Refactoring Scientific Applications for Massive Parallelism

John M. Dennis and Richard D. Loft

Abstract We describe several common problems that we discovered during our efforts to refactor several large geofluid applications that are components of the Community Climate System Model (CCSM) developed at the National Center for Atmospheric Research (NCAR). We stress tested the weak scalability of these applications by studying the impact of increasing both the resolution and core counts by factors of 10 to 100. Several common code design and implementations issues emerged that prevented the efficient execution of these applications on very large microprocessor counts. We found that these problems arise as a direct result of disparity between the initial design assumptions made for low resolution models running on a few dozen processors, and today's requirements that applications run in massively parallel computing environments. The issues discussed include non-scalable memory usage and execution time in both the applications themselves and the supporting scientific data tool chains.

1 Introduction

For the past 30 years, the amount of computing power that could be applied to scientific problems has grown exponentially. This amazing growth rate was a direct result of decreases in transistor sizes, which for decades, directly translated into increases in microprocessor clock frequency and consequently improved single thread performance. The doubling of clock frequency every 18 months became strongly associated with Moore's Law, that actually only describes the underlying

John M. Dennis
Computational & Information Systems Laboratory, National Center for Atmospheric Research, Boulder, CO 80307-3000 e-mail: dennis@ucar.edu

Richard D. Loft
Computational & Information Systems Laboratory, National Center for Atmospheric Research, Boulder, CO 80307-3000 e-mail: loft@ucar.edu

rate of improvement in photolithographic techniques. Regardless, for an application developer in the latter part of the 20th century, Moore's law meant that exponential performance improvements came on a steady schedule with little or no effort. In this regime there was little incentive to improve application performance by increasing parallelism.

In middle of the 2000s these circumstances began to change, as several fundamental factors began to limit microprocessor frequency. Local (on-chip) interconnect delays began to dominate feature size as the determining factor of clock speed. The heat densities being generated by $\sim$3 GHz microprocessors began hitting thermal design limits. The growing gap between memory and processor speeds increased memory access times, thereby creating the so-called "memory wall", in which faster processor clock speeds no longer guaranteed better performance. In response to these challenges, microprocessor architects began moving toward chip multiprocessor (CMP) designs: under this paradigm, chip performance improvements would come from doubling the number of processors or cores on a silicon die, while clock speed would only increase at a modest rate of 15% per 18 months. It seems clear now that this development is a long-term technology trend, derived from fundamental limitations of the underlying semiconductor technology. For application programmers this means that improved performance must come from speed-ups derived from increased parallelism.

A similar situation has developed in disk subsystem architecture as well: disk spindle parallelism is necessary to match I/O performance with parallel computational performance. Through parallel filesystem technology, modern petabyte filesystems aggregate thousands of rotating disk spindles and access channels to achieve high I/O bandwidth. Thus, as with CPU's, the path to performance for I/O intensive problems is through parallelism.

The impact of the return of massive parallelism is reflected in the increasing number of parallel computing initiatives sponsored by a variety of agencies of the U.S. government. In the High Performance Computing (HPC) arena, much of this effort is now focused around what is known as petascale computing: harnessing $\mathcal{O}$(100,000) or more cores to achieve a petaflop - (10^{15} floating point operations per second). These efforts include funds for the acquisition and deployment of petascale systems, as well as research and development money to develop algorithms and associated applications able to effectively use these systems. For example, the National Science Foundation (NSF) has initiated an ambitious "Track-2" and even larger "Track 1" programs to procure, deploy and operate several petascale systems over the next few years. The first so-called "Track-2" system, Ranger, was installed in Texas Advanced Computing Center in 2006. Ranger is a 62,976 core system, based on the quad-core AMD Barcelona microprocessor with an Infiniband® (IB) interconnect switch designed by Sun Microsystems. A second "Track-2" system, Kraken, a Cray XT5 system currently has 99,072 cores, and was awarded to the National Institute for Computational Sciences (NICS) in 2007. The NSF award for a super-sized "Track-1" system capable of a peak speed of least 10 petaflops, was awarded to the Illinois-based Blue Waters Consortium. Additional petascale com-

puters in the United States are being deployed by the Departments of Energy and Defense, and are reportedly planned in Japan, Europe, and China.

Of course, without scalable applications, these large systems can't provide the application acceleration that leads to scientific progress for many important problems, such as climate modeling. To address this issue, U.S. government agencies have also funneled research dollars toward developing new algorithms, frameworks, and applications. Examples of these include the DoE SciDAC and NSF PetaApps programs. Such programs have allowed developers of scientific applications (including ourselves) to make important progress in preparing such codes for operations on increased numbers of cores.

The availability of massively parallel computing systems will place a premium on the scalability of applications. Even so, not every scientific problem needs petascale computing, nor is it the case that every application has a large code base that makes it prohibitively expensive for it to be rewritten from scratch. However, we are interested in the significant class of scientific applications for which improving the parallel performance by refactoring the existing code is the only reasonable option available.

