specialized frameworks were proposed [5, 9]. The stencil algorithms are specified in high-level languages, which compile to low-level counterparts to which hardware-dependent loop transformations are applied. Recent more generally applicable approaches include SEJITS [4] and PetaBricks [1]. SEJITS is an approach, which does a just-in-time compilation of an algorithm or a part of an algorithm specified in a high productivity language (such as a scripting language like Python) to an "efficiency-level" language, which maps more directly to the underlying hardware. PetaBricks is a framework that tries to compose an algorithm from sub-algorithms selected by an auto-tuner and thereby guaranteeing that the final implementation is hardware-efficient and maps well to the selected hardware architecture.

2 Methodology

In a Chombo application, two programming languages are used: The general program logic is written in C++ in order to be able to benefit from the rich data structures and from the programming paradigms that C++ supports. Fortran is used for the performance-critical routines operating on a well-defined subset of data, since the design of the Fortran language enables more rigorous compiler optimizations and therefore it often offers superior floating-point performance.

These performance-critical routines are written in a Chombo-specific Fortran dialect (`ChomboFortran`): `ChomboFortran` is a domain-specific language originally designed to provide a clean interface for the programmer between the logic part of the framework written in C++ and the Fortran routines and, more importantly, to enable a dimension-independent implementation of the routines. It is implemented as a source-to-source translator (`ChomboFortran` to Fortran77) parsed by a set of simple Perl scripts.

Mostly, the computationally intensive code is contained in `multido` macros, which are expanded to loop nests sweeping over a box of the desired dimension. This makes the `multido` macros a perfect target for automatic fine-grained loop-level parallelization as well as loop structure optimizations, e.g., cache blocking. If a kernel is found to benefit from blocking, auto-tuning is used to determine the block sizes for which the kernel performs best on the given hardware architecture.

2.1 Automatic Loop-Level Threading

A typical `ChomboFortran multido` loop sweeps over a box doing a point-wise or a stencil (nearest neighbor) computation on each of the array elements, Chombo being an adaptive mesh refinement framework for finite difference codes. In most of the loops, the number of arithmetic operations on an array element is limited (and, in fact, constant with respect to the problem size). Hence, the performance is typically limited by the available memory bandwidth. For bandwidth-limited compute kernels it is essential to minimize memory traffic. If the grid on which the computation is made does not entirely fit into the cache, cache blocking is a means to increase data reuse and therefore reduce memory traffic.

In the `Chombo Fortran`, a `multido` loop has the following syntax:

```
CHF_MULTIDO[box;i;j;k]
  array(CHF_IX[i;j;k]) = ...
CHF_ENDDO
```

In this snippet, `box` defines the lower and upper bounds of the rectangular box over which is iterated, and `i`, `j`, `k` are the loop indices that will be used in the generated loop nest. If the dimension is lower than 3, the surplus index variables will be discarded. E.g., if the number of dimensions is 3, the original `Chombo Fortran` processor translates this snipped to a simple triply nested loop,

```
do k = boxlo2, boxhi2
  do j = boxlo1, boxhi1
    do i = boxlo0, boxhi0
      array(i,j,k) = ...
    enddo...
```

The extended, auto-parallelizing version of the `ChomboFortran` processor can block the loop nest and replace it by a $D(B+1)$-fold nested loop, where D is the number of dimensions and B is the number of blocking levels. The outermost level of blocks, or, if $B = 0$, the simply outermost loop is used for parallelization. Each thread is assigned a number of consecutive largest blocks. If no blocking is desired, the iterates of the outermost loop is dealt out to the available threads. We do not do any more elaborate loop transformations at this point, such as loop unrolling or more advanced fashions of

tiling. Also, a compiler directive is inserted that gives a hint to the compiler to treat write operations to arrays as streaming stores and thereby bypassing the cache when data is written back to main memory.

