Barrier Elision for Production Parallel Programs

Milind Chabbi
Rice University, USA
milind.chabbi@rice.edu

Wim Lavrijsen
Lawrence Berkeley National Laboratory, USA
wlavrijsen@lbl.gov

Wibe de Jong
Lawrence Berkeley National Laboratory, USA
wadejong@lbl.gov

Koushik Sen
UC Berkeley, USA
{ksen}@cs.berkeley.edu

John Mellor-Crummey
Rice University, USA
johnmc@rice.edu

Costin Iancu
Lawrence Berkeley National Laboratory, USA
cciancu@lbl.gov

Abstract

Large scientific code bases are often composed of several layers of runtime libraries, implemented in multiple programming languages. In such situation, programmers often choose conservative synchronization patterns leading to suboptimal performance. In this paper, we present context-sensitive dynamic optimizations that elide barriers redundant during the program execution. In our technique, we perform data race detection alongside the program to identify redundant barriers in their calling contexts; after an initial learning, we start eliding all future instances of barriers occurring in the same calling context. We present an automatic on-the-fly optimization and a multi-pass guided optimization. We apply our techniques to NWChem—a 6 million line computational chemistry code written in C/C++/Fortran that uses several runtime libraries such as Global Arrays, ComEx, DMAPP, and MPI. Our technique elides a surprisingly high fraction of barriers (as many as 63%) in production runs. This redundancy elimination translates to application speedups as high as 14% on 2048 cores. Our techniques also provided valuable insight about the application behavior, later used by NWChem developers. Overall, we demonstrate the value of holistic context-sensitive analyses that consider the domain science in conjunction with the associated runtime software stack.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Design, Performance

Keywords Barrier Elision, Synchronization, PGAS, NWChem, HPC, Dynamic Analysis, Dynamic Optimization

1. Introduction

The scalability of parallel software is often determined by its synchronization behavior. Synchronization operations such as locks, barriers and fences not only introduce overheads, but also affect scalability of parallel applications. Consequently, synchronization optimizations have received a fair share of attention in both shared [15, 24] and distributed [20, 38] memory programming. Despite the large existing body of work and the perceived importance of the problem, these optimizations have not been deployed in "commercial" or production compilers for the dominant programming models in the HPC realm.

The MPI *library* standard 1.0 was ratified on May 5th 1994, and yet after 20 years and multiple compiled MPI efforts [14, 17, 21, 29], synchronization optimizations are performed manually in codes. The UPC *language* standard 1.0 has been ratified on May 13th 1999, yet after 15 years of compiler development [1, 3, 18] we are in a similar position. Part of this can be attributed to the significant engineering efforts required to build production quality compilers, and part can be attributed to the inherent conservative nature of static analysis. There is a real need for deploying synchronization optimizations in production quality tools to enhance the performance of production parallel software. As we continue to scale systems both up and out, the impact of communication optimization on end-to-end performance can only grow.

A *barrier* is a classical synchronization construct used in parallel programs. Every process executing a barrier waits until all other processes participating in the barrier arrive at the barrier. Since barriers involve system-wide communication, they incur high latency and scale poorly. Furthermore, a delay in one process to arrive at a barrier, either due to load imbalance or due to operating system stalls [10, 28], causes delay in all participating processes. In the ensuing text, references to a "barrier" mean such inter-process synchronization mechanism often used in Single Program Multiple Data (SPMD) programs. Other intra-process synchronization mechanisms such as barriers between CPU threads within a same process, memory barriers a.k.a fences, and barriers between multiple hardware threads within a GPU, which are homonymous with the term barrier, are not pertinent to our discussion.

A barrier may be necessary *in only some execution contexts, but not all contexts*. Redundant barriers in *hot* execution contexts offer an excellent opportunity for optimizing communication-intensive scientific codes. In this paper, we discuss our experiences building an application-specific dynamic optimization able to detect and skip barriers that are redundant in their runtime calling contexts. The application is the NWChem [37]— a computational chemistry code with more than 6 million lines of C, Fortran77, and C++ code. NWChem is implemented on top of multiple abstraction layers for

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02…$15.00.
http://dx.doi.org/10.1145/2688500.2688502

data and communication. NWChem includes software abstractions for its own internal tasking, load balancing, memory management and checkpoint/restart mechanisms. The abstractions used for software modularity hinder optimizations. Specifically, communication libraries written for generic use introduce context-sensitive redundant synchronization that become a handicap in scaling NWChem to take full advantage of emerging supercomputers.

Our idea to address the context-sensitive redundant barrier synchronization is very simple: execute the program alongside a dynamic data race detector, identify barriers by their calling contexts (program stack), determine if they are redundant in a particular context, then speculatively skip future executions in the same calling context. In our definition, a barrier is redundant *if and only if* skipping the operation introduces no data races. Since precise data race detection is *not* our objective, we can afford to employ a coarse-grained data race detection mechanism that *may* report false positives. By relaxing precision, we achieve *low overhead* in our analysis—a key requirement for dynamic optimizations. To our knowledge, both the idea of assessing the redundancy of synchronization using online, coarse-grained data race detection and the context-sensitive elision are novel. We explore an automated online optimization and a multi-pass guided optimization. In the rest of this paper, we use the terms on-the-fly, online, and one-pass interchangeably. Similarly, we use the terms offline, multi-pass, and feedback-based optimization interchangeably.

