Scalability Analysis of SPMD Codes Using Expectations

Cristian Coarfa
Baylor College of Medicine
One Baylor Plaza
Houston, TX 77030
coarfa@bcm.edu

John Mellor-Crummey
Rice University
6100 Main St.
Houston, TX 77005
johnmc@cs.rice.edu

Nathan Froyd
CodeSourcery
9978 Granite Point Ct.
Granite Bay, CA 95746
froydnj@codesourcery.com

Yuri Dotsenko
Rice University
6100 Main St.
Houston, TX 77005
dotsenko@cs.rice.edu

ABSTRACT

We present a new technique for identifying scalability bottlenecks in executions of single-program, multiple-data (SPMD) parallel programs, quantifying their impact on performance, and associating this information with the program source code. Our performance analysis strategy involves three steps. First, we collect call path profiles for two or more executions on different numbers of processors. Second, we use our expectations about how the performance of executions should differ, *e.g.*, linear speedup for strong scaling or constant execution time for weak scaling, to automatically compute the scalability of costs incurred at each point in a program's execution. Third, with the aid of an interactive browser, an application developer can explore a program's performance in a top-down fashion, see the contexts in which poor scaling behavior arises, and understand exactly how much each scalability bottleneck dilates execution time. Our analysis technique is independent of the parallel programming model. We describe our experiences applying our technique to analyze parallel programs written in Co-array Fortran and Unified Parallel C, as well as message-passing programs based on MPI.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, performance attributes.

General Terms

Performance, measurement, languages, algorithms.

Keywords

Performance analysis, performance visualization, parallel programming models, HPCToolkit.

1. INTRODUCTION

Computational modeling and simulation have become indispensable tools for scientific research. The desire to tackle grand challenge problems is driving the development of petascale systems composed of tens of thousands of processors. To exploit the power of such machines, parallel applications must scale efficiently to large numbers of processors. However, writing and tuning complex applications to achieve scalable parallel performance is hard.

Understanding a parallel code's impediments to scalability is a necessary step for improving its performance. Often, an application's scalability bottlenecks are not obvious. They can arise from a range of causes including replicated work, data movement, synchronization, load imbalance, serialization, and algorithmic scaling issues. Having an automatic technique for identifying scalability problems would boost development-time productivity.

When analyzing an application's scaling bottlenecks, one should focus on those that are the most significant. An application's components with the worst scaling behavior are often not the most significant scaling bottlenecks for the application as a whole. For instance, a routine that displays abysmal scaling but consumes only a fraction of a percent of the total execution time is less important than a routine that is only a factor of two from ideal scaling but accounts for nearly 50% of the total execution time on large numbers of processors. For developers to tune applications for scalable performance, effective tools for pinpointing scalability bottlenecks and quantifying their importance are essential.

This paper describes a new approach for identifying scalability bottlenecks in executions of single-program, multiple-data (SPMD) parallel programs, quantifying their impact on performance, and associating this information with the program source code. Our approach involves three steps.

First, we collect call path profiles for two or more executions of unmodified, fully-optimized application binaries on different numbers of processors. Second, we use our expectations about how costs should differ between executions to compute how the costs incurred in each dynamic context affect a program execution's scalability. To help developers understand how performance bottlenecks arise, we annotate the hierarchy of dynamic contexts in a program execution with their associated scalability scores. Third, we provide application developers with an interactive browser that enables one to explore scalability bottlenecks in a top-down fashion, see the contexts in which poor scaling behavior arises, see the source code lines that fail to deliver scalable performance, and understand exactly how much each scalability bottleneck dilates execution time. Our approach to scalability analysis

- scales to large numbers of processors by using concise profiles instead of traces,
- uses only local information when measuring performance, yet can pinpoint *and quantify* the scalability impact of bottlenecks caused by communication,
- works on unmodified fully-optimized binaries,
- is programming model and problem independent—it can identify scalability problems in any SPMD program regardless of their cause (*i.e.*, computation, data movement, or synchronization), and
- provides quantitative feedback about exactly how each bottleneck affects scaling.

The rest of the paper presents our approach and explores its utility for analyzing parallel program executions. Section 2 reviews related work. Section 3 describes our analysis methodology. Section 4 describes our experiences applying this technique to analyze the performance and scalability of parallel programs written in the partitioned global address space languages Co-array Fortran (CAF) [32] and Unified Parallel C (UPC) [7], as well as message-passing programs that use MPI [37]. Section 5 summarizes our conclusions.

2. RELATED WORK

Trace-based tools for measuring parallel performance [29, 17, 48, 46, 8, 44, 27] can detail how an execution unfolds over time. However, on large-scale systems tracing can be costly and produce massive trace files [42]. In contrast, performance measurement approaches that collect summaries based on synchronous monitoring (or sampling) of library calls (*e.g.*, [42, 43]) or profiles based on asynchronous events (*e.g.*, [3, 12, 21]) readily scale to large systems because they yield compact measurement data on each processor; the size of this data is largely independent of execution time.

Synchronous monitoring of communication calls, *e.g.*, by mpiP [43] or Photon [42], yields information about communication activity but not computation. We use timer-based call path profiling [15, 16], which attributes all costs in a parallel execution (*e.g.*, computation, data movement, or waiting) to the full calling contexts in which they are incurred. For parallel programs that use library-based communication such as MPI, this is essential for understanding how costs incurred in the MPI library relate back to the application code. Although our call path profiler has novel aspects that make it more accurate and lower overhead than other call graph or call path profilers, a detailed comparison of our call path profiler with others is beyond the scope of this paper and can be found elsewhere [15, 16].

Performance measurement tools require different amounts of knowledge at run time. Quartz [3], designed for shared-memory multiprocessors, uses *global knowledge* about the amount of instantaneous parallelism to scale the cost of samples. Measurement systems requiring global knowledge are problematic for monitoring large-scale systems. Photon [42] tags sampled messages with a time stamp and an identifier specifying the source-code context that initiated the message; at the receiver, message time stamps provide *non-local knowledge* and yield insight into communication performance. This technique requires custom communication libraries with integrated instrumentation. mpiP [43], which profiles MPI functions, uses only *local knowledge*, as does our approach of using call path profiling independently on individual nodes. Measurement techniques requiring only local knowledge are the most scalable.

Tools for measuring parallel application performance are typically model dependent, such as libraries for monitoring MPI communication (*e.g.*, [47, 42, 43]), interfaces for monitoring OpenMP programs (*e.g.*, [8, 26]), or global address space languages (*e.g.*, [38]). In contrast, our approach of using call path profiling is model independent.

Performance tools also differ with respect to their strategy for instrumenting applications. Tau [27], OPARI [26], and Pablo [34] among others add instrumentation to source code during the build process. Model-dependent strategies often use instrumented libraries [8, 22, 23, 25, 42]. Other tools analyze unmodified application binaries by using dynamic instrumentation [6, 13, 24] or library preloading [12, 28, 15, 36, 19]. Our call path profiler currently uses preloading to monitor unmodified dynamically-linked binaries.

Tools for analyzing bottlenecks in parallel programs are typically *problem focused*. Paradyn [24] uses a performance problem search strategy and focused instrumentation to look for well-known causes of inefficiency. Strategies based on instrumentation of communication libraries, such as Photon and mpiP, focus only on communication performance. Vetter [41] describes an assisted learning based system that analyzes MPI traces and automatically classifies communication inefficiencies, based on the duration of primitives such as blocking and nonblocking send and receive. EXPERT [45] also examines communication traces for patterns that correspond to known inefficiencies. In contrast, our scaling analysis is *problem-independent*.

Performance analysis tools analyze scalability in different ways. mpiP [43] uses a strategy called rank-based correlation to evaluate the scalability of MPI communication primitives. Their notion of scalability is different than ours: an MPI communication routine does not scale if its rank among other MPI calls performed by the application increases significantly when the number of processors increases. As such, mpiP provides qualitative rather than quantitative results. Quartz [3] aims to highlight inefficiency caused by load imbalance and serialization. In contrast, we pinpoint and quantify *any* aspect of a program whose performance scales worse than expected.