OpenMP is used for parallelization. If the parallelizer is instructed not to do any blocking, an `OMP PARALLEL DO` sentinel is created, in which the code resulting from expanding the `CHF_MULTIDO` loop macro is wrapped, and the default OpenMP thread scheduling is applied. In case of blocking, the assignment of blocks to threads is done explicitly within the parallel region by issuing guards that let the threads only execute the blocks to which they were assigned.

First, the parallelizer needs to check whether parallelization of the loop nest is possible. E.g., if scalar variables are assigned that are outputs of the routine, that output value depends on the sequential order of the loop iterates, and it cannot be parallelized. Also, if a reduction is performed over elements depending on the loop indices, this must be recognized and handled accordingly. Currently, as the reduction loops encountered in the Chombo code do not appear in the critical routines, parallelization is omitted. The parallelizer also needs to find temporary variables in the loop that need to be privatized.

If the code is to be blocked ($B > 0$), both the block sizes and the number of consecutive blocks assigned to one thread are not fixed at compile time. We introduce variables for the block sizes and number of blocks (`AUTO_*` in the code snippet below). These variables are added temporarily as parameters to the routine. The basic idea is that the auto-tuner then measures the runtime of the code for different parameter configurations, picks the configuration for which the code has the lowest runtime, and substitutes the values back into the final code.

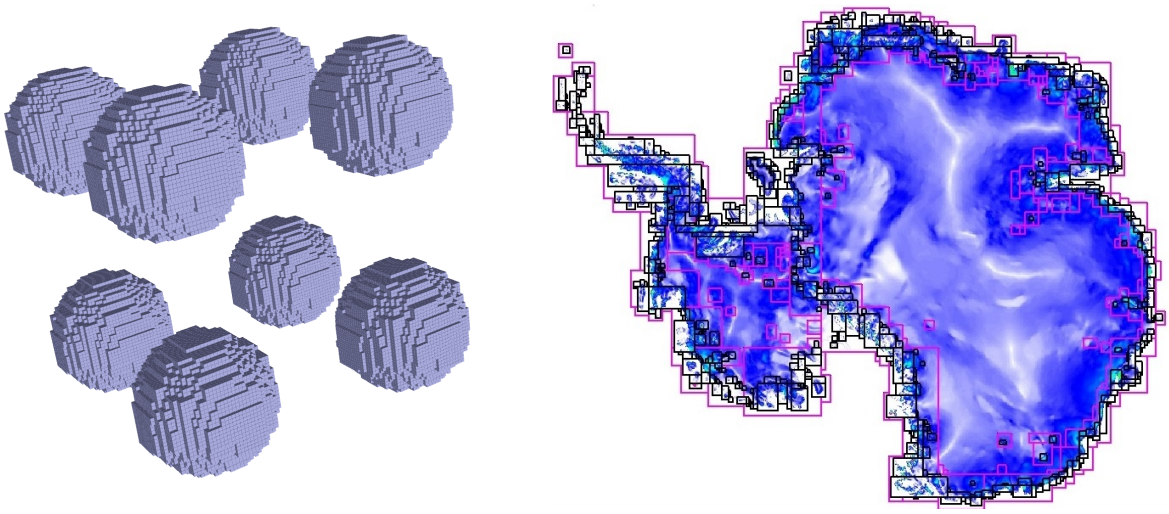
The snippet below shows an example of the result of automatic parallelization and blocking.

```
!$OMP PARALLEL FIRSTPRIVATE({private variables}) PRIVATE({loop indices})
do blk0k = boxlo2, boxhi2, AUTO_BLOCKSIZE_2
  do blk0j = boxlo1, boxhi1, AUTO_BLOCKSIZE_1
    do blk0i = boxlo0, boxhi0, AUTO_BLOCKSIZE_0
      maxblk1k = min(boxhi2, blk0k+AUTO_BLOCKSIZE_2-1)
      maxblk1j = min(boxhi1, blk0j+AUTO_BLOCKSIZE_1-1)
      maxblk1i = min(boxhi0, blk0i+AUTO_BLOCKSIZE_0-1)
      if ({thread control}) then
        do k = blk0k, maxblk1k
          do j = blk0j, maxblk1j
            !DEC$ VECTOR NONTEMPORAL(array)
            do i = blk0i, maxblk1i
              array(i,j,k) = ...
            end do
          ...
```