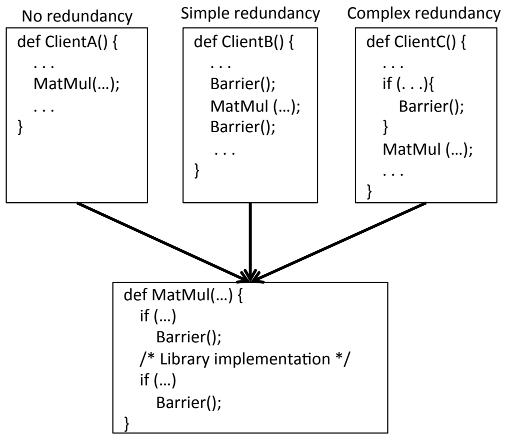
The online algorithm first learns the dynamic "characteristics" of barriers, then speculatively skips the redundant barriers. The approach is input independent and required detailed application-specific knowledge to build an ad-hoc data race detector running with less than 1% runtime overhead. *The detector is inherently conservative and engineered to not provide false negatives.*

In the multi-pass approach, we execute the program on several training inputs and collect a large set of calling contexts where the barriers may be redundant in an execution. We classify contexts into equivalences classes, where all contexts in an equivalence class share a common subcontext. Subcontexts capture the intuition that the barrier redundancy inside a library module is determined by prior barrier calls performed by its callers. Subcontexts provide useful insights about places of redundancy in the code. We then use a directed testing approach to execute the program, skipping any barrier whose full calling context contains any of the subcontexts designated for elision. Test-driven validation builds developer confidence in the elision process. The set of subcontexts that both pass testing and/or developer inspection become inputs for production-time barrier elision.

Online and offline techniques were able to elide respectively up to 45% and 63% of all barriers encountered during NWChem science production runs. Eliding barriers resulted in end-to-end application runtime improvements up to 14%, when running on 2048 cores. In addition, our approaches provide valuable tools for program understanding. We have uncovered several redundant barriers, e.g. in the ComeEx memory allocation module, which we then removed from the source repository version of NWChem.

Our approaches use dynamic program analysis and speculation. They are guaranteed to catch bad speculation at runtime and abort execution, but this may be undesirable at large scale. Bad speculation has not happened in our productions runs and as any other program transformation tool, developer confidence in their applicability has to be gained through testing coverage. Modern scientific programs utilizing the checkpoint-restart technique make our technique even more attractive.

Overall, this work showcases the benefits of a holistic approach to code analysis. We analyze computational chemistry solvers in conjunction with runtime implementations using context-sensitive analyses. This approach uncovers code patterns that induce redundant synchronization. As science moves towards multiscale multiphysiscs simulations, we expect context-sensitive analyses to become invaluable program optimization tools.



```
def ClientA() {
   . . .
   MatMul(…);
   . . .
}
```

```
def ClientB() {
   . . .
   Barrier();
   MatMul (…);
   Barrier();
    . . .
}
```

```
def ClientC() {
   . . .
   if (. . .){
      Barrier();
   }
   MatMul (…);
   . . .
}
```

```
def MatMul(…) {
   if (…)
      Barrier();
   /* Library implementation */
   if (…)
      Barrier();
}
```

Figure 1. Composition of synchronized API calls.

The rest of the paper is organized as follows. Section 2 describes the aspects of synchronization in modern scientific codes and Section 3 provides the necessary properties of safe barrier elision. Section 4 sketches the online barrier elision technique, while Section 5 presents the offline analysis approach. Section 6 highlights the structure of NWChem together with implementation details of our infrastructure. Section 7 provides the insights we gained about NWChem via our techniques. Section 8 presents an empirical evaluation, with further discussion in Section 9. Section 10 presents the related work, and finally Section 11 provides conclusions.

2. The Problem with Synchronization

Large scientific applications [16, 33] employ a hierarchy of libraries to implement layered abstractions. In the absence of contextual knowledge, libraries are designed for generality in such a way that any parallelism is quiesced upon entry and exit from the respective module. As tracking individual dependencies is challenging, the synchronization usually involves heavyweight operations such as barriers and fences. For example, ScaLAPACK [4] calls use collective synchronization semantics, which may hinder the overall application scalability. Application programmers may also add synchronization to ensure semantic guarantees when employing libraries for asynchronous operations. In cases where a lower-level library routine provides stronger synchronization guarantees than advertised in its interface specification, the ensuing communication redundancy causes non-trivial performance overheads.