The work most closely related to our own is that of McKenney [20]. He describes a *differential profiling* strategy for analysis of two or more executions by mathematically combining corresponding buckets of different execution profiles. He recognizes that different combining functions are useful for different situations. Our work applies a differential profiling strategy for scalability analysis. Unlike McKenney, we use call path profiles rather than flat profiles and have automated analysis and presentation tools. Our profiling and analysis based on call path profiles enables extremely effective top-down analysis of large parallel programs based on layers of software and libraries.

3. METHODOLOGY

Users have specific expectations about how code performance should differ under different circumstances. This is true for both serial and parallel executions. Consider an ensemble of parallel executions. When different numbers of processors are used to solve the same problem (strong scaling), one expects an execution's speedup with respect to a serial execution to be linearly proportional to the number of processors used. When different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), one expects the execution time for all executions in an ensemble to be the same. In each of these situations, we can put our expectations to work for analyzing application performance. We can use our expectations about how overall application performance should scale at each point in the program to pinpoint and quantify deviations from expected scaling. While differential performance analysis using expectations applies more broadly, in this paper we focus on using expectations to pinpoint scalability bottlenecks in ensembles of executions used to study either strong or weak scaling of a parallel application.

Call path profiling [18] is the measurement technique that forms the foundation for the scalability analysis that we describe in this paper. We use library preloading to initiate profiling of unmodified, fully-optimized application binaries without prior arrangement. HPCToolkit's call path profiler uses timer-based sampling to attribute execution time and waiting to calling contexts [15, 16]. The profiler stores sample counts and their associated calling contexts in a *calling context tree* (CCT) [2]. In a CCT, the path from each node to the root of the tree represents a distinct calling context. A calling context is represented by a list of instruction pointers, one for each procedure frame active at the time the event occurred. Sample counts attached to each node in the tree associate execution costs with the calling context in which they were recorded. After post-mortem processing, CCTs contain three types of nodes: procedure frames, call sites and simple statements.

To use performance expectations to pinpoint and quantify scalability bottlenecks in a parallel application, we first collect call path profiles of a application for an ensemble R of two or more parallel execution runs. Let $R = \{R_1, ..., R_m\}$, where R_i represents an execution run on p_i processors, $i = 1, m$, where $m \geq 2$. Let T_i be the running time of the run R_i. The call path profile of each run in the ensemble is represented by a CCT.

In our analysis, we consider both *inclusive* and *exclusive* costs for CCT nodes. The inclusive cost at n represents the sum of all costs attributed to n and any of its descendants in the CCT, and is denoted by $I(n)$. The exclusive cost at n represents the sum of all costs attributed strictly to n, and we denote it by $E(n)$. If n is an interior node in a CCT, it represents an invocation of a procedure. If n is a leaf in a CCT, it represents a statement inside some procedure. For leaves, their inclusive and exclusive costs are equal.

It is useful to perform scalability analysis for both inclusive and exclusive costs; if the loss of scalability attributed to the inclusive costs of a function invocation is roughly equal to the loss of scalability due to its exclusive costs, then we know that the computation in that function invocation doesn't scale. However, if the loss of scalability attributed to a function invocation's inclusive costs outweighs the loss of scalability accounted for by exclusive costs, we need to explore the scalability of the function's callees.

Given CCTs for an ensemble of executions, the next step to analyzing the scalability of their performance is to clearly define our expectations. Next, we describe performance expectations for strong and weak scaling and intuitive metrics that represent how much performance deviates from our expectations.

Strong Scaling

Consider two strong scaling experiments for an application executed on p and q processors, respectively, $p < q$. If the application exhibits perfect (relative) strong scaling, then the execution time on q processors would be q/p times faster than on p processors. In fact, if every part of the application scales uniformly, then we would expect that the execution time spent in *each part* of the application reduces by a factor of q/p when we move from p to q processors.

More precisely, we expect that the execution time attributed to all pairs of *corresponding nodes*[1] in the CCTs for executions on p and q processors will scale in this fashion. We can use this high-level model as the basis for identifying where execution scalability falls short of our expectations. Consider a pair of corresponding nodes n_p and n_q in CCTs measured on p and q processors respectively. Let $C(n)$ denote the cost incurred for a node n in a CCT. With strong scaling, on q processors $C(n_q)$ should be a factor of q/p less than $C(n_p)$, the cost incurred at the corresponding CCT node on p processors. Thus, we expect that $C(n_q) = (p/q)C(n_p)$, or equivalently $qC(n_q) = pC(n_p)$. For a CCT node representing an invocation of a solver, this intuitively represents our expectation that the total amount of work spent in the solver (summed over all of the processors in each execution) will be constant. To capture the departure from our expectation, we compute the excess work in the q-processor execution as $qC(n_q) - pC_p(n_p)$. Let $M_{qp}(n)$ be the mapping between corresponding nodes n_q in the CCT on q processors and n_p, in the CCT on p processors. Using this notation, we can express n_p, the corresponding node to n_q as $M_{qp}(n_q)$. To normalize this previous value, as before, we divide it by qT_q, the total work performed in experiment R_q, to obtain

$$X_s(C, n_q) = \frac{qC(n_q) - pC(M_{qp}(n_q))}{qT_q}$$

the fraction of the execution time that represents excess work attributed to any node n_q in the CCT for an execution on q processors relative to its execution on p processors.

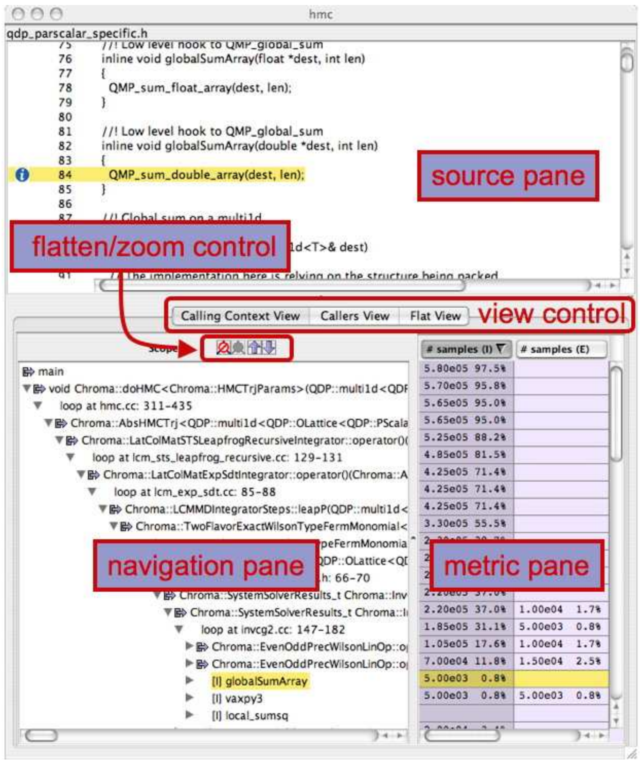
This metric tells us what fraction of the total execution time on q processors was spent executing excess work on behalf of a node in the CCT. Excess work serves as a measure of scalability. Relative parallel efficiency at node n_q can be computed as $1 - X_s(C, n_q)$.

For a node n_q, we compute the excess work for inclusive costs as $X_s(I, n_q)$, and the excess work for exclusive costs as $X_s(E, n_q)$.

Weak Scaling

Consider two weak scaling experiments executed on p and q processors, respectively, $p < q$, and two corresponding CCT nodes n_q and $M_{qp}(n_q)$. The expectation is that