2.2 Kernel Extraction and Benchmark Harness Generation

The best choices for blocking parameters may vary from loop to loop. Therefore, we want to determine the optimal blocking parameters for each `CHF_MULTIDO` loop individually, and we need a means to extract individual loops along with their contexts. We call a subroutine containing exactly one `CHF_MULTIDO` a *kernel*. If there are multiple `CHF_MULTIDO` loops in a subroutine, the kernel extractor creates multiple kernels from that subroutine. While we want to preserve the calculations in the subroutine preceding the loop, we also want to ensure that the loop is actually executed. Therefore, if, e.g., the loop is contained in an `if` statement, that `if` statement is omitted by the kernel extractor.

Along with the Fortran kernel code, a C++ source file is automatically generated that calls the kernel subroutine with random input data, and thereby mimics the way the Fortran kernel is called in Chombo, and benchmarks the kernel by means of measuring the number of clock cycles spent in the kernel execution. All the subroutines parameters added by the parallelizer are exposed to the command line of the microbenchmark.



(a) (b)

Figure 1: (a) Replicated grids at the finest AMR level used in the weak scaling performance study of the Godunov benchmark, covering the shock front of a spherical explosion in 3D. (b) Snapshot from BISICLES simulation showing ice displacement velocity, refined from 10km baseline resolution.

This automatically generated microbenchmark harness can be used for both benchmarking and profiling an individual kernel and to build an executable used by the auto-tuner.

2.3 Auto-Tuning

The auto-tuning methodology has been successfully adopted in several well-known libraries and frameworks including ATLAS [14], FFTW [7], OSKI [13], and SPIRAL [11]. Oftentimes, in codes that are written to match a hardware architecture for best performance, parameters capture certain aspects of the hardware. E.g., they can be related to cache sizes, and the performance depends highly and often non-trivially on these parameters, so that choosing good values is essential, but hard to predict. The auto-tuner methodology provides a means to select such parameters even if no performance model exists.

The Chombo auto-tuner runs the microbenchmark executable repeatedly with varying parameter configurations. For the results presented in this paper, we used two runs the Powell search method, which iteratively determines the optimum along fixed parameter axes. The second run uses the parameter configuration found in the first run as starting point.

3 Experimental Testbed

In this section we first discuss the target applications: the 3D Godunov Gas Dynamic simulation and the 2D Ice Sheet Modeling calculation. We then summarize the key architectural components of the evaluated platforms.

Hyperbolic Gas Dynamics Benchmark We benchmarked an explicit method for unsteady inviscid gas dynamics in 3D that is based on an unsplit PPM algorithm [10, 15]. This benchmark is described in previous related work [12].

We have developed benchmarking methods based on *replication scaling* which take a grid hierarchy and data for a fixed number of processors and scale the problem to higher concurrencies by making identical copies of the hierarchy and the data (Fig. 1). The full AMR code (processor assignment, problem setup, etc.) is run without any modifications so it does not take advantage of the replicated

grid structure. Replication scaling tests most aspects of weak scalability, is simple to define, and provides results that are easy to interpret. Thus, it is a very useful tool for understanding and correcting impediments to efficient scaling in an AMR context. Furthermore, it is a good proxy for the scaling behavior of real applications. For example, a large part of the gas turbine calculation will be the simulation of multiple identical burners arranged in a ring.

Ice Sheet Modeling The dynamics of ice sheets span a wide range of scales. Localized regions such as grounding lines and ice streams require extremely fine (better than 1 km) resolution to correctly capture the dynamics. Conversely, there are large regions where such fine resolution represents a waste of resources, making ice sheets a prime candidate for adaptive mesh refinement, in which finer spatial resolution is added where needed. The Berkeley ISICLES (BISICLES) [8] is studying the construction of a high-performance scalable AMR ice sheet model using the Chombo parallel AMR framework. A visualization of a BISICLES simulation showing ice displacement velocity is shown in Fig. 1.