Figure 1 illustrates scenarios where transitions from higher-level abstractions to lower-level abstractions can sometimes cause redundant barriers. The `ClientA()`$\rightarrow$`MatMul()` transition has no redundancy since the `ClientA()` relies on the barriers provided by the `MatMul()` function in the lower-level library. The `ClientB()`$\rightarrow$`MatMul()` and `ClientC()`$\rightarrow$`MatMul()` transitions can make some barriers redundant, and it is progressively more complex to detect or eliminate such redundancies. Eliminating barriers in the `MatMul()` function breaks the code in the `ClientA()`, whereas keeping barriers in the `MatMul()` function causes redundancy in other clients. As shown later in Section 7 this is a common occurrence in NWChem.

Static program analysis [15, 20, 24, 38] has been successfully used for synchronization optimizations and it may be able to handle our example. However, static analysis faces great engineering challenges when dealing with the characteristics of existing full-fledged HPC applications which: 1) use combinations of multiple

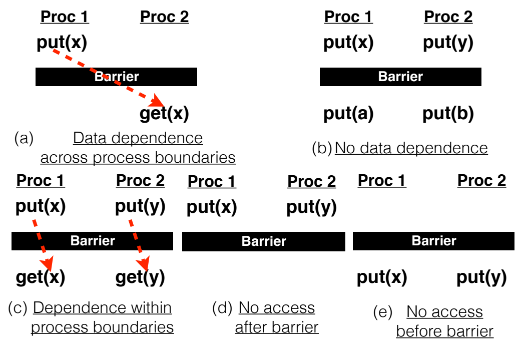


Figure 2. Data dependence and barrier redundancy

languages, such as C/C++/Fortran; 2) are written in a modular fashion with calls into manifold "libraries"; and 3) are built on layers with different semantic abstractions. While language designers [7, 11, 12] and application developers are striving to expose concurrency inside an application, software engineering practices (modularity, composability) and development tools (multiple compilers) are busy killing it.

Over-conservative synchronization already appears in the current generation of HPC codes. This is likely to be pervasive in the next generation of codes designed for extreme scaling. As the scientific community is moving towards multiphysics multiscale simulations, HPC codes are universally refactored as compositions of parallel libraries, solvers and runtimes. The next generation of codes which employ Domain Specific Languages (DSL) [22, 34] or high-productivity languages such as Chapel [11] will exhibit similar characteristics, as their compilers use source-to-source translation with calls into libraries implementing the language-level abstractions. In these cases statements are compiled mostly independently from one another into complicated hierarchies of parallel calls.

Reasoning about synchronization is particularly challenging in codes that use Partitioned Global Address Space (PGAS) abstractions and one-sided Remote Direct Memory Access (RDMA) [11, 12, 25, 26] based communication. Memory in the PGAS model is classified as either local or global. Local memory can be accessed only by a single task. Global memory can be accessed by any task using load/store instructions or RDMA operations. Unlike MPI, where Send/Recv pairs couple data transfers and synchronization, in PGAS the two are decoupled.

The optimizations thereinafter are designed to eliminate redundant barrier operations in the NWChem computational chemistry code described in Section 6. NWChem combines PGAS and RDMA concepts; the scientific community foresees that future codes will employ at least one of these mechanisms.

3. Reasoning About Barrier Elision

A barrier may be redundant if there are no data dependence edges (*read-write, write-read, or write-write*) originating from before the the barrier on one task and terminating after the barrier on another task. Figure 2(a) shows a barrier necessary to resolve a write-read conflict. Figures 2(b)- 2(e) show several cases of redundant barriers with no data dependence between processes across the barrier. A data race detector finds dependencies by tracking individual addresses accessed by the processing elements (PEs) before a barrier and comparing these with accesses after the barrier.

In practice, address tracking for data race detection may incur higher costs than barrier elision could hope to gain. For example, Park et al. [27] describe a precise data race detector using dynamic analysis for PGAS programs with one-sided communication, which incurs 50% overhead at 2000 cores. Their technique is directly applicable here, but its runtime overhead is higher than the total time spent by NWChem in barriers (up to 20%) in our experiments.

Memory accesses in PGAS models can be distinguished as follows:

- N - access to memory that is private to a processing element (PE) and cannot be accessed by any other PE.
- L - access to memory that has affinity with one PE, which can perform load/stores into it. Access from any other PE involves RDMA or other calls into the runtime.
- R - access to memory that is remote and can be accessed only using RDMA.

To achieve lower overhead, we over-approximate the dependencies by assuming that any access (L,R) of shared data *before* the barrier can alias with any (L,R) shared data access *after* the barrier. Furthermore, we don't distinguish a write from a read, and treat any access as a write operation. Because precise knowledge of individual addresses is no longer needed, this assumption reduces the overhead of instrumenting each memory access and simplifies analysis. We only need to intercept access to shared data.

In the following, the term "barrier", denoted by $\mathbf{B}$, refers to a dynamic instance of a barrier operation, regardless of its source code location. We associate a memory access summary value (S_i) with any portion of the program executed between two barrier operations. This access summary has one of the (N,L,R) values and it is computed as the transitive closure of the types of all memory accesses in that particular code region on i^{th} PE. The access summary is computed using the meet operation described below.