[1]Corresponding nodes in a pair of CCTs represent the same calling context, *e.g.*, `main` calls `solve`, which calls `sparse matrix-vector multiply`.



```
//! Low level hook to QMP_global_sum
inline void globalSumArray(float *dest, int len)
{
  QMP_sum_float_array(dest, len);
}

//! Low level hook to QMP_global_sum
inline void globalSumArray(double *dest, int len)
{
  QMP_sum_double_array(dest, len);
}

//! Global sum on a multi1d
                              1d<T>& dest)

  // The implementation here is relying on the structure being packed
```

```
main                                                             5.80e05 97.5%
▼ void Chroma::doHMC<Chroma::HMCTrjParams>(QDP::multi1d<QDF      5.70e05 95.8%
  ▼ loop at hmc.cc: 311-435                                      5.65e05 95.0%
    ▼ Chroma::AbsHMCTrj<QDP::multi1d<QDP::OLattice<QDP::PScala   5.65e05 95.0%
      ▼ Chroma::LatColMatSTSLeapfrogRecursiveIntegrator::operator()(  5.25e05 88.2%
        ▼ loop at lcm_sts_leapfrog_recursive.cc: 129-131         4.85e05 81.5%
          ▼ Chroma::LatColMatExpSdtIntegrator::operator()(Chroma::A   4.25e05 71.4%
            ▼ loop at lcm_exp_sdt.cc: 85-88                      4.25e05 71.4%
              ▼ Chroma::LCMMDIntegratorSteps::leapP(QDP::multi1d<     4.25e05 71.4%
                ▼ Chroma::TwoFlavorExactWilsonTypeFermMonomial<       3.30e05 55.5%
                                        peFermMonomia
                                        QDP::OLattice<QD
                                        h: 66-70
                      ▼ Chroma::SystemSolverResults_t Chroma::Inv
                        ▼ Chroma::SystemSolverResults_t Chroma::I    2.20e05 37.0%  1.00e04  1.7%
                          ▼ loop at invcg2.cc: 147-182               1.85e05 31.1%  5.00e03  0.8%
                            ▶ Chroma::EvenOddPrecWilsonLinOp::o      1.05e05 17.6%  1.00e04  1.7%
                            ▶ Chroma::EvenOddPrecWilsonLinOp::o      7.00e04 11.8%  1.50e04  2.5%
                            ▶ [I] globalSumArray                     5.00e03  0.8%
                            ▶ [I] vaxpy3                             5.00e03  0.8%  5.00e03  0.8%
                            ▶ [I] local_sumsq
```

Figure 1: Overview of the hpcviewer user interface.

$C_q(n_q) = C_p(M_{qp}(n_q))$, and the deviation from the expectation is $C_q(n_q) - C_p(M_{qp}(n_q))$. We normalize this value by dividing it by the total execution time of experiment R_q, and define the fraction of the execution time representing excess work attributed to node n_q as follows

$$X_w(C, n_q) = \frac{C_q(n_q) - C_p(M_{qp}(n_q))}{T_q}$$

For a node n_q, we compute the excess work for inclusive costs as $X_w(I, n_q)$, and the excess work for exclusive costs as $X_w(E, n_q)$.

3.1 Automating Scalability Analysis

The HPCToolkit performance tools provide support for performance measurement, attribution, and analysis [15, 16, 21, 35, 39]. To perform scalability analysis of a parallel application, we first collect call path profiles on individual processes for two or more program executions at different scales of parallelism. Next, we analyze the call path profiles, correlate them with program source code, and produce a calling context tree annotated with performance metrics. These profiles can then be examined with HPCToolkit's `hpcviewer` browser, which supports interactive examination of performance databases. Support for computing and analyzing scalability metrics is integrated into `hpcviewer`. We compute scalability metrics by performing differential analysis between a pair of profiles collected on different numbers of processors and annotate the calling context tree at all levels with these metrics. Figure 1 shows a screenshot of the `hpcviewer` interface with panes and key controls labeled. `hpcviewer` supports three principal views of an application's performance data:

- *Calling context view.* This top-down view represents the dynamic calling contexts (call paths) in which costs were incurred.

- *Callers view.* This bottom up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are used in more than one context, such as communication library routines.

- *Flat view.* This view organizes performance measurement data according to the static structure of an application. All costs incurred in *any* calling context by a procedure are aggregated together in the flat view.

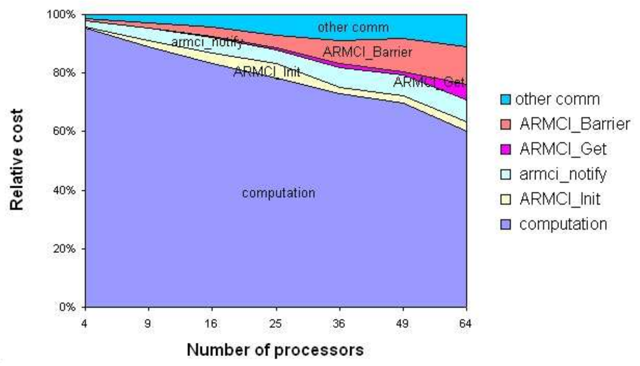
The browser window is divided into three panes: the navigation pane, the source pane, and the metrics pane. The navigation pane presents a hierarchical tree-based structure that is used to organize the presentation of an application's performance data. The source pane displays the source code associated with the current entity selected in the navigation pane. The metric pane displays one or more performance metrics associated with entities in the navigation pane.

3.2 Analysis Using Excess Work

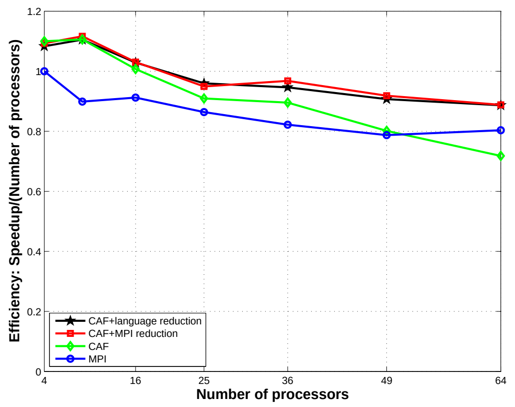
The `hpcviewer` interface enables developers to explore both top-down and bottom-up views of the annotated CCTs, helping them to quickly pinpoint trouble spots. Typically, a user would begin analyzing an application's scalability and performance using the top-down calling context tree view. Using this view, one can readily see how costs and scalability losses are associated with different calling contexts. If costs or scalability losses are associated with only a few calling contexts, then this view suffices for analyzing the bottlenecks. When scalability losses are spread among many calling contexts, often it is useful to switch to a bottom-up view of the data to see if many of these costs are due to the same underlying cause. In the bottom-up view, one locates the routine with the highest aggregated scalability losses and then looks upward to see how these costs apportion across the different calling contexts in which the routine was invoked.

The excess work metrics are intuitive; perfect scaling corresponds to a value of 0, sublinear scaling yields positive values, and superlinear scaling yields negative values. Typically, CCTs for SPMD programs have similar structure; we report excess work where CCTs for different experiments diverge.

For strong scaling experiments, $X_s(I, n)$ and $X_s(E, n)$ serve as complementary measures of scalability of CCT node n. By using both metrics, one can determine whether the application scales well or not at node n, and also pinpoint the cause of any lack of scaling. If a node n corresponding to a function invocation has comparable positive values for $X_s(I, n)$ and $X_s(E, n)$ then the loss of scaling is due to computation in n. However, if the excess work reflected in n's inclusive costs outweighs that accounted for by its exclusive costs, then one should explore the scalability of n's callees. To isolate code that is an impediment to scalable performance, one simply inspects CCT nodes along a path starting at the root to pinpoint the cause of positive $X_s(I, n)$ values. For weak scaling, $X_w(I, n)$ and $X_w(E, n)$ play a similar role.



(a) Scalability of relative computation and communication costs for LBMHD.



(b) Parallel efficiency for the timed phase of MPI and CAF variants of LBMHD.

Figure 2: LBMHD relative costs scalability and parallel efficiency.

4. CASE STUDIES

To demonstrate the broad utility of our strategy for analyzing the scalability of parallel codes, we apply it to codes written in CAF, UPC and MPI. Our examples show experiments on modest numbers of processors (16–64 CPUs) with small problem sizes. Such configurations expose communication and synchronization inefficiencies that are impediments to scalability without consuming excessive resources for scalability studies.

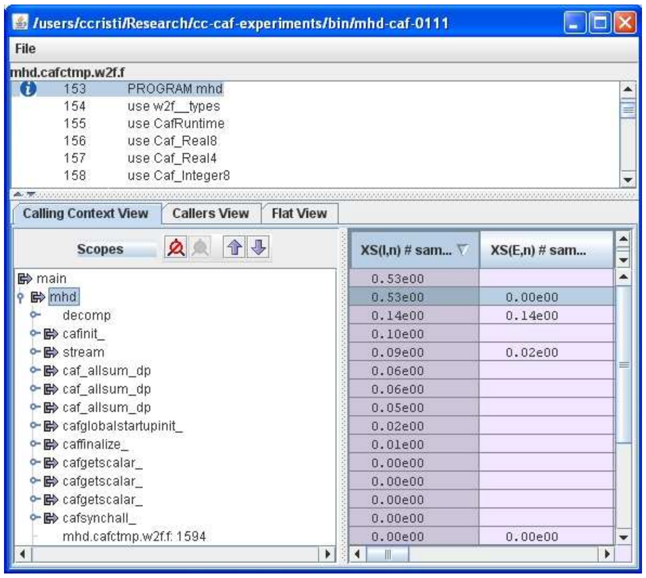
The experiments presented in this section were performed on a cluster of dual processors nodes with Itanium 2 processors and a Myrinet 2000 interconnect. Each node is running the Linux operating system. All codes were compiled with the Intel 9.0 compilers. We used the native Myrinet implementation of MPI.