Code refactoring is usually defined as the process of modifying the internal structure of an application without changing the external functional behavior. Such refactoring can be done for a variety of purposes: for example, for readability, performance, or maintainability. The cost, both in terms of human and computational resources, of validating the refactored application is an important determiner of the overall cost of such projects. Refactoring scientific applications for parallel scalability is especially challenging, often requiring new, more suitable algorithms. Using existing parallel programming paradigms, such as Message Passing Interface (MPI) described in Snir et al (2000) a distributed memory parallel programming library or a shared memory parallel programming standard OpenMP (2005), means that changing the level of parallelism often requires new data structures, and introduces new design issues, unique to parallel execution, that usually have not been considered, let alone addressed, by earlier application developers. Examples of these issues include race conditions, resource contention, and load balancing. The challenge of these restructuring issues is exacerbated by the extraordinary cost of testing and validating many complex applications at scale.

Here we discuss our teams experiences in refactoring six large climate model components for massively parallel execution at significantly increased resolutions. Climate applications represent an especially rigorous test of existing parallel refactoring techniques. First, climate applications, particularly at high resolutions, can be extraordinarily expensive to run: for example, a single simulated year of a high resolution version of the Community Climate System Model requires roughly 80,000 CPU-hours per year on the Kraken XT-5 system at NICS. Second, the climate system is an interacting system of nonlinear PDEs with a multitude of computationally intensive forcings and feedbacks: this makes the system sensitive to initial conditions . Even tiny changes in results at machine-level precision will generate completely different realizations of the climate system after a sufficient amount of time has passed. Third, since the scientific predictions of climate applications are sta-

tistical in nature, their results are expressed in monthly and seasonal means and variances of measurable quantities. Thus they require long (perhaps multi-year) runtimes to completely validate.

Previous work by Dennis (2007) and Dennis and Tufo (2008) has revealed numerous structural issues in these codes that prevent successful or efficient execution at higher resolutions and on very large core counts. The reasons for these design issues are complex. Many climate system applications have long histories and have been refactored several times in the past. For example, NCAR's Community Atmosphere Model (CAM) was refactored for multitasking parallelism on vector systems in the 1980's, refactored again for distributed memory (message passing) execution in the early 1990's, and finally modified to support hybrid message passing and OpenMP execution in the late 1990's. For scientific reasons and until recently, climate application resolutions have remained around one to three degrees (100-300 km), as the focus of model development centered on improving the representation of processes and capturing new climate system feedbacks. For these resolutions, a modest level of parallelism 32 to 128 processors was sufficient to achieve acceptable scientific throughput rates. Now, with a growing sense that the resolution of the climate system must be improved to address remaining model biases, there is a new focus on conducting exploratory research at resolutions increased by one to two orders of magnitude. Not surprisingly perhaps, the need to run efficiently on 10,000 to 100,000 or greater cores counts, which was never considered by developers, is now needed to provide these high-resolution simulation capabilities.

The essence of the problem we faced in refactoring these applications is that many of their design assumptions, which worked fine on roughly a hundred cores, became problematic on tens or hundreds of thousands of cores. At a high level, the impediments we have uncovered may be classified as either non-scalable memory usage or execution time. Common non-scalable memory usage problems include use of replicated metadata, excessive number of global arrays and serialized I/O. Common problems that impact the scalablity of execution time include: non-scalable initialization and communication time. In addition, we describe prevalent scalability issues within the toolchain that also impedes scientific progress and discovery. We present this description of common design issues in the hope that they could be used as a guideline to design future applications for efficient execution at these and even higher levels of parallelism.

2 Background

We have refactored six applications. The Parallel Ocean Program (POP) described in Jones (2003) and Jones et al (2005), developed at Los Alamos National Laboratory is an important multi-agency ocean model used for global ocean modeling. The Community Ice CodE (CICE) described in Hunke and Lipscomb (2008) also developed at Los Alamos National Laboratory is also an important multi-agency code used to model sea ice. The Community Atmosphere Model (CAM) descried

in Collins et al (2006), is a widely used atmospheric model, whose development is based at the National Center for Atmospheric Research (NCAR), but has a large international community of contributors and collaborators. The Community Land Model (CLM) described in Hoffman et al (2005), which was developed at NCAR in collaboration with many national collaborators, models the land surface. The High Order Methods Modeling Environment (HOMME), an atmospheric dynamical core developed at NCAR and described in Dennis et al (2005) and **??** [[check]], is used to evaluate new numerical and computational algorithms. The flux coupler (CPL) combined with POP, CICE, CAM, and CLM form the state of the art climate model, the Community Climate System Model (CCSM). CCSM is one of the most extensively used climate models in the world and has participated in the Intergovernmental Panel on Climate Change (IPCC) Assessment reports by Solomon et al (2007). POP, CICE, CAM, and HOMME are hydrostatic models that solve the equations of motion on multiple coupled horizontal computational meshes. Without the use of the hydrostatic approximation, the equations and algorithms are considerably more complex. The size of the horizontal computational mesh that is decomposed across cores is significantly larger than the number of levels in the vertical dimension. While CLM does not solve any equations of motion, it in addition to CPL, decomposes the horizontal computational mesh across cores.