In this paper, we examine the top 5 kernels by time in a BISICLES simulation. All five are 2D kernels (depth averaged from the 2.5D discretization). Moreover, 4 are viscous tensor operators from the elliptic solver while `COMPUTEL1L2MU` computes depth averaged viscosity.

3.1 Evaluated Platforms

In order to gauge the value of our threading and performance optimization techniques at scale, we employed two large supercomputers located at NERSC — a Cray XT4 named Franklin, and a IBM Infiniband cluster named Carver. By exploring both architectures, we gain insights into how changes in per-thread and per-process memory capacity, memory bandwidth, and network performance affect simulation capability, performance, and scalability.

Cray XT4 (Franklin) Franklin is a Cray XT4 supercomputer with over 9500 compute nodes each containing one 2.3GHz quad-core Opteron processor and one SeaStar2 network chip. Collectively the network chips form a low-latency, high-bandwidth 3D torus. Each Opteron processor instantiates four superscalar cores, a shared 2MB L3 cache, and a dual-channel DDR2-800 memory controller capable of delivering a theoretical DRAM bandwidth of 12.8GB/s. DRAM latency is hidden via hardware stream prefetching, and DRAM capacity is limited to 8GB. Each core includes both a private 64KB L1 cache and a private 512KB L2 cache. Each core is capable of executing one double-precision SIMD add and one double-precision SIMD multiply per cycle for a theoretical peak performance of 9.2GFlop/s/core (36.8 GFlop/s/processor).

Nehalem Infiniband Cluster (Carver) Carver is an IBM iDataPlex cluster with 400 SMP compute nodes each containing two 2.66GHz quad-core Xeon X5550 (Nehalem) processors and a 4X QDR InfiniBand card. The machine is built on a hybrid network topology with local fat trees inside a global 2D mesh. The Nehalem architecture is similar to the Opteron processor found in Franklin. However, the L1, L2, and L3 caches are 32KB, 256KB, and 8MB respectively. Moreover, each Nehalem processor is connected to 12GB of DDR3-1333 DRAM. In practice, the typical memory bandwidth of about 19GB/s is far less than the theoretical bandwidth of 32GB/s. Like the Opteron, a Nehalem core can execute one double-precision SIMD add and one double-precision SIMD multiply per cycle for a theoretical peak performance of 10.66GFlop/s/core (42.66 GFlop/s/processor). HyperThreading is disabled on this machine. To obviate the complexity of NUMA management, in all of our experiments a threaded MPI process never spans more than one processor. Overall, an MPI process on Carver has 16% more compute, 50% more DRAM capacity, and almost double the DRAM bandwidth of a process on Franklin.

4 Kernel Results

In this section, we present kernel-only performance results for five kernels from each of the applications discussed in section 3 in which most of the compute time is spent during the simulations. The kernel benchmarks were carried out on both the Nehalem ("Carver"; the sub-figures (a) and (c) in Fig. 2 and the Opteron ("Franklin"; sub-figures (b) and (d)) hardware platforms; in each case on one compute node (more specifically: one NUMA domain) only, since for the kernel-only benchmarks we are only interested in thread level parallelization results. Section 5 will concentrate on the hybrid MPI+OpenMP full applications.

On Carver, we used the gcc 4.4.2 to compile the extracted kernel codes; on Franklin, we used both the vendor compiler – the Cray compiler 7.2.7 – and gcc 4.5.1. In this section, we only show the results obtained with the Cray compiler, since it proved to be more amenable to the blocking optimization. We used the `-O3 -ffast-math -msse2` code generation and optimization flags for the gcc, and `-O3 -O fp3` for the Cray compiler.

The benchmarks were carried out as follows: after one warmup kernel run we did 5 runs, counting the number of clock cycles spent in each individual run, from which the median timing was selected. Before each run the cache was flushed.