4.1 Co-array Fortran

Here, we describe experiments with three benchmarks coded in CAF. All CAF codes were compiled with the open-source CAF compiler `cafc` [14] and used the ARMCI library [30] for communication.

4.1.1 LBMHD Benchmark

The LBMHD benchmark [33] simulates a charged fluid



mhd.cafctmp.w2f.f

```
PROGRAM mhd
use w2f__types
use CafRuntime
use Caf_Real8
use Caf_Real4
use Caf_Integer8
```

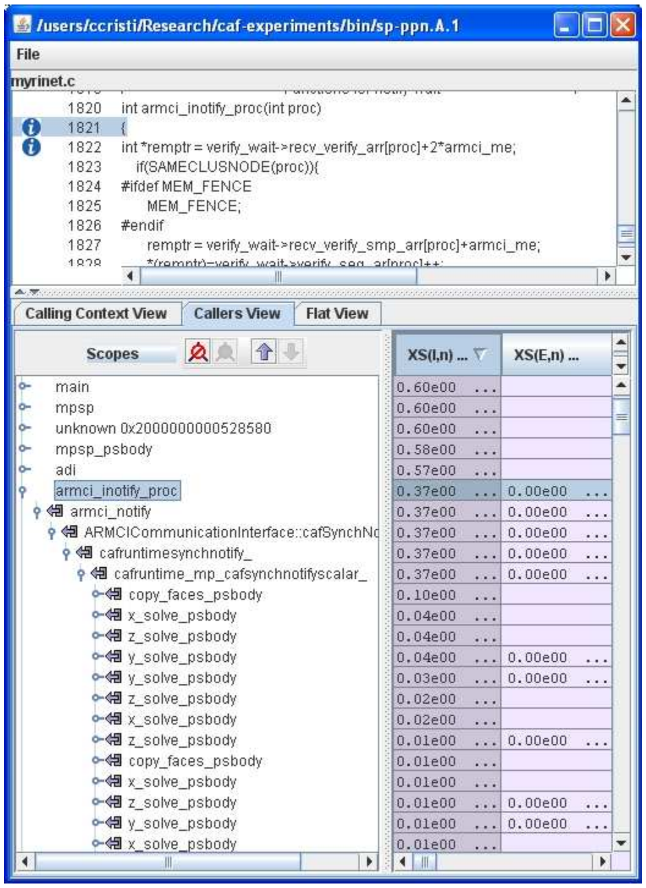
Scopes	XS(I,n) # sam...	XS(E,n) # sam...
main	0.53e00	
mhd	0.53e00	0.00e00
decomp	0.14e00	0.14e00
cafinit_	0.10e00	
stream	0.09e00	0.02e00
caf_allsum_dp	0.06e00	
caf_allsum_dp	0.06e00	
caf_allsum_dp	0.05e00	
cafglobalstartupinit_	0.02e00	
caffinalize_	0.01e00	
cafgetscalar_	0.00e00	
cafgetscalar_	0.00e00	
cafgetscalar_	0.00e00	
cafsynchall_	0.00e00	
mhd.cafctmp.w2f.f: 1594	0.00e00	0.00e00

Figure 3: Strong scaling analysis results for CAF LBMHD (size 1024^2) on 4 and 64 CPUs.

moving in a magnetic field using a Lattice Boltzmann formulation of the magnetohydrodynamic equations. The code uses block data partitioning onto a 2D processor grid and requires both contiguous and strided communication between processors. After each computation phase, each processor exchanges data with four communication partners in an ordered fashion to acquire data from eight neighbors.

Figure 2(a) shows how the relative costs of computation and select communication primitives vary for LBMHD as the number of CPUs increases from 4 to 64. The chart shows that the overall loss of efficiency on 64 CPUs due to communication overhead is 39%. The relative cost of barriers increases with the number of CPUs. In the original LBMHD source code that we received from LBNL, barriers were used to implement scalar reductions at the source level. The code performed three consecutive reductions on scalars. By replacing these scalar reductions with a three-element MPI vector reduction, performance improved by 25% on 64 processors. Figure 2(b) shows the parallel efficiency for timed phases of several CAF and MPI versions of the LBMHD benchmark.

Figure 3 shows we present screenshots with results of strong scaling analysis for CAF LBMHD, using relative excess work, on 4 and 64 CPUs. The figure shows that the excess work for the main routine `mhd` is 53%. The routine `decomp`, which performs the initial problem decomposition, has both inclusive and exclusive excess work of 14%, which means that its excess work is due to local computation. The routine `caf_allsum_dp` leads to an overall excess work of 17%. This routine uses a barrier-based implementation of a reduction, which is not efficient on clusters. This accounts for the growing overhead due to `ARMCI_Barrier` shown in Figure 2(a). The causes responsible for excess work in `cafinit` and `stream` can be discovered similarly. It is worth noting that the overall excess work of 53% shown in Figure 3 is significantly higher than the excess work of 39% indicated by the chart in Figure 2(a); this is because computation is contributing to excess work (*e.g.*, the routine



```
int armci_inotify_proc(int proc)
{
int *remptr = verify_wait->recv_verify_arr[proc]+2*armci_me;
   if(SAMECLUSNODE(proc)){
#ifdef MEM_FENCE
     MEM_FENCE;
#endif
     remptr = verify_wait->recv_verify_smp_arr[proc]+armci_me;
```

Scopes	XS(I,n) ...	XS(E,n) ...
main	0.60e00 ...	
mpsp	0.60e00 ...	
unknown 0x2000000000528580	0.60e00 ...	
mpsp_psbody	0.58e00 ...	
adi	0.57e00 ...	
armci_inotify_proc	0.37e00 ...	0.00e00 ...
armci_notify	0.37e00 ...	0.00e00 ...
ARMCICommunicationInterface::cafSynchNo	0.37e00 ...	0.00e00 ...
cafruntimesynchnotify_	0.37e00 ...	0.00e00 ...
cafruntime_mp_cafsynchnotifyscalar_	0.37e00 ...	0.00e00 ...
copy_faces_psbody	0.10e00 ...	
x_solve_psbody	0.04e00 ...	
z_solve_psbody	0.04e00 ...	
y_solve_psbody	0.04e00 ...	0.00e00 ...
y_solve_psbody	0.03e00 ...	0.00e00 ...
z_solve_psbody	0.02e00 ...	
x_solve_psbody	0.02e00 ...	
z_solve_psbody	0.01e00 ...	0.00e00 ...
copy_faces_psbody	0.01e00 ...	
x_solve_psbody	0.01e00 ...	
z_solve_psbody	0.01e00 ...	0.00e00 ...
y_solve_psbody	0.01e00 ...	0.00e00 ...
x_solve_psbody	0.01e00 ...	

Figure 4: Strong scaling analysis of the CAF version of NAS SP class A (size 64^3) on 4 and 64 CPUs.

`decomp`) in addition to communication overhead.

