Final Report

Center for Programming Models for Scalable Parallel Computing*

Co-Array Fortran for Terascale Clusters

Cooperative Agreement No. DE-FC03-01ER25504

John Mellor-Crummey

Principal Investigator at Rice University

Department of Computer Science, MS 132
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Voice: 713-348-5179
FAX: 713-348-5930
Email: johnmc@cs.rice.edu

*Multi-institutional center Project Director: Rusty Lusk, ANL, lusk@mcs.anl.gov

1 Achievements

Rice University's achievements as part o the Center for Programming Models for Scalable Parallel Computing include

- design and implemention of `cafc`, the first multi-platform CAF compiler for distributed and shared-memory machines,
- performance studies of the efficiency of programs written using the CAF and UPC programming models,
- a novel technique to analyze explicitly-parallel SPMD programs that facilitates optimization,
- design, implementation, and evaluation of new language features for CAF, including communication topologies, multi-version variables, and distributed multithreading to simplify development of high-performance codes in CAF, and
- a synchronization strength reduction transformation for automatically replacing barrier-based synchronization with more efficient point-to-point synchronization.

The prototype Co-array Fortran compiler `cafc` developed in this project is available as open source software from `http://www.hipersoft.rice.edu/caf`.

1.1 Design, implementation, and performance evaluation of `cafc`

We designed and implemented `cafc`, the first multi-platform CAF compiler for distributed and shared-memory architectures. `cafc` is a widely portable source-to-source translator. By performing source-to-source translation, `cafc` can leverage the best Fortran 95 compiler available on the target architecture to compile translated programs, and the ARMCI and GASNet communication libraries to support systems with a range of interconnect fabrics, including Myrinet, Quadrics, and shared memory.

We ported many parallel benchmarks into CAF and performed extensive evaluation studies [1, 5, 6, 2, 3, 4] to investigate the evaluate the CAF programming model and its ability to deliver high performance. As a result of our studies and the concomitant refinements of the `cafc` code generator, carefully-written CAF codes compiled with `cafc` can now match the performance and scalability of their MPI counterparts. As part of our studies, we identified three classes of performance impediments that initially precluded CAF codes from achieving the same level of performance and scalability as that of their MPI counterparts: scalar performance of a translated program, communication efficiency, and synchronization.

Scalar performance. Natural source-to-source translation of co-arrays in CAF into Fortran 90/95 introduces apparent aliasing in the translated program due to `cafc`'s representation of co-arrays using Fortran 90 pointers. This hinders the platform's Fortran 95 compiler

to efficiently optimize code accessing local co-array data. We developed a *procedure splitting transformation* that converts each procedure s referencing `COMMON` and `SAVE` co-array local data into two subroutines s_1 and s_2. s_1 resembles s, but instead of performing computation, it calls s_2 and passes the co-arrays as arguments. s_2 performs the original computation in which each `COMMON` and `SAVE` co-array reference is converted into a reference to the corresponding co-array parameter. In `cafc`, co-array arguments are represented via explicit shape subroutine dummy arguments, which do not alias in Fortran 95. As a result, the lack of aliasing among `COMMON` and `SAVE` co-arrays, their bounds and contiguity are conveyed to the Fortran 95 compiler. The procedure splitting transformation implemented in `cafc` enables a translated program to achieve the same level of scalar performance as an equivalent Fortran 95 program that uses `COMMON` and `SAVE` variables.

Communication efficiency. Our experiments showed that it is imperative to vectorize and/or aggregate communication on distributed memory machines to deliver performance and scalability. Without coarse-grain communication, CAF programs perform abysmally on cluster architectures. An advantage of CAF over UPC is that it can express vectorization and aggregation at source level without calls to bulk library primitives. Communication vectorization yielded benefits as high as 30% on Myrinet cluster architectures. When writing CAF communication, the Fortran 95 array sections enable a programmer to conveniently express communication of strided data.

Our experiments also showed that even when using communication libraries that support efficient non-contiguous strided communication, it is beneficial to perform *communication packing* of strided data at source level, sending it as contiguous message, and unpacking it at its destination. We found that one-sided communication aggregation using active messages is less efficient than library-optimized strided communication transfers because libraries such as ARMCI can overlap packing of communication chunks at the source, communication of strided chunks and unpacking of chunks on the destination. Communication packing at source level boosted performance about 30% for both CAF and UPC on clusters, but yielded minuscule benefits on shared memory platforms. Our findings apply also to other partitioned global address space languages such as UPC: we showed that communication packing for a UPC version of BT yielded improvements as high as 30% on an Itanium2+Myrinet cluster.