Definition 1 (ACCESS: $\bigwedge$). *N, L, and R form a monotonically descending lattice, where the meet operation ($\bigwedge$) between two types of accesses is defined as follows:*

$$\begin{aligned}
N \bigwedge N & &= N\\
N \bigwedge L = L \bigwedge N &= L\\
N \bigwedge R = R \bigwedge N &= R\\
L \bigwedge L & &= L\\
L \bigwedge R = R \bigwedge L &= R\\
R \bigwedge R & &= R
\end{aligned}$$

Intuitively, this provides a hierarchy of observability for memory accesses. Remote operations (R) take precedence over all other types. Operations on shared data with local affinity (L) take precedence over access to private data (N).

3.1 Ideal Barrier Elision

Given any execution trace $S_i^b\mathbf{B}S_i^a$, where S_i^b and S_i^a are the memory access summaries of task i before and after the barrier $\mathbf{B}$, we compute a trace with global access summaries as $S^b\mathbf{B}S^a$, where $S^b = \bigwedge_{i=0}^{Procs} S_i^b$ and $S^a = \bigwedge_{i=0}^{Procs} S_i^a$.

For any sequence $S^b\mathbf{B}S^a$, there can be nine different orderings of global access summaries—shown in column 2 of Table 1. For each combination, we can decide whether the barrier is redundant based on the observability of the access summary. When a barrier is deleted the global access summaries before and after it need to be combined into a single value, $S^b \bigwedge S^a$. If the barrier is retained, the access summary before the barrier has no relevance to what ensues after the barrier and hence remains S^a. As Table 1 shows, in six of the nine variations, the barrier can be safely eliminated. Intuitively, any barrier preceded by remote accesses R is retained, any barrier

Table 1. The rules that dictate allowable transformations, given an observed execution trace $S^b\mathbf{B}S^a$. The new memory access summary comes into effect when the transformation is applied.

Rule	Trace	Transformation	Memory access summary
1	$N\mathbf{B}N$	$N\cancel{\mathbf{B}}N$	$N \bigwedge N = N$
2	$N\mathbf{B}L$	$N\cancel{\mathbf{B}}L$	$N \bigwedge L = L$
3	$N\mathbf{B}R$	$N\cancel{\mathbf{B}}R$	$N \bigwedge R = R$
4	$L\mathbf{B}N$	$L\cancel{\mathbf{B}}N$	$L \bigwedge N = L$
5	$L\mathbf{B}L$	$L\cancel{\mathbf{B}}L$	$L \bigwedge L = L$
6	$L\mathbf{B}R$	$L\mathbf{B}R$ (none)	R
7	$R\mathbf{B}N$	$R\cancel{\mathbf{B}}N$	$R \bigwedge N = R$
8	$R\mathbf{B}L$	$R\mathbf{B}L$ (none)	L
9	$R\mathbf{B}R$	$R\mathbf{B}R$ (none)	R

Table 2. Example application of rules from Table 1.

$$\underbrace{N\mathbf{B_1}L}_{\text{Rule 2}}\mathbf{B_2}N\mathbf{B_3}R$$
$$N\cancel{\mathbf{B_1}}\underbrace{L\mathbf{B_2}N}_{\text{Rule 4}}\mathbf{B_3}R$$
$$N\cancel{\mathbf{B_1}}L\cancel{\mathbf{B_2}}\underbrace{L\mathbf{B_3}R}_{\text{Rule 6}}$$
$$N\cancel{\mathbf{B_1}}L\cancel{\mathbf{B_2}}L\mathbf{B_3}R$$

that neighbors purely local accesses N can be elided, as well as barriers surrounded by shared accesses with local affinity L.

Given a trace, the rules can be applied in any order. Considering a sample trace $N\mathbf{B_1}L\mathbf{B_2}N\mathbf{B_3}R$, Table 2 shows a sequence of valid barrier elision transformations.

While providing good opportunity for barrier elision, there are several challenges to implementing this approach:

1. The transformations require inspecting the access summary both *before* and *after* the barrier.
2. The transformation performed at one barrier affects the resulting access summary after the barrier, hence the transformation possible at the subsequent barrier. In the previous example, applying Rule 4 at barrier B_2 changes the access summary after B_2 from N to L; consequently, at B_3, one can't use Rule1 $N\mathbf{B_3}R \implies N\cancel{\mathbf{B_3}}R$. Thus, an optimal barrier deletion algorithm *requires processing the whole program execution trace.*
3. Knowing the system-wide access summary at a barrier *requires a communication with all processes*, which defeats the purpose of deleting the barrier.

3.2 Practical Barrier Elision

For practical reasons we adopt a simplified approach that uses *only* information about the memory access summary *before* a barrier in a manner that is independent of the order of the barrier elision. From Table 1 the following is apparent:

1. Any barrier preceded by purely local accesses (global access summary N) can be elided.
2. Elision of barriers preceded by purely local access does not affect the "redundancy" of any barriers that follow.