Inspired by the work of Shingu et al (2002), an attempt was made to execute a predecessor to HOMME at 10 km resolution, which represented a significant increase relative to what had previously been attempted. Due to its use of replicated metadata, which will be described in Section 3.2, HOMME was not even able to complete the initialization of the message passing communication library before it exhausted memory. While a refactoring of the data structures within the communication library enabled it to initialize, a large number of additional issues prevented its successful execution. After concerted effort to improve the scalability of HOMME by a number of developers, CAM based on the HOMME dynamical core (CAM/HOMME) has demonstrated excellent scalability on a range of core counts and resolution combinations. The integration rate of CAM/HOMME on a Aquaplanet simulation courtesy of Taylor et al (2008) at 0.5° (56 km), 0.25° (28 km), and 0.125° (14 km) is illustrated in Figure 1. Note that CAM/HOMME achieves excellent scalability on the Cray XT4, and the IBM Blue Gene/L (BGL) and Blue Gene/P (BGP) systems.

Our preliminary work with HOMME illustrated that the ultimate scalability of an application was determine not only by the underlying scalability of its numerical algorithms but by the efficiency and quality of its implementation. Subsequent work with other applications revealed that issues that had limited the parallelism or even prevented successful execution of HOMME were also commonly found in other applications. We next describe a set of issues that were discovered in multiple applications that prevented efficient execution on very large core counts.

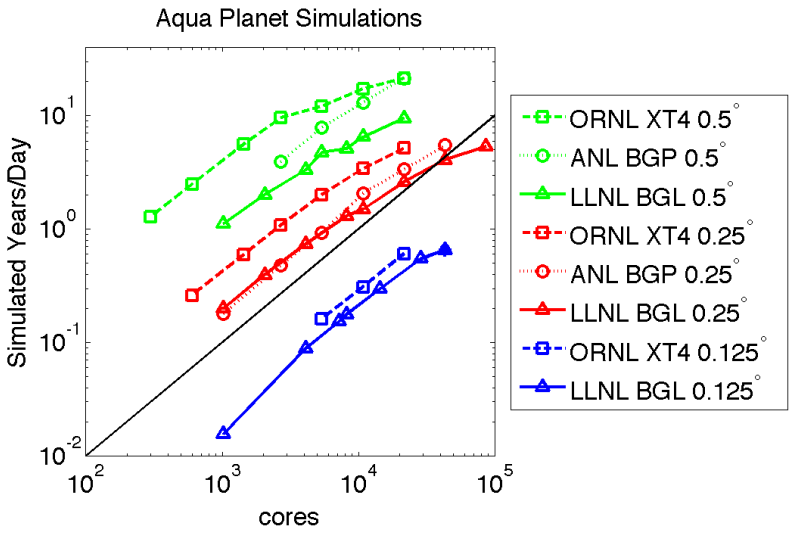

Fig. 1 Integration rate of CAM/HOMME on Aqua-planet simulation. Note that scalability is achieved 0.5° 0.25° and 0.125° on Cray XT4, IBM Blue Gene/L and Blue Gene/P systems

3 Scalability

We base our observations on the work performed preparing these previously mentioned applications for execution at high resolution on very large core counts. Within the group of applications, the one with most limited parallelism has been tested on a maximum of approximately 3,300 cores, while the most parallelism demonstrated by an application was 96,000 cores. Scalability testing has been performed on several very large systems, including: a 40,960 core IBM Blue Gene/L system described in Adiga and et al. (2002) at Thomas J. Watson Research, a 128,000 core IBM Blue Gene/L system at Lawrence Livermore National Laboratory (LLNL), a 38,912 core IBM Blue Gene/L system at Brookhaven National Laboratory, a 2048 core IBM Blue Gene/L system at NCAR, a 10,000 core Cray Redstorm system described in RedStorm (2006) at Sandia National Laboratory, a 13,000 core Cray XT3/4 at Oak Ridge National Laboratory, a 9,000 core Appro linux cluster at Lawrence Livermore National Laboratory, and a 99,072 core Cray XT5 at National Institute for Computational Science (NICS). It is useful to evaluate scalability on more than just a single compute platform. The use of multiple compute platforms allows for differentiation between scalability problems in an application and the

compute platforms message-passing network. Each application had between five to seven issues. Table 1 provides a listing of which applications had which issues.

We found it particularly striking that many of the same design issues were found in applications created by different developers. While the applications were developed by groups of researchers from interrelated scientific disciplines, there is no common origin for all the applications. Therefore we believe that many issues that we have discovered represent inherent stumbling blocks for developers creating parallel applications.

Problems	Applications					
	POP	CAM	CICE	HOMME	CPL	CLM
replicated metadata	yes	yes	yes	yes	yes	yes
excessive global arrays	yes	yes	no	yes	yes	yes
serial I/O	yes	yes	yes	no	yes	yes
non-scalable initialization	yes	no	yes	yes	yes	yes
non-scalable communication	yes	no	yes	no	yes	no
debugging at scale	yes	yes	yes	yes	yes	yes
pre/post processing	yes	yes	yes	yes	yes	yes

Table 1 Presence of scalability issues within applications.

There are two different types of scalability, strong and weak. To test an applications strong scalability, a fixed size problem is executed on variety of processor or core counts. For a code with ideal strong scaling, use of twice the number of cores will reduce the execution time in half. In contrast to strong scaling, weak scaling fixes the size of the part of problem allocated to each core. For a code with ideal weak scaling, use of twice the number of cores will enable the execution of an application with twice the number of grid points in the same amount of time. Because climate modeling if frequently concerned with reducing the time to solution, and resolution changes very infrequently, only the strong scaling characteristics of an application is typically reported. However successful use of very large-scale parallelism will likely involve improvements in both the weak and strong scaling characteristics of the application.