To give a CAF programmer the ability to overlap computation and communication, we extended CAF with *non-blocking communication regions*. Skilled CAF programmers can use pragmas to specify the beginning and the end of regions in which all communication events are issued by using non-blocking communication primitives, assuming the underlying communication library provides them. Using these regions enabled us to improve the performance of NAS BT by up to 7% on an Itanium+Myrinet2000 cluster.

Synchronization. The burden that PGAS languages impose on programmers is the need to synchronize shared one-sided data access. We observed that using barriers for synchronization was much simpler than using point-to-point synchronization, which is painstaking and error-prone. However, point-to-point synchronization may provide much better scalabil-

ity; we observed up to a 51% performance improvement for the NAS CG benchmark (14000 size) for a 64-processor execution.

We observed that using extra communication buffers can shorten the critical path by removing anti-dependence synchronization needed to coordinate communication buffer reuse. This yielded up to a 12% performance improvement for the ASCI Sweep3D benchmark (150x150x150 size) as compared to the standard MPI version. However, coding such multi-buffer solutions is difficult due to the need for explicit buffer management and complex point-to-point synchronization.

CAF and UPC. We also compared CAF with UPC for regular scientific codes (principally the NAS benchmarks) and found that it is easier to match MPI's performance with CAF. We attribute this to the more explicit nature of communication in CAF and language-level support for multi-dimensional arrays.

1.2 Scalability analysis

To improve parallel performance of CAF programs or programs written using other SPMD programming models, one needs to determine the impediments to scalability. To understand how scalability bottlenecks arise, one needs to analyze them within the calling context in which they occur. This enables program analysis at multiple levels of abstraction: we could choose to analyze the cost of user-level routines, user-level communication abstractions, compiler runtime primitives, or the implementation of the underlying communication library (*e.g.*, ARMCI, GASNet, or MPI).

Users have certain *performance expectations* of their codes. For strong scaling parallel applications users expect that their execution time decreases linearly with the number of processors. For weak scaling applications, they expect that the execution time stays constant while the number of processors increases and the problem size per processor remains constant. Our goal was to develop an efficient technology that *quantifies* how much a certain code deviates from the performance expectations of the users, and then quickly *guides* them to the scaling bottlenecks.

Our approach has several steps. First, users are required to formally define their expectations, such as linear scaling for strong scaling programs or constant execution time for weak scaling applications. Second, we measure the program during executions on different number of processors. Third, we compute how much each node in the call tree deviates from the performance expectation; we proposed an intuitive metric, called *relative excess work*. Finally, we use an interactive viewer that assists a user in identifying and navigating to the scaling hot spots. We developed an intuitive metric for analyzing the scalability of application performance based on excess work.

We used this scaling analysis methodology to analyze the parallel performance of MPI, CAF, and UPC codes. A major advantage of our scalability analysis method is that it is effective regardless of the SPMD programming model, underlying communication library, processor type, application characteristics, or partitioning model. We plan to incorporate our scaling analysis into HPCToolkit, so it would be available on a wide range of platforms.

Our scaling study pointed to several types of problems. One performance issue we identified using our scalability analysis was the inefficiency of user-level implementation of reductions in both CAF and UPC codes. A drawback of source-level user-implemented reductions is that they introduce performance portability problems. The appropriate solution is to use language-level or library implementations of reductions, that can be tuned offline to use the most efficient algorithms for a particular platform. An obstacle to scalability for CAF codes was a blocking implementation of the `notify` synchronization primitive using the ARMCI and GASNet communication libraries. Finally, for both CAF and MPI applications we found that some codes performed successive reductions on scalars; the natural remedy for that is to perform aggregation of reductions by using the appropriate vector operations. An important result was that the relative excess work metric readily identified these scalability bottlenecks.

The scaling analysis of CAF codes indicated the urgency of language-level support for collective operations. Consequently, we explored and evaluated collective operations extensions to the CAF model and presented an implementation strategy based on the MPI collectives. For the NAS MG benchmark, using the language-level collectives led to a reduction of the initialization time by 60% on 64 processors, and led to a reduction of the measured running time for LBMHD of 25% on 64 processors.

1.3 Enhanced language, compiler, and runtime technology for CAF

Co-spaces: communication topologies for CAF. CAF's multi-dimensional co-shape is not convenient and expressive enough to be useful for organizing parallel computation. It does not provide support for process groups, group communication topologies, nor expression of communication partners relative to the process image. Instead, programmers often use Fortran 95 arrays and integer arithmetic to represent communication partners. Such ad hoc methods of structuring parallel computation render CAF impenetrable to compiler analysis.