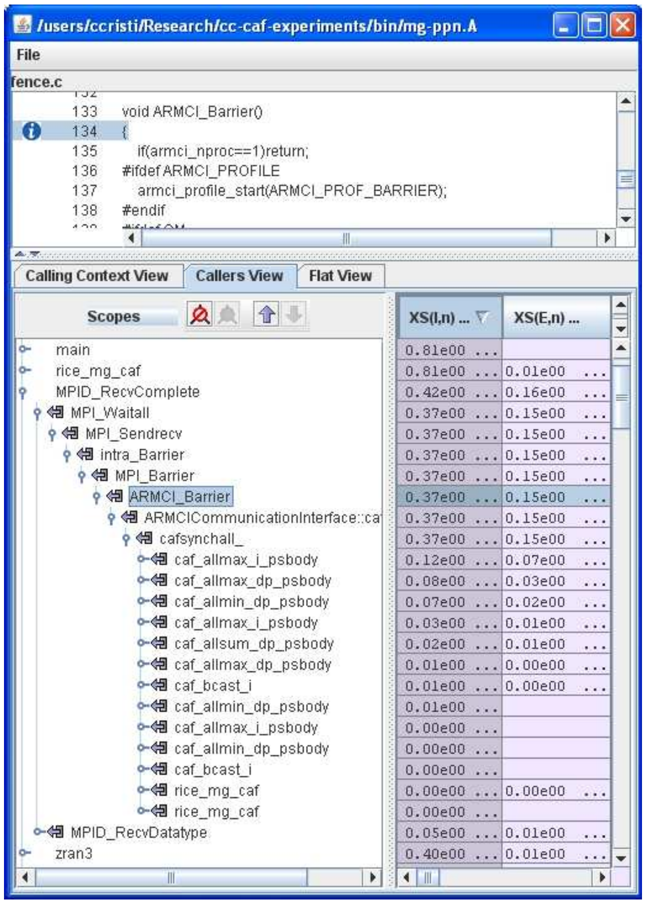
In [10] we explored CAF extensions with collective operations, such as broadcast and reductions, and presented an initial implementation strategy, using the MPI collectives. In Figure 2(b) we present the parallel efficiency plot for the timed phase of LBMHD using the the CAF level reductions; one observation if that our translation scheme does not introduce a high overhead over direct calls of MPI primitives.

An important observation is that by using the scaling analysis with scalability information attributed to the call-tree nodes, we obtained results similar to the one obtained using the communication primitives relative costs plots. However, the scaling analysis is vastly more accurate and more useful, leading a user to non-scaling call tree nodes, rather than just to non-scaling commumication primitives.

4.1.2 NAS SP Benchmark

The second CAF benchmark we study is NAS SP, described in [14]. This benchmark solves scalar penta-diagonal systems of equations resulting from an approximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [4]. SP employs a parallelization based on a skewed-cyclic block distribution known as multipartitioning.

Figure 4 shows a screenshot of the strong scaling analysis results for the CAF version of NAS SP on 4 and 64 CPUs, using the bottom-up view. The figure shows that



```
void ARMCI_Barrier()
{
   if(armci_nproc==1)return;
#ifdef ARMCI_PROFILE
   armci_profile_start(ARMCI_PROF_BARRIER);
#endif
```

Scopes	XS(I,n) ...	XS(E,n) ...
main	0.81e00 ...	
rice_mg_caf	0.81e00 ...	0.01e00 ...
MPID_RecvComplete	0.42e00 ...	0.16e00 ...
MPI_Waitall	0.37e00 ...	0.15e00 ...
MPI_Sendrecv	0.37e00 ...	0.15e00 ...
intra_Barrier	0.37e00 ...	0.15e00 ...
MPI_Barrier	0.37e00 ...	0.15e00 ...
ARMCI_Barrier	0.37e00 ...	0.15e00 ...
ARMCICommunicationInterface::ca	0.37e00 ...	0.15e00 ...
cafsynchall_	0.37e00 ...	0.15e00 ...
caf_allmax_i_psbody	0.12e00 ...	0.07e00 ...
caf_allmax_dp_psbody	0.08e00 ...	0.03e00 ...
caf_allmin_dp_psbody	0.07e00 ...	0.02e00 ...
caf_allmax_i_psbody	0.03e00 ...	0.01e00 ...
caf_allsum_dp_psbody	0.02e00 ...	0.01e00 ...
caf_allmax_dp_psbody	0.01e00 ...	0.00e00 ...
caf_bcast_i	0.01e00 ...	0.00e00 ...
caf_allmin_dp_psbody	0.01e00 ...	
caf_allmax_i_psbody	0.00e00 ...	
caf_allmin_dp_psbody	0.00e00 ...	
caf_bcast_i	0.00e00 ...	
rice_mg_caf	0.00e00 ...	0.00e00 ...
rice_mg_caf	0.00e00 ...	
MPID_RecvDatatype	0.05e00 ...	0.01e00 ...
zran3	0.40e00 ...	0.01e00 ...

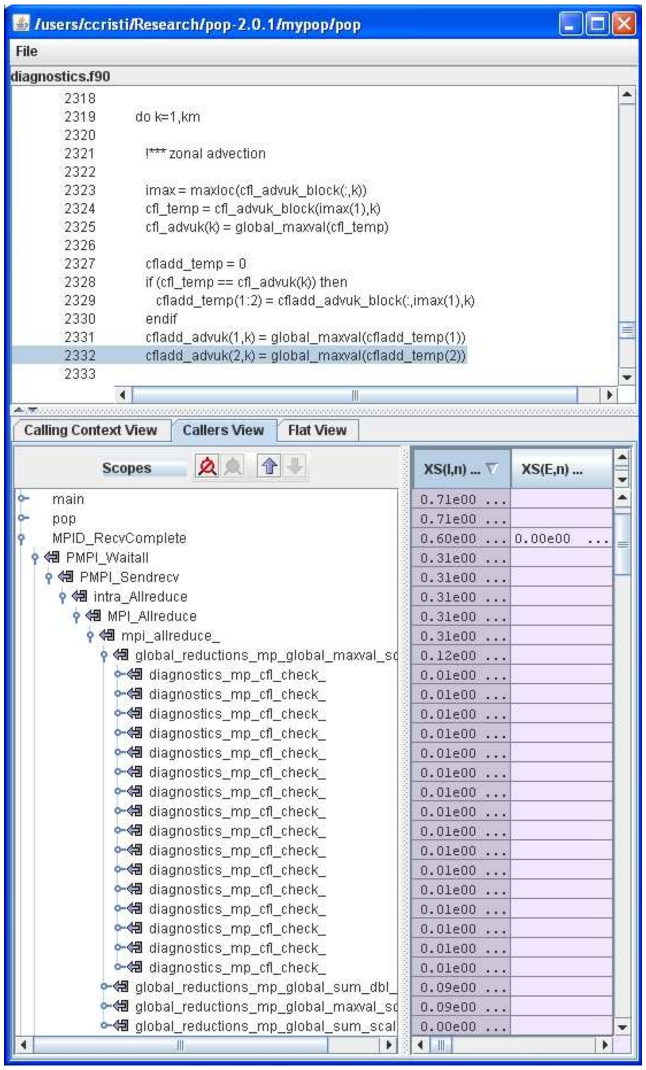
Figure 5: Strong scaling analysis of the CAF version of NAS MG class A (size 256^3) on 4 and 64 CPUs.

the overall excess work for the main routine `mpsp` is 60%, out of which the routine `adi`, which performs alternating direction integration, accounts for 58% excess work. By exploration of the bottom-up view we determined that the ARMCI routine `arcmi_inotify_proc`, which we used to implement point-to-point synchronization in the CAF runtime system, contributes a total of 37% to the excess work. Despite the fact that this synchronization is used in numerous calling contexts, the bottom-up view makes diagnosing its scalability problems trivial.

4.1.3 NAS MG Benchmark

The third CAF code that we studied is the NAS MG multigrid benchmark. This code calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [4]. The CAF implementation of NAS MG is described in [14].

Figure 5 shows a screenshot of the strong scaling analysis results for CAF NAS MG, class A (size 256^3), using relative excess work on 1 and 64 processors, and the bottom-up view. The benchmark exhibits 81% excess work. The routine `MPID_RecvComplete` accounts for 37% excess work, and the bottom-up view shows that it is used for a variety of barrier-based implementations of reductions such as sum and maximum. These implementations of reductions,



```
do k=1,km

  !*** zonal advection

  imax = maxloc(cfl_advuk_block(:,k))
  cfl_temp = cfl_advuk_block(imax(1),k)
  cfl_advuk(k) = global_maxval(cfl_temp)

  cfladd_temp = 0
  if (cfl_temp == cfl_advuk(k)) then
    cfladd_temp(1:2) = cfladd_advuk_block(:,imax(1),k)
  endif
  cfladd_advuk(1,k) = global_maxval(cfladd_temp(1))
  cfladd_advuk(2,k) = global_maxval(cfladd_temp(2))
```

Scopes	XS(I,n) ...	XS(E,n) ...
main	0.71e00 ...	
pop	0.71e00 ...	
MPID_RecvComplete	0.60e00 ...	0.00e00 ...
PMPI_Waitall	0.31e00 ...	
PMPI_Sendrecv	0.31e00 ...	
intra_Allreduce	0.31e00 ...	
MPI_Allreduce	0.31e00 ...	
mpi_allreduce_	0.31e00 ...	
global_reductions_mp_global_maxval_sc	0.12e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
diagnostics_mp_cfl_check_	0.01e00 ...	
global_reductions_mp_global_sum_dbl_	0.09e00 ...	
global_reductions_mp_global_maxval_sc	0.09e00 ...	
global_reductions_mp_global_sum_scal	0.00e00 ...	