A critical prerequisite to efficiently utilize very large-scale parallel systems is that the underlying numerical method is fundamentally scalable. In other words, scalability is only limited by the quality of implementation of the application or the scalability of the computing platform. The difference between numerical method and application scalability is illustrated by considering the parallelization approach used by POP and an older version of CAM. In the older version of CAM, which decomposed across latitude, a computational grid with 128 grid points in longitude and 64 grid points in latitude could only be parallelized across 64 cores. Additional parallelism within CAM was only possible through either a major change in the computational infrastructure, or an alternative numerical method. The scalability of CAM was limited from a structural perspective. A significant change to the computational structure and numerical method of CAM has since occurred, which has

improved its structural scalability. However, CAM still has structural scalability limitations. Alternatively, POP has a much more flexible approach to scalability. The only limit to parallelism within POP is that imposed on it by the compute platform. For example it is possible to place as few as several grid points per core, though with current compute platform characteristics this would not be an efficient configuration. However the flexibility of POP is an important feature and enables it to adapt to computational platforms with different balances between computational and communication costs. The flexibility of POP has allowed it to successfully adapt to both the Cray XT and IBM Blue Gene family of supercomputers. While structural scalability is necessary, it is not sufficient for efficient execution for very large-scale parallelism.

3.1 Scalability of Memory Usage

We begin our examination of scalability by discussing the scalability of an applications memory usage. The single most critical, and common issue in the application suite is the presence of non-scalable memory constructs. While problems with the scalability of an applications execution time may just reduce the simulation rate, problems with memory scalability will prevent an application from running at all. Recall that the initial attempt to execute a predecessor of HOMME at 10 km failed because it exhausted memory during initialization. Note that we are referring to memory that is due to the data-structures of an application, not that is devoted to the scientific calculations. Frequently, non-scalable data-structures can limit the type of science questions that can be asked. Even when it is possible to successfully execute an application, non-scalable memory may limit its execution to systems with larger memory. Because of the cost of memory is a considerable component of the overall system cost, systems with a large amount of memory will tend to be rarer and in greater demand than systems with comparatively smaller amount of memory per core. Therefore the ability of an application to run successfully in a small amount of memory gives it a competitive advantage relative to other applications that require larger amounts of memory.

3.2 Replicated metadata

One of the most common design issues discovered in the application suite is the unnecessary use of replicated metadata. We refer to metadata as something that describes the location of something else, like data structures that describe the location of grid cells in the domain decomposition, or the message-passing schedule. An example of message passing metadata is a data structure that indicates that core i (p_i) sends n bytes to core j (p_j). In reality, only the cores p_i and p_j truly require this piece of metadata. Cores p_k, where $k \neq i$,$k \neq j$ does not need metadata for cores

p_i and p_j. The replication of message passing metadata adds an $\mathcal{O}(numNeigh * P)$ sized data structure where P is the total number of cores and $numNeigh$ is the number of communication neighbors. If only relevant metadata is stored, the size of the data structure is reduced to $\mathcal{O}(numNeigh)$. The importance of eliminating replicated metadata has been demonstrated by Baker et al (2006). It should be noted that retaining a single serial algorithm within a parallel application forces replication of some metadata. For example, if an entire application is parallel except for serial I/O, then a data structure of size $\mathcal{O}(P)$ is necessary to describe the domain decomposition.

The most extreme example of the growth of metadata storage was located in CLM. Because of the particular design of CLM's dynamic domain decomposition strategy, $4 * P$ global arrays were present in the initialization subroutines. Global arrays in this case are arrays that are the size of the computational grid replicated on each core. For CLM with $1/6^{\circ}$ separation between grid points (2160×1080 total horizontal grid points), these four integer arrays would require 29 TBytes of memory on 10,000 cores. While it was easy to discover large data structures, it is more challenging to discover issues in smaller ones. Thus, long after addressing the blatantly obvious metadata issues, a number of additional replicated data structures continued to emerge.

A good technique for discovering the presence of replicated metadata is to measure the maximum memory usage and plot it as a function of core count. Maximum memory usage is the maximum memory used by a single core at any time during the simulation. While we found the Blue Gene memory usage tool *memmon* developed by Walkup (2007) to be very useful to identify memory usage problems within CLM, it is also possible to develop similar tools that use the Unix system call *getrusage*, or the Linux /proc file system to measure memory usage. The memmon tool provides function calls that print out current memory usage. Calls to the memmon utility were added to CLM at key points in the code. The resulting analysis of memory usage yielded surprising results. Figure 2 is a plot of the memory usage as a function of core count for CLM at high-resolution. The maximum memory usage for the original version of CLM (CLM-orig), along with two modified version of CLM (CLM-mod1) and (CLM-mod2) are shown. Note that CLM-mod2 uses an updated version of the Model Coupling Toolkit (MCT) described in Jacob et al (2005) and Larson et al (2005), while CLM-mod1 does not. Note that the memory usage for the CLM-orig was estimated from lower resolution configuration because its memory requirements were too large to successfully execute on Blue Gene. It is interesting to note that memory usage for both the CLM-orig and CLM-mod1 increase as core count increases. This behavior is a clear indicator of the presence of replicated metadata. The huge difference in memory usage for small core counts between CAM-orig and CLM-mod1 is an indication of another form of non-scalable memory, excessive global arrays, which will be discussed in Section 3.3.