We explored replacing CAF's multi-dimensional co-shapes with more expressive communication topologies, called co-spaces, such as group, Cartesian, and graph. They simplify programming by providing convenient abstractions for organizing parallel computations. Group co-spaces enable support for process groups as well as remapping process image indices. Cartesian or graph co-spaces are used to impose a Cartesian or graph communication topology on a group; they provide functionality to systematically specify the targets of communication and point-to-point synchronization. These abstractions, in turn, expose the structure of communication to the compiler, facilitating compiler analysis and optimization.

Communication analysis. We devised a novel technology for analyzing *explicitly-parallel* CAF programs suitable for a large class of scientific applications with structured communication. When parallel computation is expressed via a combination of a co-space, textual co-space barriers, and co-space single-valued expressions, the CAF compiler can infer communication patterns from explicitly-parallel code. At present, communication analysis is limited to a procedure scope with structured control flow. Our work focuses on two patterns that are common for nearest-neighbor scientific codes. The first pattern is a group-executable

PUT/GET in which the target image is expressed via a co-space interface neighbor function with co-space single-valued arguments. The second is a non-group-executable PUT/GET with the target image expressed via a co-space interface neighbor function with co-space single-valued arguments. Knowing the communication pattern for each process image of the co-space enables determination of the origin image(s) of communication locally. This is a fundamental enabling analysis for powerful communication and synchronization optimizations such as synchronization strength reduction.

Synchronization strength reduction (SSR). We developed a procedure-scope synchronization strength reduction optimization that replaces textual co-space barriers with asymptotically more efficient point-to-point synchronization where legal and profitable. This transformation is both difficult and error-prone for application developers to exploit manually at the source code level. SSR optimizes the communication patterns inferred by our analysis of communication partners. As of now, it operates on a procedure scope with a single co-space and textual co-space barriers for synchronization. To extend SSR's applicability to real codes, we use compiler hints to compensate for the lack of interprocedural analysis. Understanding communication structure enables the CAF compiler to convert barrier-based synchronization into more efficient form. We investigated the conversion of textual co-space barriers into point-to-point synchronization. SSR-optimized programs are more asynchrony tolerant and show better scalability and higher performance than their barrier-based counterparts.

We implemented prototype support for SSR in a version of `cafc`. SSR-optimized Jacobi iteration, NAS MG, and NAS CG benchmarks show performance comparable to that of our fastest hand-optimized versions that use point-to-point synchronization. Compared to their barrier-based counterparts, they demonstrate noticeable performance improvements. For 64-processor executions on an Itanium2 cluster with a Myrinet 2000 interconnect, we observed run-time improvements of 16% for a 2D Jacobi iteration of 1024^2 size, 18% for NAS MG classes A and B, and 51% for NAS CG class A. In our prior studies, we observed similar benefits from using point-to-point synchronization instead of barriers on other parallel platforms and for other benchmarks as well.

Multi-version variables. Many scientific codes such as wavefront, line-sweep, and loosely-coupled parallel applications exhibit the producer-consumer communication pattern, in which the producer(s) sends a stream of values to the consumer(s). Expressing high performance producer-consumer communication in PGAS languages is difficult. The programmer has to explicitly manage several communication buffers, orchestrate complex point-to-point synchronization (to hide the latency of anti-dependence synchronization due to buffer reuse), and use non-blocking communication.

We explored extending CAF with multi-version variables (MVVs), a language-level abstraction we devised to simplify the development of high performance codes with producer-consumer communication. An MVV can store more than one value. Only one value can be accessed at a time; others are queued by the runtime. A producer commits new values into

an MVV and a consumer retrieves them. MVVs offer limited support for two-sided communication in CAF, which is a natural choice when developing producer-consumer codes. MVVs simplify program development by insulating the programmer from the details of buffer management, complex point-to-point synchronization, and non-blocking communication.

MVVs are the right abstraction for codes in which each process communicates streams of values to a small subset of processors. MVVs might not be the best abstraction for codes in which each process communicates data to a lot of processes, which might cause excessive MVV buffering. While MVVs insulate the programmer from managing the anti-dependence synchronization, sometimes no such synchronization is necessary because it is enforced elsewhere in the application. However, we believe that programmability benefits of the MVV abstraction outweigh slight performance losses due to unnecessary anti-dependence synchronization in this case.