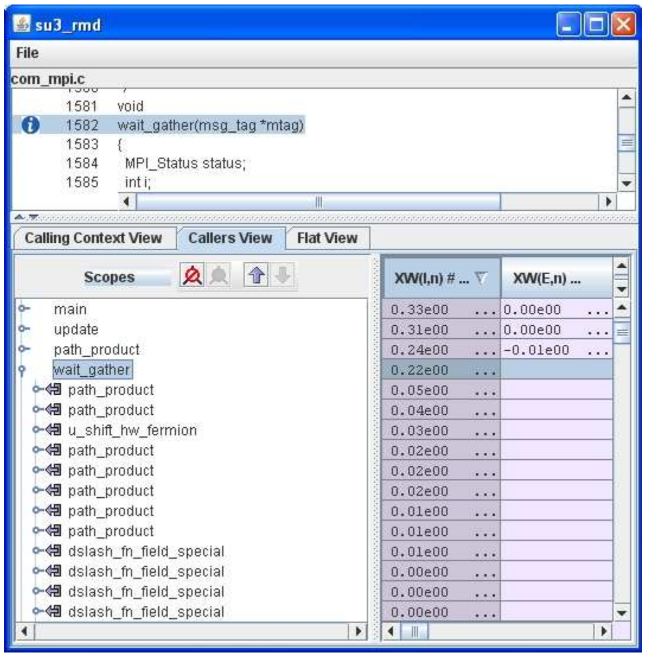
Figure 6: Strong scaling analysis results for POP on 4 and 64 CPUs using the bottom-up view.

which we received from Robert Numrich, were implemented at the source level in CAF. On Cray systems, barriers typically have fast hardware support, so these implementations are fast; however,they are not well-suited for clusters and lead to poor scalability. Using `hpcviewer`, one can discover this bottleneck in seconds. In the MG benchmark, reductions are used in the initialization phase; however, they still contribute to the program's overall execution time.

4.2 MPI Applications

4.2.1 LANL's Parallel Ocean Program (POP)

To explore the utility of our scalability analysis on MPI applications, we analyzed version 2.0.1 of LANL's Parallel Ocean Program (POP) [40]. POP is an ocean circulation model in which depth is used as the vertical coordinate. The model solves the three-dimensional primitive equations for fluid motions on the sphere under hydrostatic and Boussinesq approximations. Spatial derivatives are computed using finite-difference discretizations which are formulated to



```
void
wait_gather(msg_tag *mtag)
{
 MPI_Status status;
 int i;
```

Scopes	XW(I,n) # ...	XW(E,n) ...
main	0.33e00 ...	0.00e00 ...
update	0.31e00 ...	0.00e00 ...
path_product	0.24e00 ...	-0.01e00 ...
wait_gather	0.22e00 ...	
path_product	0.05e00 ...	
path_product	0.04e00 ...	
u_shift_hw_fermion	0.03e00 ...	
path_product	0.02e00 ...	
path_product	0.02e00 ...	
path_product	0.02e00 ...	
path_product	0.01e00 ...	
path_product	0.01e00 ...	
dslash_fn_field_special	0.01e00 ...	
dslash_fn_field_special	0.00e00 ...	
dslash_fn_field_special	0.00e00 ...	
dslash_fn_field_special	0.00e00 ...	

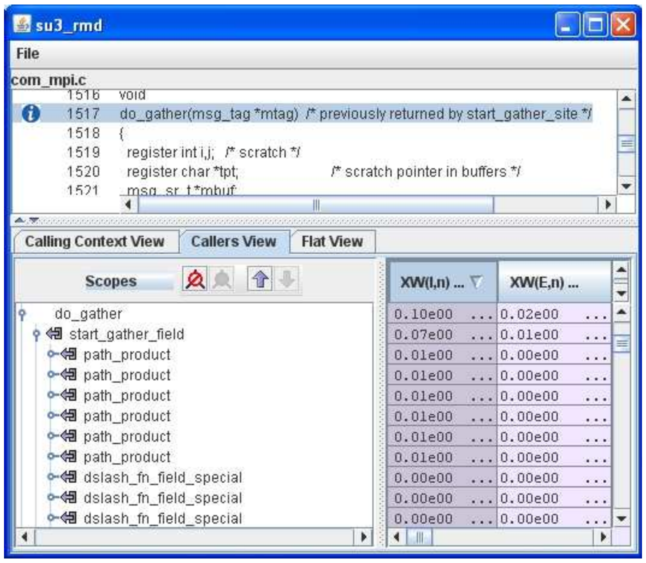
Figure 7: Weak scaling analysis for `su3_rmd` on 1 and 16 processors using the bottom-up view.

handle any generalized orthogonal grid on a sphere, including dipole and tripole grids which shift the North Pole singularity into land masses to avoid time step constraints due to grid convergence.

We analyzed POP for a "large" test domain, with 384x288 domain size, 32 vertical levels, and 2 tracers. The relative excess work results for 4 and 64 CPUs, using the top-down view, show that the main program has 71% excess work. By using the bottom-up view, as shown in Figure 6, we discovered that the communication routine `MPID_RecvComplete` accounts for 60% of the excess work. Further examination of the calling contexts of this subroutine revealed that 21% of the excess work is due to calls from scalar reductions on integer and double precision variables. By inspecting the source code associated with these contexts, we discovered that the scalar reductions were invoked in succession by a single routine. This deficiency can be addressed by aggregating the reductions. Using `hpcviewer`'s bottom-up view pinpointed this scaling bottleneck quickly.

4.2.2 Lattice QCD: MILC

MILC [5] represents a set of parallel codes developed for the study of lattice quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics. These codes were designed to run on MIMD parallel machines. They are written in C, and they are based on MPI. MILC is part of a set of codes used by NSF as procurement benchmarks for petascale systems [31]. Version 7 of MILC uses the SciDAC libraries [1] to optimize the communication. We present an analysis of the version 7.2.1 of MILC using MPI as communication substrate. Our goal for analysis of MILC is to demonstrate the applicability of our method for analysis of SPMD codes subjected to weak scaling. From the MILC applications, we analyzed `su3_rmd`—an



```
void
do_gather(msg_tag *mtag)  /* previously returned by start_gather_site */
{
  register int i,j;   /* scratch */
  register char *tpt;                    /* scratch pointer in buffers */
  msg_sr_t *mbuf;
```

Scopes	XW(I,n) ...	XW(E,n) ...
do_gather	0.10e00 ...	0.02e00 ...
start_gather_field	0.07e00 ...	0.01e00 ...
path_product	0.01e00 ...	0.00e00 ...
path_product	0.01e00 ...	0.00e00 ...
path_product	0.01e00 ...	0.00e00 ...
path_product	0.01e00 ...	0.00e00 ...
path_product	0.01e00 ...	0.00e00 ...
path_product	0.01e00 ...	0.00e00 ...
dslash_fn_field_special	0.00e00 ...	0.00e00 ...
dslash_fn_field_special	0.00e00 ...	0.00e00 ...
dslash_fn_field_special	0.00e00 ...	0.00e00 ...

Figure 8: Weak scaling analysis for `su3_rmd` on 1 and 16 processors using the bottom-up view for the routine `do_gather`.

Kogut-Susskind molecular dynamics code using the R algorithm. We chose our input sizes so that as we increased the number of processors, the work on each processor remained constant. With weak scaling, one's expectation is that the overall running time should be the same on any number of processors. In Figures 7 and 8 show screenshots of the weak scaling analysis results for `su3_rmd` using relative excess work on 1 and 16 CPUs, displayed in the bottom-up view. Overall, `su3_rmd` loses 33% efficiency.