Indeed, consider the execution trace $N\mathbf{B_1}X\mathbf{B_2}Y$, where $X, Y \in \{N, L, R\}$. After the $N\cancel{\mathbf{B_1}}$ transformation, the access summary before $\mathbf{B_2}$ is $N \bigwedge X = X$. Hence, deleting $\mathbf{B_1}$ has no effect on the access summary before $\mathbf{B_2}$. Consequently, the transformations possible at $\mathbf{B_2}$ are unaffected by the transformation performed at $\mathbf{B_1}$.

We summarize our simplified transformation rules in Table 3. We apply the simplified transformation to the earlier example in order in Table 4.

These rules capture three out of the previous six cases where elision is possible and performed really well in practice for NWChem.

Table 3. Simplified rules used in our implementation.

Rule	Trace	Transformation
1	$N\mathbf{B}$	$N\cancel{\mathbf{B}}$
2	$R\mathbf{B}$	$R\mathbf{B}$ (none)
3	$L\mathbf{B}$	$L\mathbf{B}$ (none)

Table 4. Example application of rules from Table 3.

$$\underbrace{N\mathbf{B_1}}_{\text{Rule 1}}L\mathbf{B_2}N\mathbf{B_3}R$$
$$N\cancel{\mathbf{B_1}}\underbrace{L\mathbf{B_2}}_{\text{Rule 3}}N\mathbf{B_3}R$$
$$N\cancel{\mathbf{B_1}}L\mathbf{B_2}\underbrace{N\mathbf{B_3}}_{\text{Rule 1}}R$$
$$N\cancel{\mathbf{B_1}}L\mathbf{B_2}N\cancel{\mathbf{B_3}}R$$

Some codes using PGAS paradigms but optimized for locality may have barriers surrounded by L summaries, i.e. accesses in the global space but with local affinity. Our simplified approach will classify these barriers as necessary. Note that this case is optimized explicitly by the NWChem developers.

The final issue is that of necessity of performing communication in order to compute the global access summary before a barrier. We identify a barrier by its call path, a.k.a. *calling context*. We assume that *a barrier that is repeatedly redundant in a calling context is likely to remain redundant for rest of the execution under the same calling context.* This behavior forms the basis of our online elision technique described in detail in Section 4.

Alternatively, the program behavior can be observed for entire training executions at small scale and barriers that are always redundant in some calling contexts can be identified in a postmortem analysis. A production run can elide a barrier whenever its calling context matches that of the preprocessed-designated calling contexts. This approach forms the basis of our offline analysis discussed in detail in Section 5.

Because of this speculative elision, our techniques work well for the repeatable behavior present in indirect HPC solvers. For other programs, a future iteration or a production run might behave differently than what has been learned for a calling context. In these cases, our approach is guaranteed to catch and report the misspeculation.

4. Automatic Single-Pass Barrier Elision

The online barrier elider works in two phases: it learns about barriers within their full calling contexts and then speculatively elides those deemed redundant (see Algorithm 1).

We identify each barrier $\mathbf{B}$ by its calling context c, represented as $\hat{B}_c$. We encode the calling context as a hash, $H(c)$, formed from the call chain starting from `main` to the current barrier inclusive of all functions[1] along the path. The algorithm maintains a monotonically increasing barrier sequence number, incrementing on encountering each barrier, to distinguish between different barrier episodes.

During the learning phase, we observe and memorize if a barrier is redundant in its calling context across all participating processes. Our algorithm *replaces the barrier with a reduction*[2] to compute the global access summary. The reduction performs a *min* operation $S^b = \bigwedge_{p=0}^{Procs} S_p^b$ on the local access summaries (S_p^b) before the barrier. A global access summary result can be either R or N.

[1] Return address of the callee, to be specific.

[2] On current systems, the cost of a reduction is comparable to a barrier, if the message sent is small (eight bytes in our case) and the reduction operation is short (less than 10 instructions in our case).

Algorithm 1: Automatic on-line barrier deletion

```
Input: $p = self$ /* implicit process identifier */
Result: SUCCESS for elided barrier or return value from a participated barrier
enum {PARTICIPATE=0, SKIP=1, LEARNING=THRESHOLD+SKIP, CB=LEARNING+1}
/* ctxtId is same as H(c) in the prose. */

ctxtId = Hash(GetCallingContex())
/* dict is a dictionary of <context hash, memorized transformation> */

if $ctxtId \notin dict$ then
    /* First visit to this barrier */
    val = MinReduce(in $S^b_p$, out $S^b$, ...)
    if $S^b$ == $N$ then
        dict[ctxtId] = LEARNING
    else //$S^b \neq$ N
        dict[ctxtId] = PARTICIPATE
        /* reset local state */
        $S^b_p$ = N
    return val
if $S^b_p$ == $N$ then
    switch dict[ctxtId] do
        case SKIP
            return SUCCESS
        case PARTICIPATE
            return MinReduce(in N, ... )
        otherwise
            /* Learning */
            val = MinReduce(in N, out $S^b$, ...)
            if $S^b$ == $N$ then
                /* When dict[ctxtId] reaches SKIP, we will start eliding */
                dict[ctxtId] −−
            else // $S^b \neq$ N
                dict[ctxtId] = PARTICIPATE
            return val
else // $S^b_p \neq$ N
    switch dict[ctxtId] do
        case PARTICIPATE
            val = MinReduce(in $S^b_p$, ...)
        case SKIP
            /* Breaking consensus, Optimistic */
            val = MinReduce(in CB, out $S^b$,... )
            if $S^b \neq CB$ then
                /* Not all PEs broke consensus. */
                Report misspeculation.
            /* else, All PEs broke the consensus, program is safe. */
        otherwise
            /* Veto in skipping */
            val = MinReduce(in $S^b_p$, out $S^b$, ...)
            dict[ctxtId] = PARTICIPATE
    /* reset local state $S^b_p$ */
    $S^b_p$ = N
    return val
```