The vertical axes of the charts presented in this section show relative speedup numbers of the automatically parallelized versions of the kernels with respect to the original serial version. One stacked bar shows the scalability from 1 to 4 threads (the number of threads per NUMA domain on both architectures) in blue, as well as the effect auto-tuned blocking in green, using 4 threads and block sizes determined by the auto-tuner. The horizontal major axis represents the kernels, the minor axis the size of the grid on which the kernel was run.

The selected kernels have different flavors of computations and memory access patterns. Notably, in Fig. 2 (a) and (b), which shows the hyperbolic gas dynamics benchmark kernels, the `RIEMANNF` kernel scales almost perfectly, which is due to its high arithmetic intensity. The fact that blocking (an optimization that addresses cache capacity misses) showed little benefits is no surprise given this kernel performs point-wise updates that do not demand a large cache working set. Using the Cray compiler on Franklin, blocking results in a substantial speed improvements for 3 of the 5 kernels in the hyperbolic gas dynamics benchmark, whereas with gcc blocking did not have any noticeable effect, both on Carver and also on Franklin (which is not shown in the figure). In particular, the performance of the `GETADWDXF` kernel increases by a factor of more than 3 with a single thread compared to the serial reference implementation. With 4 threads the performance benefit is around a factor of 8 to 11. This might be due to the fact that gcc seems to optimize better than the Cray compiler. The absolute execution times were better when the GNU compiler was used. The blocked kernels compiled with Cray are roughly on par with GNU. However, Cray's OpenMP implementation seems to be more efficient than the one of GNU: Using GNU, the relative speedup of a single thread with OpenMP turned on drops, in certain cases, significantly below 1, whereas the Cray compiler's performance stays stable when OpenMP is turned on, or, for some reason, is even increased compared to the base version without the OpenMP pragma.

On the other hand, the parallel efficiency for the majority of the 2-dimensional ice sheet simulation kernels is rather poor. Most of the BISICLES kernels considered do only a limited number of floating point operations per memory reference, and the relatively few data elements per grid (as few as 256) make it difficult to apply fine-grained thread-level parallelization; the overhead of OpenMP is too large to be overcome by adding more concurrency. This can result in relative speedups that are < 1, which of course is not desirable. Hence, we need to detect such cases and omit inserting the OpenMP pragmas. The `COMPUTEL1L2MU` kernel, which is a point-wise kernel and does a large number of floating point operations per grid point, again scales almost perfectly.