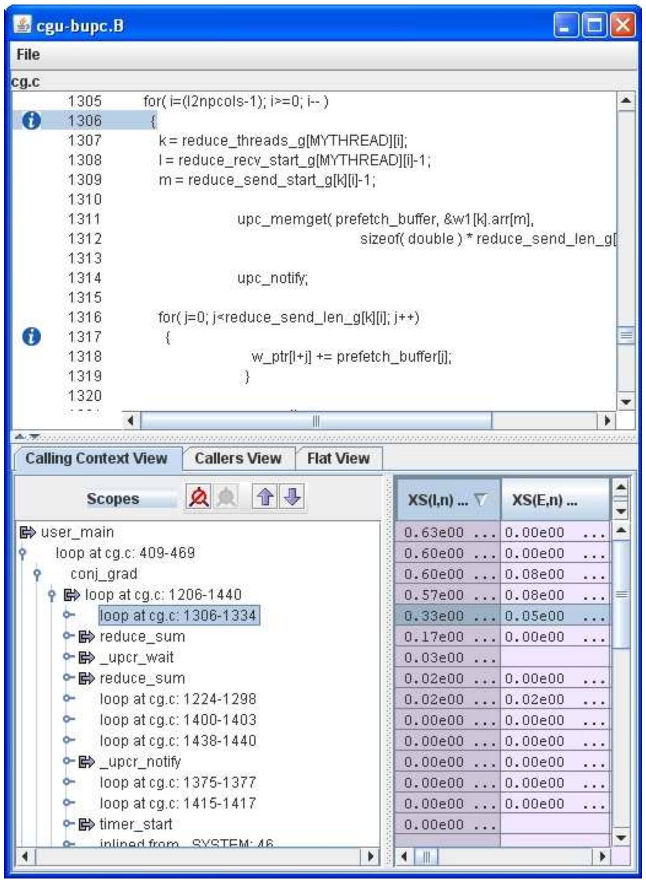
By inspecting the bottom-up view, we were able to attribute the loss of scaling to the routines `wait_gather` and `do_gather`. As shown in Figure 7, `wait_gather` is called from a multitude of callsites and routines, accounting for 22% of the excess work. The routine `do_gather` is responsible for 10% excess work, and similarly is called from multiple callsites, as shown in Figure 8. It posts a series of non-blocking receives, then performs blocking sends, while `wait_gather` waits for completion of the non-blocking receives.

Overall, we demonstrated that our scaling analysis technique can be applied as well to the analysis of weak scaling parallel codes, and it pointed to a pair of communication routines as the signficiant source of inefficiency.

4.3 Unified Parallel C

To evaluate the applicability of the expectations-based scaling analysis to UPC codes, we analyzed the UPC version of NAS CG. The CG benchmark uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [4]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The UPC version of NAS CG is described elsewhere [11]. The UPC code was compiled using Berkeley's UPC compiler [9].

In Figures 9, we present screenshots of the scaling analysis results for UPC NAS CG class B (size 75000), using



```
for( i=(l2npcols-1); i>=0; i-- )
 {
  k = reduce_threads_g[MYTHREAD][i];
  l = reduce_recv_start_g[MYTHREAD][i]-1;
  m = reduce_send_start_g[k][i]-1;

                upc_memget( prefetch_buffer, &w1[k].arr[m],
                                    sizeof( double ) * reduce_send_len_g[

                upc_notify;

  for( j=0; j<reduce_send_len_g[k][i]; j++)
   {
                  w_ptr[l+j] += prefetch_buffer[j];
                }

```

Scopes	XS(I,n) ...	XS(E,n) ...
user_main	0.63e00 ...	0.00e00 ...
loop at cg.c: 409-469	0.60e00 ...	0.00e00 ...
conj_grad	0.60e00 ...	0.08e00 ...
loop at cg.c: 1206-1440	0.57e00 ...	0.08e00 ...
loop at cg.c: 1306-1334	0.33e00 ...	0.05e00 ...
reduce_sum	0.17e00 ...	0.00e00 ...
_upcr_wait	0.03e00 ...	
reduce_sum	0.02e00 ...	0.00e00 ...
loop at cg.c: 1224-1298	0.02e00 ...	0.02e00 ...
loop at cg.c: 1400-1403	0.00e00 ...	0.00e00 ...
loop at cg.c: 1438-1440	0.00e00 ...	0.00e00 ...
_upcr_notify	0.00e00 ...	0.00e00 ...
loop at cg.c: 1375-1377	0.00e00 ...	0.00e00 ...
loop at cg.c: 1415-1417	0.00e00 ...	0.00e00 ...
timer_start	0.00e00 ...	
inlined from _SYSTEM: 46		

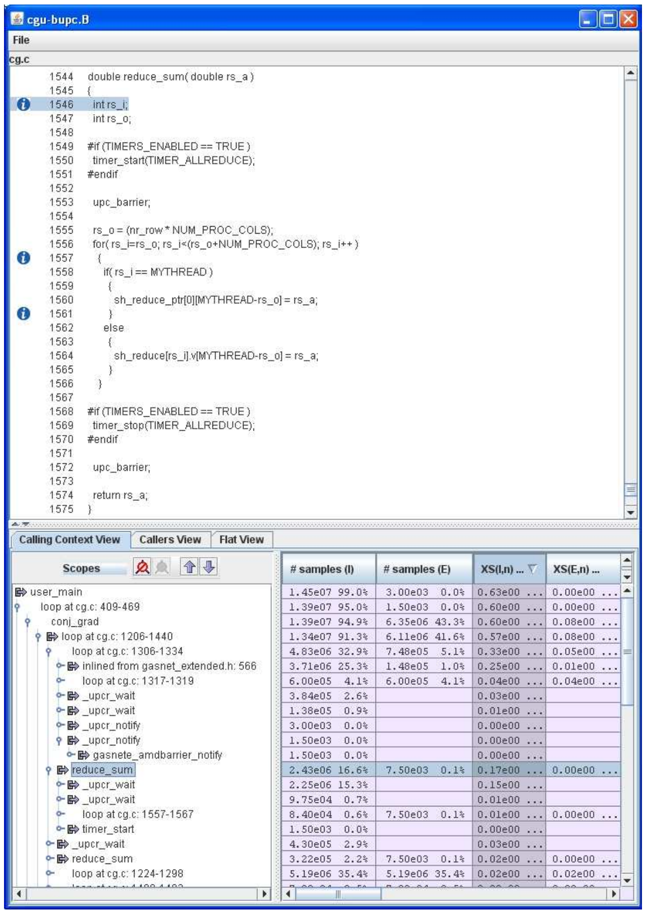
Figure 9: Strong scaling analysis for UPC NAS CG class B (size 75000), on 1 and 16 CPUs for the prefetching loop.

relative excess work on 1 and 16 CPUs. The main program loses 63% efficiency, out of which the `conj_grad` routine, which performs conjugate gradient computation, accounts for 60% excess work. By further analyzing the top-down view of the calling context tree, we determined that a remote data prefetching loop accounts for 33% excess work, split between calls to `upc_memget` with 25% inclusive excess work, and local computation using the prefetch buffer, with 4% exclusive excess work. Next, calls to `reduce_sum` account for 19% inclusive excess work. The source code correlation indicated that `reduce_sum` has a suboptimal implementation, using barriers, as shown in Figure 10; a solution would be to employ one of the UPC collective operations.

5. CONCLUSIONS

Differential analysis of call path profiles using performance expectations is a powerful technique for pinpointing scalability bottlenecks in parallel programs. It is applicable to a broad range of applications because it is not limited to any particular programming model. By analyzing performance using a metric based on the fraction of excess work, our scalability analysis focuses attention on what matters; absolute scalability is less relevant than the overall cost incurred in an execution due to lack of scalability.