We discovered $\mathcal{O}(numNeigh * P)$ sized metadata on all six applications that were examined. The simple replication strategy is understandable because most of the applications were initially designed on 32 to 128 cores. The size of the metadata only become problematic when attempting to scale to much larger core counts.

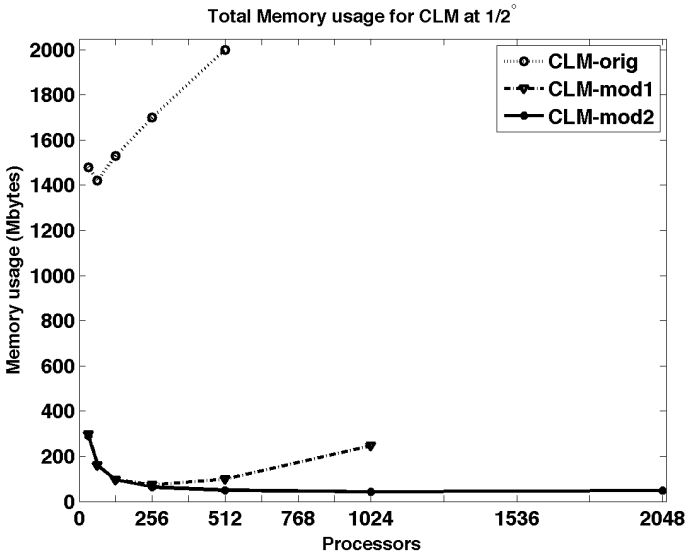

Fig. 2 A plot of the memory usage of several different versions of CLM as function of processor count. Elimination of both replicated data structures and global memory reduces memory usage for CLM at high resolution on 512 processors by a factor of 50.

3.3 Excessive global-sized arrays

Most of the applications in this study had an excessive number of arrays that were the size of the entire grid. We discovered that CLM used a large number of persistent global-sized arrays. We differentiate between persistent global arrays that are used for the duration of a run, versus temporary global arrays that are allocated within a subroutine and subsequently deallocated. While the elimination of persistent global arrays will always reduce the necessary memory usage for the application, the reduction of temporary global arrays may or may not reduce the maximum memory usage of the application. Elimination of temporary global arrays will only reduce maximum memory if the subroutine in which they are allocated is the actual location causing the maximum memory usage of the application to be reached.

It is possible to identify the presence of global arrays within an application by looking at memory usage as a function of core count for a fixed resolution. The memory usage of an application with global arrays will not decrease as core count is increased. Another approach is to examine the weak scalability of memory usage by fixing the per core domain size. An increase in memory usage for the larger resolution problem is a sign of the presence of global arrays. Figure 2 is a plot of memory usage for a fixed resolution for several different versions of CLM. The reduction in total memory usage for the CLM-mod1 versus the CLM-orig is a result of the reduction in the number of persistent global arrays from approximately 500

to 1. The reduction of excessive global arrays and replicated metadata reduces the memory usage for CLM at high resolution on 512 cores from 2000 to 42 Mbytes, a reduction of a factor of 50.

3.4 Serial I/O

Five of the six applications performed serial I/O. This design decision was likely made when the applications were first refactored for modest levels of distributed memory parallelism. This "triage" was made at the time because 1) parallel I/O to a single file was not supported in the initial implementations of MPI, 2) it is easier to implement and 3) it provided acceptable performance when running on small core counts. Not surprisingly, it becomes problematic when the parallelism of the application is significantly increased. In addition to the creation of a serial performance bottleneck, the serialization of I/O creates several other problems. In particular, it requires the scattering and gathering of data to and from the distributed data representation. The straightforward implementation involves allocating a global-sized array on a single core, which may by itself exhaust memory on a system with limited memory like Blue Gene. Further, the gather/scatter of data may cause MPI to allocate large amounts of buffer memory on the core performing serial I/O. Work with a high resolution (0.1° or 10 km resolution) configuration of POP on Blue Gene revealed that while it was possible to allocate a single global array, the MPI buffer allocation overhead caused the application to fail.

The impact that MPI buffer memory allocation has on overall memory usage is illustrated our by measurements of the memory usage of a high-resolution CCSM configuration on the Cray XT4. This CCSM configuration coupled 0.1° POP and CICE components to 0.5°CAM and CLM. We concentrate on determining memory usage for a single configuration of CCSM where the CICE, CAM, and CPL components executing sequentially on 480 cores, and POP is executing concurrently on a separate set of cores. We estimate each components memory usage by using the resident working set size reported by the Linux /proc file system, and by placing each component on disjoint sets of cores. Note that the MPI buffer space which is a non-separable component of memory usage is easily determined because it is set by environment variable. The results of our analysis for this configuration are shown in Table 2, with the memory usage broken down by component model and that used by MPI buffers. The rather large MPI buffers (53% of the total memory usage shown) are necessary to support gathering/scattering related to serial I/O.

The impact of a single gather of a global array, inherent in a serialized I/O design while acceptable on small core counts, has a profound impact on application memory usage at large core counts, and can even prevent the use of certain types of compute systems.