- If R, then each PE memorizes $\hat{B}_c$ as necessary, and learning stops for that barrier; *any barrier deemed necessary once, will remain classified as necessary.*
- If N, then each PE records $\hat{B}_c$ as a candidate for elision, and learning continues. If subsequent $threshold$ number of visits to $\hat{B}_c$ also result in the N state, then $\hat{B}_c$ is promoted to an elidible barrier; *any barrier once classified as redundant will be skipped for the rest of the execution.*

Speculation can fail, but the implementation will detect any misspeculation and then tries to recover, or aborts the execution. The implementation continues to maintain up-to-date global access summaries between barriers and piggybacks the summary for elided barriers onto the barriers still executed. If a PE decides that a barrier previously elided became necessary, there are two options:

1. *Pessimistic.* Record misspeculation (and abort).
2. *Optimistic.* Assume that all PEs detect the same broken consensus, allowing all to participate in the barrier and maintain the integrity of the system.

We take the optimistic approach and replace the barrier with a *min* reduction with a special status CB, representing the fact that the consensus is broken. The meet operation is augmented such that $CB \bigwedge CB = CB$, $CB \bigwedge R = R$, $CB \bigwedge N = N$. If the resulting global status after the reduction is also CB, it means that all PEs broke the consensus and each one participated in the synchronization. If such is the case, we have two options:

1. *Pessimistic.* Locally downgrade the context $\hat{B}_t$ as necessary in all PEs to avoid future consensus breaks.
2. *Optimistic.* Assume the pattern will hold and expect future instances of $\hat{B}_t$ to be skippable or equally recoverable, and thus retain the barrier as redundant.

We take again the optimistic approach. Finally, a subtle case arises when some PEs break consensus for a specific barrier and wait in a reduction operation with the CB state, while other PEs skip that barrier, advance, and break consensus in a later, different, barrier. This leads to two mismatching reduction operations, and possible incorrect behavior or even deadlock. To handle this case, each PE passes the aforementioned unique barrier sequence number in each reduction operation; if the sequence numbers do not match, we record the misspeculation.

Training Threshold: The duration of the learning phase is tunable. Shortening the learning phase might increase the number of barriers classified as redundant, but it might also increase the chance of misspeculation. In the current implementation we use a static threshold determined through testing. The value can also be selected at runtime by developers. We discuss more details about learning in the context of NWChem in Section 8.

For an arbitrary application, the suggested method to choose the right training threshold is to first run the application on training inputs. During training runs, we log the decisions for all the barriers in the program without performing the barrier elision. Then, we analyze the logs offline to determine the upper bound on repetitions before a barrier stabilizes. We can use this threshold for the actual production runs where the barrier elision will be performed.

We have not encountered misspeculation in our experiments. NWChem contains a checkpoint-restart feature. We plan to blacklist misspeculated barriers and replace abortions with restarts.

5. Guided Multi-Pass Barrier Elision

Figure 3 presents the workflow for a multi-pass procedure that separates learning about barrier redundancy into offline training. The training inputs are small-scale representatives of various real inputs. After learning, the redundant contexts are classified, validated, and approved for consumption by a *barrier elider* module for production runs. The offline analysis has different characteristics from the online transformation:

1. It increases the chance of deleting the barriers missed during online learning.
2. It provides developers with an opportunity to inspect redundant contexts.
3. It increases developer confidence by inducing a validation step.

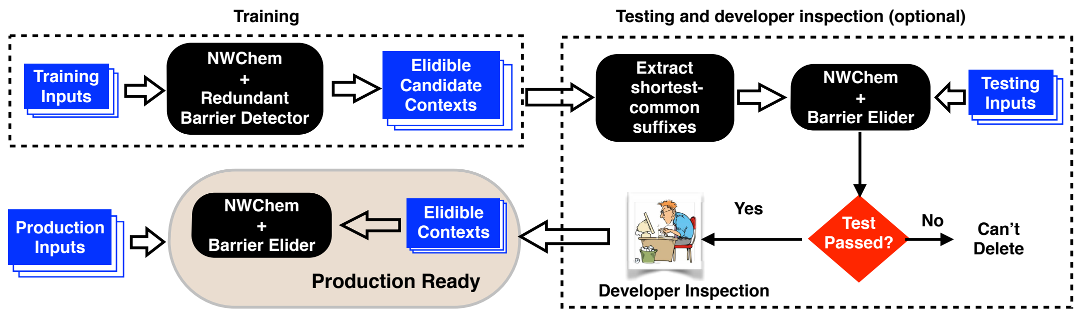


Figure 3. Workflow of offline barrier elision.