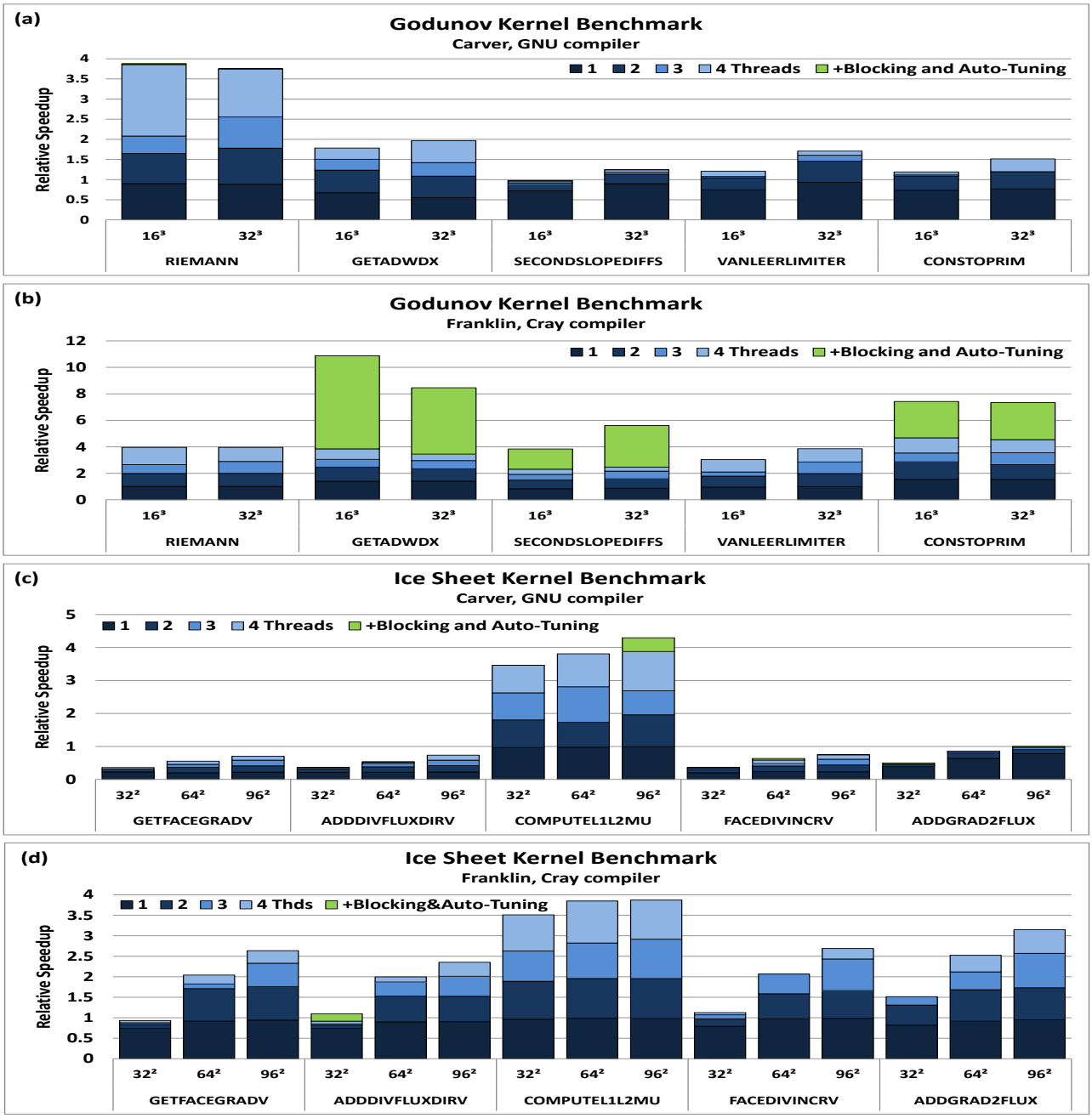


Figure 2: Kernel-only benchmarks of the top 5 kernels of the hyperbolic gas dynamics benchmark using one NUMA domain on both Carver and Franklin, for (a–b) Godunov and (c–d) Ice Sheet. Performance is shown relative to the original (unthreaded) version.

Total Cores	Points Updated	Total Memory (GB)		Solve Time (sec)	
		Original	Hybrid	Original	Hybrid
Franklin					
512	8.0e09	221.7	166.8	177.3	361.8
1024	1.6e10	485.4	345.0	180.4	363.4
2048	3.3e10	1136.7	733.7	179.8	363.2
4096	6.7e10	2938.8	1646.3	186.4	372.0
8192	1.3e11	8534.3	3979.0	200.4	378.4
16384	2.5e11	—	10519.3	—	411.3
Carver					
64	1.0e09	24.1	19.7	122.3	169.1
128	2.0e09	49.2	39.6	121.2	173.1
256	4.0e09	102.4	80.1	111.0	170.0
512	8.0e09	218.9	163.2	115.0	170.7

Table 1: Godunov application characteristics comparing total memory requirements and solve time for original and hybrid schemes using weak scaling.

Overall, we believe hybrid implementations of Chombo must be selective in deciding whether to apply threads to intra-grid parallelism (parallelize one grid at a time) or to apply threads for inter-grid parallelism (each thread operates independently on a subset of the box list). The former pools on-chip caches and overlapping working sets to maximize locality while the latter amortizes OpenMP overhead.

5 Godunov Application Results

In this section we explore the behavior of our automated threading method when integrated with the full Chombo application. Parallel executions of standard Chombo applications place one MPI task per core. The experiments in Table 1 show performance comparisons of the original version versus our hybrid MPI/OpenMP methodology, when running in weak scaling mode (increasing problem size with respect to concurrency). Note that the individual kernel times showed similar performance behavior both within the free standing harness and the full Chombo application.