We extended CAF with prototype support for MVVs. MVVs significantly simplify development of wavefront applications such as Sweep3D, and MVV-based codes deliver performance comparable to that of the fastest CAF multi-buffer hand-optimized versions, up to 39% better than that of CAF one-buffer versions, and comparable to or better (up to 12%) than that of their MPI counterparts on a range of parallel architectures. MVVs greatly simplify coding of line-sweep applications, such as the NAS BT and SP benchmarks, and deliver performance comparable to that of the best hand-optimized MPI and CAF versions.

Distributed multithreading. Distributed memory is necessary for the scalability of massively parallel systems. Systems in which memory is co-located with processors continue to dominate the architecture landscape. The nodes of these distributed memory architectures are also becoming parallel, e.g., multi-core multiprocessors. Distributed multithreading (DMT) is based on the concepts of function shipping and multithreading, which provide two benefits. First, DMT enables co-locating computation with data. Second, it enables exploiting hardware threads available within a node. DMT uses *co-subroutines* and *co-functions* to co-locate computation with data and to enable local and remote asynchronous activities. Using DMT to co-locate computation with data is an effective way of avoiding exposed latency, especially when performing complex operations on remote data structures. In addition, concurrent activities running within a node would enable utilizing available hardware parallelism.

We presented design principles behind multithreading in an SPMD language and provided the DMT specification for CAF, featuring blocking and non-blocking activities that can be spawned remotely or locally. We extended `cafc` with prototype support for DMT. We developed a micro-benchmark to compute the maximum value of a co-array section to quantify the performance gain due to co-locating computation with data. In our experiments on an Itanium2 cluster with a Myrinet 2000 interconnect, we observed that, for large sections, it is up to 40 times faster to ship computation and get the result back than fetch data and obtain the result locally; for accesses to more complex remote data structures, this benefit is likely to be much higher. We developed several fine-grain and bucketed versions of the RandomAccess benchmark to gain a better understanding for DMT design. Our ex-

perimentation revealed that it is necessary to use a pool of OS threads to execute activities, rather than to spawn each activity in a separate OS thread, to deliver best performance; it is also necessary to allow programmers to control the thread pool to tune the runtime for the application's concurrency needs. Better asynchrony tolerance allowed the performance of a DMT-based implementation of bucketed RandomAccess to exceed that of the standard MPI version, which uses `MPI_AlltoAll` to exchange remote updates.

We found that DMT improves programmability of applications that benefit from asynchronous activities. We experimented with a branch-and-bound traveling salesman problem (TSP), which we selected as representative of parallel search applications. We found that the DMT-based CAF version is simpler than a master-slave message-passing implementation in MPI. The simplicity comes from not having to implement a two-sided protocol when using DMT; instead, the programmer can use co-functions to execute asynchronous remote activities. DMT-based TSP demonstrates better performance, because, in our experiments, the MPI implementation dedicates a processor to be the master, and this mater processor does not perform useful computation.

2 Future Directions

While the research and development of both the CAF language and compiler technology for PGAS languages has demonstrated that CAF can be used to achieve high performance on clusters, significant additional work is needed before CAF an attractive technology for computational scientists. Several issues need significant attention.

Implementation issues. The ideas explored in the development of the `cafc` compiler need have proven sound, but the implementation needs several enhancements to improve programmer productivity. First, `cafc` needs support for co-array variables as part of Fortran 90/95 modules. Second, `cafc` needs enhanced support for inheriting implicit procedure interfaces for procedures that manipulate co-arrays.

Enhanced support for manipulating remote data. Currently, CAF supports an MPI-like model for SPMD programming. For CAF to support a broader range of application styles, it needs better support for manipulation of remote data and more flexible data-oriented synchronization. This is a focus of work in the PModels2 project.

Improving thread support in PGAS languages. Co-locating computation with data and utilizing intra-node parallelism is essential to fully utilize hardware capabilities of modern parallel architectures. While experimenting with distributed multithreading, we discovered that operating systems do not provide adequate support for precisely controlling multithreading for high performance codes. A promising research direction is to work with OS developers to develop an efficient, flexible, and portable threading system that enables applications, rather than the OS, to schedule threads. This would enable us to extend a multithreaded programming model with user-defined scheduling policies that best accommodate

the concurrency needs of the application, as well as compiler analysis and optimization to appropriately mix & schedule concurrent computations. Better run-time support would also be necessary to enable massive (millions of threads) multithreading within a node.

It is worth investigating whether a programming model can provide convenient abstractions for efficient work-sharing that can be optimized for automatic load-balancing in the presence of distributed memory.