Our case studies show that scalability analysis using expectations can highlight bottlenecks including inefficient im-



```
1544  double reduce_sum( double rs_a )
1545  {
1546    int rs_i;
1547    int rs_o;
1548
1549  #if (TIMERS_ENABLED == TRUE )
1550    timer_start(TIMER_ALLREDUCE);
1551  #endif
1552
1553    upc_barrier;
1554
1555    rs_o = (nr_row * NUM_PROC_COLS);
1556    for( rs_i=rs_o; rs_i<(rs_o+NUM_PROC_COLS); rs_i++ )
1557      {
1558        if( rs_i == MYTHREAD )
1559          {
1560            sh_reduce_ptr[0][MYTHREAD-rs_o] = rs_a;
1561          }
1562        else
1563          {
1564            sh_reduce[rs_i].v[MYTHREAD-rs_o] = rs_a;
1565          }
1566      }
1567
1568  #if (TIMERS_ENABLED == TRUE )
1569    timer_stop(TIMER_ALLREDUCE);
1570  #endif
1571
1572    upc_barrier;
1573
1574    return rs_a;
1575  }
```

Scopes	# samples (I)	# samples (E)	XS(I,n) ...	XS(E,n) ...
user_main	1.45e07 99.0%	3.00e03 0.0%	0.63e00 ...	0.00e00 ...
loop at cg.c: 409-469	1.39e07 95.0%	1.50e03 0.0%	0.60e00 ...	0.00e00 ...
conj_grad	1.39e07 94.9%	6.35e06 43.3%	0.60e00 ...	0.08e00 ...
loop at cg.c: 1206-1440	1.34e07 91.3%	6.11e06 41.6%	0.57e00 ...	0.08e00 ...
loop at cg.c: 1306-1334	4.83e06 32.9%	7.48e05 5.1%	0.33e00 ...	0.05e00 ...
inlined from gasnet_extended.h: 566	3.71e06 25.3%	1.48e05 1.0%	0.25e00 ...	0.01e00 ...
loop at cg.c: 1317-1319	6.00e05 4.1%	6.00e05 4.1%	0.04e00 ...	0.04e00 ...
_upcr_wait	3.84e05 2.6%		0.03e00 ...	
_upcr_wait	1.38e05 0.9%		0.01e00 ...	
_upcr_notify	3.00e03 0.0%		0.00e00 ...	
_upcr_notify	1.50e03 0.0%		0.00e00 ...	
gasnete_amdbarrier_notify	1.50e03 0.0%		0.00e00 ...	
reduce_sum	2.43e06 16.6%	7.50e03 0.1%	0.17e00 ...	0.00e00 ...
_upcr_wait	2.25e06 15.3%		0.15e00 ...	
_upcr_wait	9.75e04 0.7%		0.01e00 ...	
loop at cg.c: 1557-1567	8.40e04 0.6%	7.50e03 0.1%	0.01e00 ...	0.00e00 ...
timer_start	1.50e03 0.0%		0.00e00 ...	
_upcr_wait	4.30e05 2.9%		0.03e00 ...	
reduce_sum	3.22e05 2.2%	7.50e03 0.1%	0.02e00 ...	0.00e00 ...
loop at cg.c: 1224-1298	5.19e06 35.4%	5.19e06 35.4%	0.02e00 ...	0.02e00 ...

Figure 10: Strong scaling analysis for UPC NAS CG class B (size 75000) on 1 and 16 CPUs.

plementations of synchronization and reductions, inefficient use of reduction primitives, and computation costs that don't scale appropriately. These examples demonstrate the utility of our approach for pinpointing scalability bottlenecks no matter what their underlying cause. When used in conjunction with performance analysis based on expectations, our tools attribute scalability bottlenecks to full calling contexts, which enables them to be precisely diagnosed.

We are in the process of refining HPCToolkit's performance measurement and analysis tools, which includes support for our scalability analysis, with the aim of deploying them on the emerging petascale systems at the national laboratories in the United States. We plan to use these tools as the basis for helping applications harness the power of these enormous machines.

Acknowledgments

We thank Nathan Tallent, and Fengmei Zhao for their contributions to collection and analysis of call path profiles in HPCToolkit. We thank Adam Bordelon for his work on integrating scalability analysis into HPCToolkit's `hpcviewer` user interface. This work was supported in part by the DOE Office of Science through Cooperative Agreement numbers DE-FC03-01ER25504, DE-FC02-06ER25762, DE-FC02-07ER25800, and DE-FC02-06ER25754. Experiments were performed on equipment purchased with support from Intel and the NSF under Grant No. EIA-0216467.

6. REFERENCES

[1] US Lattice Quantum Chromodynamics Software. `http://www.usqcd.org/usqcd-software`, 2006.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[3] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 115–125, Boulder, CO, USA, 1990.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[5] J. Bailey et al. MIMD lattice computation (MILC) collaboration. `http://www.physics.indiana.edu/~sg/milc.html`, 2006.

[6] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[7] W. W. Carlson et al. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Ctr. for Computing Sciences, May 1999.

[8] J. Caubet et al. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proc. of the Intl. Workshop on OpenMP Appl. and Tools*, pages 53–67, London, UK, 2001. Springer-Verlag.

[9] W. Chen et al. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th ACM International Conference on Supercomputing*, San Francisco, California, June 2003.

[10] C. Coarfa. *Portable High Performance and Scalability for Partitioned Global Address Space Languages*. PhD thesis, Department of Computer Science, Rice University, January 2007.

[11] C. Coarfa et al. An evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proc. of the* 10^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[12] D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin 2000 and Onyx2 performance tuning and optimization guide. Technical Report 007-3430-003, Silicon Graphics, Inc., 2001.

[13] L. DeRose, T. Hoover, and J. K. Hollingsworth. The dynamic probe class library-an infrastructure for developing instrumentation for performance tools. In *Proc. of the* 15^{th} *Intl. Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001.

[14] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-Array Fortran Compiler. In *Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September 29 - October 3 2004.

[15] N. Froyd, J. Mellor-Crummey, and R. Fowler. Efficient call-stack profiling of unmodified, optimized code. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, 2002.

[16] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *Proceedings of GCC Summit*, Ottawa, Canada, June 2006.

[17] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[18] R. J. Hall. Call path profiling. In *ICSE '92: Proceedings of the* 14^{th} *International Conference on Software Engineering*, pages 296–306, Melbourne, Australia, 1992. ACM Press.

[19] Krell Institute. Open SpeedShop for Linux. `http://www.openspeedshop.org`, 2007.

[20] P. E. McKenney. Differential profiling. *Software: Practice and Experience*, 29(3):219–234, 1998.

[21] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.

[22] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface Standard*, 1997.

[23] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1999.

[24] B. P. Miller et al. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[25] B. Mohr et al. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP*, Rome, Italy, 2002.

[26] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001. CD-ROM.

[27] S. Moore et al. *A Scalable Approach to MPI Application Performance Analysis*, volume 3666 of *Lecture Notes in Computer Science*, pages 309–316. Springer-Verlag, 2005.

[28] P. J. Mucci. PapiEx - execute arbitrary application and measure hardware performance counters with PAPI. `http://icl.cs.utk.edu/m̃ucci/papiex/papiex.html`, 2007.

[29] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[30] J. Nieplocha and B. Carpenter. *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer-Verlag, 1999.

[31] Benchmarking information referenced in the NSF 05-625 High Performance Computing System Acquisition: Towards a Petascale Computing Environment for Science and Engineering. Technical Report NSF0605, Nov. 2005.

[32] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, August 1998.

[33] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proceedings of Supercomputing 2004*, Pittsburgh, November 2004.

[34] D. A. Reed et al. Scalable performance analysis: The Pablo performance analysis environment. In *Proc. of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[35] Rice University. HPCToolkit performance analysis tools. `http://www.hipersoft.rice.edu/hpctoolkit`, 2007.

[36] Silicon Graphics, Inc. (SGI). SpeedShop User's Guide. Technical Report 007-3311-011, SGI, 2003.

[37] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[38] H.-H. Su et al. GASP! a standardized performance analysis tool interface for global address space programming models. Technical Report LBNL-61659, Lawrence Berkeley National Laboratory, 2006.

[39] N. Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. M.S. thesis, Department of Computer Science, Rice University, May 2007.

[40] The Climate, Ocean and Sea Ice Modeling (COSIM) Team. The Parallel Ocean Program (POP). `http://climate.lanl.gov/Models/POP`.

[41] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, 2000.

[42] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 240–250, NY, NY, USA, 2002. ACM Press.

[43] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proc. of the* 8^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, 2001.

[44] F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Julich, May 2004.

[45] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proc. of the European Conference on Parallel Computing*, Pisa, Italy, Aug. 2004.

[46] P. H. Worley. MPICL: a port of the PICL tracing logic to MPI. `http://www.epm.ornl.gov/picl`, 1999.

[47] C. E. Wu et al. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.

[48] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.