As can be seen from Table 1, the total solve time slows down by about a factor of two using our hybrid approach. This is because only a subset of the code has been threaded, resulting in significant slowdowns for unthreaded code portions as less MPI processes are used. Future work will focus on threading the remainder of the C++ computations and Fortran routines not amenable to our existing methodology. Observe, however, that the memory footprint is dramatically reduced using our hybrid approach. This is due to the metadata whose size is a function of the number of MPI tasks. Thus we see more than a 2× reduction of memory requirements at 8K Franklin cores, and the ability to run the hybrid version at 16K cores — whereas the original code fails due to memory constraints.

Another key advantage of the hybrid approach is improved load balancing for strong scaling simulation. The hybrid scheme reduces the overall number of MPI tasks, which in turn decreases the number of Chombo box components. Since load balancing the boxes is a function of MPI task count, the hybrid approach allows for potentially significant improvement in overall performance when strong scaling at large concurrences.

6 Conclusions

In this paper we have shown that our attempt at re-purposing the existing domain-specific language `ChomboFortran` to serve as input to a parallelization and auto-tuning framework gave us an effective performance tool, from which especially loops with high arithmetic intensity benefit. It is also a productivity tool: The portion of the gas dynamics benchmark code in which around 50% of the compute time is spent was automatically brought into a hybrid threaded/MPI model without any user interaction. In the kernel-only benchmarks, we measured speedups up to a factor of 11 with 4 threads. On the full application scale, using the hybrid model the memory requirements are reduced substantially (by more than 2×), resulting in an improved code scalability over the flat MPI baseline.

The ability of kernel extraction and automated benchmarking can both engage an auto-tuner to find optimal block sizes and provide a tool for detailed kernel-level performance studies. We carried out validations which showed that the performance behavior of individual kernels is similar to the performance behavior of the respective embedded in an application.

In this work, we chose to follow the initial macro expansion approach of `ChomboFortran`. While it is possible to add automatic parallelization in this way and implement some basic loop transformations, there are still serious limitations. Rigorous parallelization legality tests and more sophisticated loop transformations, which require data dependence analysis and expression manipulation, necessitate a more elaborate parser and building an abstract syntax tree to manipulate the code structure. This will be addressed in future work.

There is not much more thread-level parallelism available in these fine block loops. Finer decomposition will just create interference with processor instruction-level parallelism. Harnessing higher concurrency architectures will require threading to operate in both the box interation loop as well as the `multido` loop level. This multi-level thread parallelism will be a challenging next step.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Jason Ansel, Yee Lok Won, Cy Chan, et al. Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms. Technical Report MIT-CSAIL-TR-2010-032, MIT, Cambridge, MA, Jul 2010.

[2] M. Berger and P. Collela. Local adaptive mesh refinement for shock hydrodynamics. *J. Computational Physics*, 82:64–84, May 1989.

[3] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, et al. SEJITS: Getting Productivity and PerformanceWith Selective Embedded JIT Specialization. In *First Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[5] M. Christen, O. Schenk, and H. Burkhart. Patus: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[6] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen. Chombo software package for AMR applications: design document. `http://davis.lbl.gov/apdec/designdocuments/chombodesign.pdf`.

[7] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[8] High-Performance Adaptive Algorithms for Ice-Sheet Modeling. `http://crd.lbl.gov/SCG/bisicles/`.

[9] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An Auto-tuning Framework For Parallel Multicore Stencil Computations. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, April 2010.

[10] G. Miller and P. Colella. A conservative three-dimensional Eulerian method for coupled solid-fluid shock capturing. *Journal of Compuational Physics*, 183:26–82, 2002.

[11] Markus Püschel, José M. F. Moura, Jeremy Johnson, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[12] Brian Van Straalen, John Shalf, Terry Ligocki, Noel Keen, and Woo-Sun Yang. Scalability Challenges for Massively Parallel AMR Application. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[13] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[14] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[15] P. R. Woodward and P. Colella. The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Compuational Physics*, 54:115–173, 1984.