Enhancing scalability analysis. For our scalability analysis technique to be broadly useful, we need more widely available support for measurement of the call path profiles it requires. Developing support for collecting such profiles on the leadership-class platforms is a focus of Rice University's work as part of the Performance Engineering Research Institute. Developing more usable support for scalability analysis of parallel profiles is a focus of research and development of performance tool interfaces in the Center for Scalable Application Development Software.

Extending CAF analysis and communication/synchronization optimization. To develop scalable, high performance explicitly-parallel programs, programmers must use efficient communication and orchestrate complex point-to-point synchronization, which is difficult. Barriers are the simplest synchronization mechanism to use in PGAS languages. Thus, the role of the compiler is to enable application developers to use barriers for synchronization, while optimizing communication and synchronization into a more efficient form delivering performance and scalability. SSR is an example of such an optimization.

Today, our novel CAF analysis and SSR are limited to a procedure scope with single co-space and structured control flow. It is possible to extend the analysis to handle arbitrary control flow. There is also a good indication that interprocedural analysis can be developed to eliminate the necessity of hints for SSR. Such analysis would include: (1) detecting whether a procedure may access local or remote co-arrays or perform synchronization in any invocation; (2) propagation of single values across procedure calls; (3) propagation of unsynchronized PUT/GET across procedure boundaries. It is still an open question whether an analysis can be developed to analyze scopes where communication/synchronization is done for multiple co-spaces.

In addition to SSR, our CAF analysis technology enables a set of promising communication and synchronization optimizations. SSR does not change the communication primitive. Doing so will enable conversion of one-sided PUT/GET communication into two-sided send and receive. Such two-sided communication can be buffered, and would enable us to automatically generate more asynchrony tolerant code, since buffering can move anti-dependence synchronization off the critical path, and packing/unpacking of strided communication. Conversion of GET into PUT will enable us to utilize interconnect RDMA capabilities, when accessing remote data via PUTs, for architectures with RDMA support for PUTs, but not for GETs. The push (PUTs) strategy would also enable us to hide exposed latency inherent to the pull (GETs) strategy as well as to tile producer-consumer loop nests to entirely hide communication latency.

Finally, our SSR algorithm is not based on array section dependence analysis. Developing such an analysis, which must also include remote co-array sections, might improve the precision of our CAF analysis and SSR; however, we have not yet seen opportunities that would benefit from such analysis in the limited set of codes we have studied.

Enhancing multi-version variables and beyond. Producer-consumer communication is typical in many scientific codes; however, it is difficult to develop scalable, high performance producer-consumer applications in PGAS languages. We offer MVVs as a pragmatic and convenient way to simplify development of high-performance producer-consumer codes in CAF.

It would be interesting to consider whether multi-version variables can benefit from extensions such as GET-style remote `retrieve`, the `commit` and `retrieve` primitives of partial MVV versions, and an adaptive buffer management strategy.

It is worth investigating the stream abstraction as an alternative to MVVs, especially for codes that stream values of unequal size. While streams are a more general abstraction than MVVs, they would require the programmer to establish explicit connections. For streams, it would also be harder to optimize unnecessary memory copies, which MVVs achieve via adjusting an F90 pointer.

The clocked final model (CF) of X10 is another more general alternative to MVVs that does not require the programmer to specify the number of buffers and explicitly manage `commit`s and `retrieve`s. It would be interesting to investigate whether it is possible to develop sophisticated compiler and runtime technology to optimize CF-based scientific codes to deliver as high performance as that of using MVVs on a range of parallel architectures.

3 Project Personnel

Faculty, staff, and students that were supported in part by this cooperative agreement include

- John Mellor-Crummey, Principal Investigator
- Fengmei Zhao, Research programmer
- Daniel Chavarria-Miranda, Post-doctoral researcher
- Yuri Dotsenko, Ph.D. Candidate
- Cristian Coarfa, Ph.D. Candidate

This project was able to significantly leverage the products of other ongoing work on performance analysis tools and enhancement of the Open64 compiler infrastructure.

Publications

[1] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran performance and potential: An NPB experimental study. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS, College Station, TX, Oct. 2003. Springer-Verlag.

[2] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. In *Proceedings of the Los Alamos Computer Science Institute Fifth Annual Symposium*, Santa Fe, NM, Oct. 2004. Distributed on CD-ROM.

[3] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. *Journal of Supercomputing*, 36(2):101–121, May 2006.

[4] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and Chavarría-Miranda. An evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[5] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multiplatform Co-Array Fortran compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, Sept.–Oct. 2004.

[6] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on hardware shared memory platforms. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, IN, Sept. 2004.