	CCSM component			MPI	Total
	CAM	CICE	CPL	buffers	
Memory usage (MB)	250	160	29	500	939
percentage	27%	17%	3%	53%	100%

Table 2 Memory usage of the CAM-CICE-CPL part of a high-resolution CCSM configuration running on 480 cores of the Cray XT4.

4 Scalability of Execution Time

We next describe some examples of poor weak scalability discovered within the six applications. While the previous section concentrated on addressing issues associated with allowing codes to run at all, this section concentrates on reducing the cost to run applications.

4.1 Non-scalable initialization

Several of the applications in this study had non-scalable initialization execution times. All applications in this study are typically run for as long as the queueing system will allow, in which case the cost of initialization is amortized across a 6 to 48 hour long job. However, the impact of large initialization costs becomes particularly problematic when performing refactoring or development work, and can serious limit the ability to test an application on very large cores counts. For example, an $\mathcal{O}(P^2)$ initialization algorithm, where P is the number of processors, was discovered in POP when it attempted to run it on more than 10,000 cores. What was a modest cost of a few minutes on smaller cores counts grew to 45 minutes at this scale.

An effective technique for identifying non-scalable initialization is to plot the application's initialization time as a function of core count. Unexpected increases in initialization time at large core counts may indicate an algorithmic problem in the initialization. In POP, the initialization issue turned out to be an $\mathcal{O}(P^2)$ algorithm to calculate the message passing schedule in a single routine: a problem that was easily addressed. The original initialization algorithm contained a nested loop over P tasks that searched for neighbors among a list of P-1 tasks. An alternative $\mathcal{O}(P)$ version that computed and stored each tasks neighbors, which eliminated the $\mathcal{O}(P^2)$ search, was developed and reduced the total initialization time at 10,000 processors from 45 to 10 minutes. Note that at low resolution and core counts the improved $\mathcal{O}(P)$ algorithm only reduced the initialization time from several minutes to 40 seconds versus the $\mathcal{O}(P^2)$ version, a minor improvement that would not have made sense to pursue at low resolutions and core counts.

4.2 Non-scalable inter-processor communication

Scalable and efficient boundary exchange strategies are essential to the successful parallel implementation of many commonly-used numerical methods for solving partial differential equations. However, several of the applications we have studied had scalability issues with their boundary exchange routines. These problems can be classified as either unnecessary message serialization or excessive latency sensitivity. Message serialization is typically the result of the serial treatment of special points, edges, or surfaces found in the underlying grids. Latency sensitivity usually involves sending too many small messages, and becomes critical only at large processor counts.

Both types of problem were discovered in the POP ocean model. POP uses curvilinear displaced-pole grids described by Murray (1996) and Smith et al (1995) to address coordinate singularies at the North Pole. A popular grid variation available in POP is the tripole grid, which provides a quasi-uniform mesh over the Arctic Ocean through the addition of a third pole and a coordinate seam across the Arctic Ocean. An image of the POP tripole grid from Jones (2003) is provided in Figure 3. Because of the complexity of the coordinate seam in the tripole grid at this interface, the boundary exchanges in the parallel implementation of the tripole grid were partially serialized. In particular the POP implementation of the tripole grid duplicate grid points along one logical dimension of the computational grid. POPs update algorithm collects all the duplicated grid points along the tripole boundary on a single core, perform the update coordinate transformation calculation, and then distribute the solution back to all the cores along that edge. Consider the cost to perform a boundary update using a serialized versus a distributed algorithm. For the serialized algorithm, approximately $\sqrt{P}$ cores need to communicate resulting in an $\mathcal{O}(\sqrt{P})$ algorithm. For a distributed algorithm, the cost to update the boundary should be an $\mathcal{O}(1)$, i.e. communicate with one or a small number of neighbors. On 32 cores the serialize algorithm is approximately 6 times as expensive, while on 32,000 it would be 179 times more expensive. While serialization problems in the communication algorithm will show up in both strong and weak scaling experiments, it will be particularly apparent for weak scaling. The elimination of the serialized POP tripole algorithm and replacement with a distributed version reduced the total execution time of POP at 0.1° on a 2000-core Linux cluster with an Infiniband® interconnect by approximately 15%.

POP also contained a latency-sensitive boundary exchange that underscored the importance of message aggregation. A communication routine may be considered latency sensitive if greater than 50% of the cost to send a message is due to latency cost versus bandwidth cost (message size divided by bandwidth). Because the climate community is primarly concerned with strong scalability, the impact of message latency on application performance is important. Message aggregation is a standard technique to reduce the impact latency has on application scalability, albeit at the cost of increased MPI message buffer sizes. For example, POP's finite difference boundary exchange library was designed to work on two dimensional grid objects and did not provide separate subroutines for boundary exchange of 3-

Fig. 3 The tripole grid used by POP. The coordinate seam in the tripole grid connects a pole in the Yukon Territory of Canada to one near Arkhangelsk, Russia.

dimensional (3D) or 4-dimensional (4D) variables. Instead, POP implemented the 3D boundary exchange update as a series of 2-dimensional ones, thus sequentially performing a number of 2D boundary exchange updates equal to the number of levels. In POP, which typically uses 60 vertical levels, this strategy has the potential to incur a substantial latency overhead penalty. By writing boundary exchange subroutines specific for both 3D and 4D variables, this latency overhead is paid once per variable, rather than once per vertical level, thus reducing latency sensitivity.