We start with a set of training inputs and run a redundant barrier detector (RBD) alongside the execution. Currently, we use the same detector from the online analysis, but note that this can be replaced with any other race detector, even a more imprecise one. The RBD observes each barrier in its full calling context, classifying it as either redundant (if all operations preceding the barrier are always private) or necessary (if at least one operation preceding the barrier was not private). RBD logs *all* barrier contexts for each test input.

We then perform an offline analysis, where we merge all redundant barrier contexts from all training runs into a single set (RBC) and all all non-redundant barrier contexts into another set ($NRBC$). In this phase, we classify a barrier in a calling context as redundant if it appears in the set of redundant barrier contexts but not in the set of non-redundant barrier contexts (in any inputs). This forms the set of Elidible Candidate Contexts $ECC = RBC - NRBC$.

Sub-Context Classification: We classify the contexts in ECC based on the intuition that in modular software paths through a certain library module are probably determined by initial conditions set by the module's callers. In this case there may be paths local to the module where a barrier is always redundant and there may be callers that always exercise these. This is captured by the common parts of the calling context leading to redundant barriers, referred to as sub-contexts.

Consider the example in Figure 4. Module 1 always forces a barrier before calling Module 4, which eventually calls a barrier—represented by the calling context suffix $N \to B$. Barriers called through myriad call paths all sharing the suffix $M \to N \to B$ are redundant. However, the same is not true for Modules 2 and 3, which do not enforce a barrier when calling a barrier through context suffix $N \to B$. This provides the insight that the call-path suffix $M \to N \to B$ causes a redundant barrier, whereas the suffix $N \to B$ alone is not redundant. Inspecting myriad redundant full call paths such as $\{main \to \cdots W \to M \to N \to B\}, \cdots,$ $\{main \to \cdots Z \to M \to N \to B\}$ is tedious and cannot provide this type of insight.

Based on the aforementioned observation, our subcontext classification employs the Algorithm 2 to find the *shortest common suffix* of a context inside all redundant barriers. The algorithm groups redundant barrier contexts in equivalence classes such that each class can be represented by a shortest common suffix. The *distinguishing* shortest common suffix meets the following two criteria:

1. It does not appear as a suffix of any contexts in the non-redundant barriers set ($NRBC$). This ensures that the barrier is *redundant*.
2. The suffix of length one less is present in the ($NRBC$) set. This ensures that the *distinguishing* suffix is the *shortest* to classify the subcontext as redundant.

The algorithm terminates since ECC monotonically decreases in size reaching $\emptyset$, when the length of suffix equals the maximum length context. In the aforementioned example, the call paths $\{main \to \cdots W \to M \to N \to B\}, \cdots, \{main \cdots Z \to M \to N \to B\}$ will be classified into one equivalence class represented by the common suffix $M \to N \to B$.

The output of this stage is the set ECS, which contains subcontexts necessary to designate a barrier as redundant.

Test Validation: The test module orders the sets in ECS by their cardinality, i.e. the number of redundant barriers that share a particular sub-context, picks the largest one, and starts extending the set including one sub-context at a time to test. The testing employs a *barrier elider* module that elides barriers whose runtime calling contexts match the sub-contexts being tested. Test validation cross-checks against the expected results.

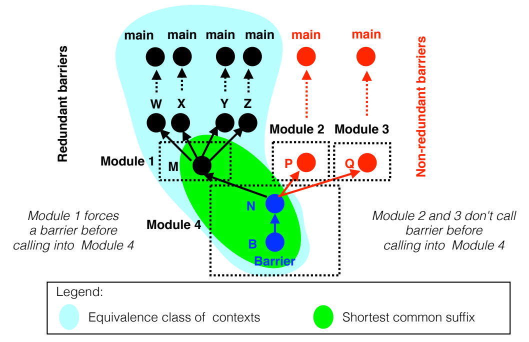

Redundant barriers
main
main
main
main
main
main
Non-redundant barriers
W
X
Y
Z
Module 1
M
Module 2
Module 3
P
Q
N
B
Barrier
Module 4
Module 1 forces a barrier before calling into Module 4
Module 2 and 3 don't call barrier before calling into Module 4
Legend:
Equivalence class of contexts
Shortest common suffix

Figure 4. Suffix of redundant barriers.