Ideally it is possible to diagnose problems in the scalability of boundary exchanges through both strong and weak scaling experiments on systems with different latency and bandwidth attributes. For example, POP's lack of message aggregation was discovered by comparing the scalability of its boundary exchanges on the Cray XT and Blue Gene systems. It was observed that the scalability of boundary exchanges within POP on the Cray XT was worse than on Blue Gene, a direct result of the Cray XT's higher message latency versus Blue Gene. Further, an application may only exhibit scalability problems on compute platforms with less capable message passing networks that either the Blue Gene or Cray XT systems. The relative

cost of the boundary exchange may vary widely depending on the precise system and model configuration. In the HOMME dynamical core, the relative cost of boundary exchanges at low resolution ranges from 1-5% to as high as 30-50% of the total cost at very large core counts.

5 Other Impediments

A number of additional impediments were discovered when attempting to significantly increase both the resolution and core counts for these applications. These obstacles include debugging and software development, pre-processing of input, and post-processing of output files. The overarching goal of increased parallelism is to accelerate scientific discovery and thus must therefore involve not only application execution but also a whole host of related tasks.

5.1 Debugging at Scale

Apart from memory and performance issues, when applications are run at high resolutions, bugs emerge. This is to be expected, since we are exercising code in a way that has never been tested when we explore higher process counts. However, we were surprised by the large number of bugs in these six applications that emerged when we increased both resolution and core count. Decomposition issues on large core counts were particularly common. Other bugs were discovered in options that engaged model physics not generally used in lower resolution configurations. Experience debugging these applications has led us to make several best practice recommendations. First, the domain decomposition strategies of applications should be subjected to rigorous testing at a variety of scales. Unfortunately, the majority of the domain decomposition bugs would not have been caught using a simplified unit test methodology, but rather involve testing of the entire system by comparing the results obtained using one decomposition to another. Second, while the number of possible combinations of physics options, resolutions, and core count available in scientific applications is daunting, routine testing is critical. Third, routine access to multiple, very large systems is a critical requirement for the testing of applications in order to spot problems introduced during development. In our experience, most large systems, because of their cost, are competitively allocated on scientific merit. As a result, resource providers and scientific users of large systems tend to neglect performance and validation testing. Fourth, successful debugging of large, parallel scientific applications not only requires routine access to large systems, but also routine interactive access, or at the very least, batch access with rapid turn around time. For example, addressing several bugs in POP required daily access to 2,048 cores for a total of a week. A concerted debugging effort of a high-resolution CCSM configuration involved regular access to 1800 cores, and consumed 600,000 CPU

hours in several months time. The development work associated with highly scalable scientific applications frequently require substantial resources. The best way to minimize these costs is to reduce debugging costs through routine testing at a variety of processor counts scales and resolutions.

5.2 Pre/Post Processing

The ultimate goal of our refactoring applications for performance is to speed up scientific progress. It does little good to increase the performance of the model, if the tools around it slow down or cease to function. Thus, one must examine the entire scientific workflow for bottlenecks. As in the case of applications themselves, CCSMs pre/post analysis tools were designed to deal with low-resolution simulations executed on small processor counts, indeed many are still serial.

An illustrative example is the serial generation of a river runoff input file for an ultra-high resolution CCSM run. At low resolution, the calculation took 3 hours and 2 GBytes of memory. Since the application only needed to be run once, these requirements were acceptable. However, when applied to a high-resolution CCSM configuration, we discovered that the algorithm for generating a river runoff mapping from the 0.5° land model to the ocean would have taken 60 days to execute on a single core, and would have required 128 Gbytes of memory. The existing river runoff algorithm was rewritten by replacing the local search algorithm and by limiting its calculation to be near the coastline. The redesign (still serial) reduced the required resources to 30 minutes on a system with 5 Gbytes of memory for the high resolution case.

The attempts to analyze a 100-year high-resolution coupled simulation has recently highlighted the lack of scalability within CCSMs post processing workflow and the impact it has on scientific discovery. While we are now able to simulate high-resolution climate at approximately 2 simulated years per day, we are certainly not able to analyze the approximately 3.5 TB of history files per week it generates. A number of the standard post-analysis scripts had to be rewritten to eliminate excessive memory usage in order to even execute successfully. Our inability to analyze this data at a rate commensurate to its generation stems from the fact that most of the analysis processing is serial.

6 Conclusions

We have described several common issues gleaned from experience refactoring six scientific computing applications for efficient execution on very large-scale computing systems. The six applications in total represent over a million lines of code that have been developed by multiple scientific communities over the last 25 years. Fortunately, improving their scalability involved addressing a small number of sim-

ilar issues in $\sim$1% of the source code. Unfortunately, there is no simple solution, or magic bullet. Refactoring, testing and debugging complex scientific applications on large numbers of processors is inherently difficult. However, there is an approach that has been remarkable effective on all six applications that we have worked with. All the problems were discovered by systematically stress testing each application, that is, testing the applications in ways that they have never before been tested, and then systematically studying the way in which they behave or fail. This approach requires routine, and in the case of debugging, rapid turn-around access to very large scalable, parallel systems, and a commitment of resources for this purpose. Routine application testing regimens can then ensure that these issues do not recur.

Our approach of stress testing while refactoring revealed that, despite the unique origin for most of the applications studied, the issues discovered were strikingly similar. At a high level the impediments involved non-scalable memory usage and execution time. Non-scalable memory usage issues were discovered that actually limited the type of science questions that could be asked. However, we also found that a large number of these problems could be addressed by modifying a small amount of source code. All of these issues were the results of design choices made years ago for a single range of resolution and processor counts, and for which they had little impact. The scalability implications of these choices were either ignored or deferred. It is a cautionary tale as we contemplate the exascale systems with millions of processors that loom on the horizon.

Acknowledgements

We would like to thank our colleagues Mariana Vertenstein, Tony Craig for all their work addressing the many code design issues discovered during this study. We would like to thank Dr. Mark Taylor for running several of the applications on compute platforms at Sandia National Laboratory, and Lawrence Livermore National Laboratory. We also thank Brookhaven National Laboratory, and Oak Ridge National Laboratory for access to their large compute platforms. We thank Fred Mintzer for access to the Thomas J. Watson Research facility through the 2^{nd} and 3^{rd} Blue Gene Watson Consortium Days event. Significant computational resources were provided through grants by the LLNL 2^{nd} and 3^{rd} Institutional Grant Challenge program. Code development would not have been possible without the access to the Blue Gene system at NCAR, which is funded through NSF MRI Grants CNS-0421498, CNS-0420873, and CNS-0420985 and through the IBM Shared University Research (SUR) Program with the University of Colorado.

The work of these authors was supported through National Science Foundation Cooperative Grant NSF01 which funds the National Center for Atmospheric Research (NCAR), and through the grants: #OCI-0749206 and #OCE-0825754. Additional funding is provided through the Department of Energy, CCPP Program Grant #DE-PS02-07ER07-06.

References

Adiga NR, et al (2002) An overview of the Blue Gene/L supercomputer. In: Proceedings of SC2002, Baltimore, MD

Baker AH, Falgout FD, Yang UM (2006) An assumed partition algorithm for determing processor inter-communication. Parallel Computing 32:394–414

Collins WD, Rasch P, Boville BA, Hack J, McCaa J, Williamson DL, Bitz BPBCM, Lin SJ, Zhang M (2006) The formulation and atmospheric simulation of the Community Atmosphere Model version 3 (CAM3). Journal of Climate 19(11):2144–2161

Dennis JM (2007) Inverse space-filling curve partitioning of a global ocean model. In: IEEE International Parallel & Distributed Processing Symposium, Long Beach, CA

Dennis JM, Tufo HM (2008) Scaling climate simulation applications on IBM Blue Gene. IBM Journal of Research and Development: Applications for Massively Parallel Systems 52(1/2)

Dennis JM, Fournier A, Spotz WF, St-Cyr A, Taylor MA, Thomas SJ, Tufo H (2005) High resolution mesh convergence properities and parallel efficiency of a spectral element atmospheric dynamical core. Int J High Perf Comput Appl 19:225–235

Hoffman FM, Vertenstein M, Kitabata H, III JBW (2005) Vectorizing the community land model. International Journal of High Performance Computing Applications 19:247–260

Hunke EC, Lipscomb WH (2008) CICE: the Los Alamos sea ice model documenation and software user's manual version 4.0. Tech. Rep. LA-CC-06-012, Los Alamos National Laboratory, T-3 Fluid Dynamics Group

Jacob R, Larson J, Ong E (2005) MxN communication and parallel interpolation in CCSM3 using the Model Coupling Toolkit. Int J High Perf Comp Appl 19(3):293–307

Jones P (2003) Parallel Ocean Program (POP) user guide. Tech. Rep. LACC 99-18, Los Alamos National Laboratory

Jones PW, Worley PH, Yoshida Y, White JBI, Levesque J (2005) Practical performance portability in the Parallel Ocean Program (POP). Concurrency Comput Prac Exper 17:1317–1327

Larson J, Jacob R, Ong E (2005) The Model Coupling Toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. Int J High Perf Comp App 19(3):277–292

Murray RJ (1996) Explicit generation of orthogonal grids for ocean models. J Comp Phys 126:251–273

OpenMP (2005) OpenMP application programing interface. http://www.openmp.org/

RedStorm (2006) The Cray XT3 Supercomputer. Http://www.cray.com/products/xt3/index.html

Shingu S, Y T, Ohfuchi W, Otsuka K, Takahara H, Hagiwara T, Habata S, Fuchigami H, Yamada M, Sasaki Y, Kobayashi K, Yokokawa M, Itoh H (2002) A 26.58 Tflops global atmospheric simulation with the spectral transform method on the

earth simulator. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pp 1–19

Smith R, Kortas S, Meltz B (1995) Curvilinear coordinates for global ocean models. LANL Technical Report LA-UR-95-1146

Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (2000) MPI: The Complete Reference: Volume 1, The MPI Core. The MIT Press

Solomon S, Qin D, Manning M, Chen Z, Marquis M, Tignor KAM, Miller H (eds) (2007) Contribution of Working Group 1 to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change. Cambridge University Press, Cambridge United Kingdom and New York, NY, USA

Taylor MA, Edwards J, Cyr AS (2008) Petascale atmospheric models for the community climate system model: New developments and evaluation of scalable dynamical cores. J Phys Conf Ser 125

Walkup B (2007) Personal Communication