Algorithm 2: Barrier suffix context classifier

```
   Input: NRBC/*Non-redundant Barrier Contexts*/
   Input: RBC/*Redundant Barrier Contexts*/
   Output: ECS/*Elidible Context Suffixes*/
1  ECC = RBC-NRBC /*Elision candidate contexts*/
2  ECS = $\emptyset$
3  len = 1
4  while $ECC \neq \emptyset$ do
5      /*Find suffixes of length=len*/
6      A = {Suffix (C, len) $\forall$ C $\in$ ECC}
7      B = {Suffix (C, len) $\forall$ C $\in$ NRBC}
8      /*Find elidible suffixes of length=len*/
9      S = A-B
10     ECS = ECS $\bigcup$ S
11     /*Remove classified contexts from ECC*/
12     ECC = ECC - {$\forall$ C $\in$ ECC s.t. Suffix (C, len) $\in$ S}
13     len++
14 return ECS
```

We also present the sorted ECS and the testing results to the developer for further investigation. The developer then has the option to further filter the contexts based on intuition.

Production-Time Barrier Elision: The developer-approved contexts become inputs to production runs. The production runs use a barrier elider module to skip barriers whose contexts match the developer approved elidible contexts. We present the details of insights gained from our technique in Section 7. These production runs have the same safety guarantees of identifying misspeculation and the possibility of restarting as the online algorithm.

6. Barrier Elision in NWChem

6.1 NWChem Scientific Details

NWChem [37] is a computational chemistry code widely used in the scientific community. It provides a portable and scalable implementation of several Quantum Mechanics and Molecular Mechanics methods: Coupled-cluster (QM-CC), Hartree-Fock (HF), Density functional theory (DFT), Ab initio molecular dynamics (AIMD) etc. The results in this paper are for high-accuracy QM-CC, where many of the NWChem computational cycles are spent. QM-CC is by far the most computationally intensive and therefore the most scalable method in NWChem. The other methods have increasing demand for global synchronization, such as the HF method that generates starting vectors for QM-CC. As we move along to the methods that demand more synchronization, we expect the optimizations developed in this paper to deliver even better performance gains. Two different science production runs were used to evaluate the benefits of barrier elision:

DCO: Accurate simulation of the photodissociation dynamics and thermochemistry of the dichlorine oxide (Cl_2O) molecule. This simulation is important for understanding the catalytic destruction of ozone in the stratosphere, where this molecule plays the role of reaction intermediate.

OCT: Simulation of the thermochemistry and radiation absorption of the oxidized cytosine molecule ($C_4H_6N_3O_2$). This simulation is key to understanding the role of oxidative stress and free radical induced damage to DNA.

Both runs first perform a HF simulation to obtain starting vectors. Subsequently, each run performs a different type, but algorithmically similar, QM-CC simulation.

6.2 NWChem Code Structure

NWChem contains more than 6 million lines of code written in C, C++, Fortran and Python. In addition to the chemistry solvers (in Fortran, C++, Python), it contains a complicated runtime infrastructure (in C) that implements support for tasking, load balancing, memory management, resiliency, communication and synchronization. Communication and synchronization in NWChem is handled across multiple software modules. Global Arrays [26] (GA) provide shared memory array abstractions for distributed memory programming with primitives such as Get and Put on array sections. GA is implemented on top of MPI, as well as the Aggregate Remote Memory Copy Interface (ARMCI) [25] and Communication runtime for Exascale (ComEx) [13] communication libraries. These libraries make the Global Arrays layer portable by hiding the low-level RDMA and synchronization primitives. ARMCI and ComEx are implemented on top of native communication APIs such as Cray DMAPP, InfiniBand Verbs, and IBM PAMI, among others. Our analysis and instrumentation spans all layers from chemistry to DMAPP.

Interestingly, we found places in NWChem, where the developers have hand optimized the code to elide unnecessary synchronizations. Such cases are present in situations where all tasks perform local operations. However, hand optimization is very limited in its scope. It does not address redundancies that happen across several layers of abstractions in a highly modular software. We believe these are all characteristics of future scientific codes for the Exascale era. Our work makes the following useful observations related to this type of code architectures: 1) as modular software implies "modular" contexts, flow sensitive and context sensitive analyses yield good results; and 2) as runtimes are written with portability in mind, holistic dynamic analyses that examine applications and runtimes in conjunction yield good results.

6.3 Instrumenting NWChem

The important design concerns related to instrumenting the program execution for our analyses are portability and overhead.

We aim to provide portability of NWChem together with the analysis to other hardware. We also aim to provide a portable analysis that can be applied to other code infrastructures. Our design choice was to intercept the lowest-level RDMA APIs to identify all remote memory accesses, while our algorithm and workflow remains independent of the application, run-time environment, and the hardware. We achieve this by performing link-time [3] wrapping of lowest-level communication libraries and diverting the necessary calls through our instrumentation layer. Library interposing via `LD_PRELOAD` will work similarly. Other portable runtimes such as GASNet also present a single interface that resolves into system specific API calls. Note that portability is enabled by the fact that we are interested in the presence of these operations at runtime, rather than their exact semantics.

On the Cray hardware, the DMAPP layer provides a low-level system communication API. In any scientific application running on Cray's supercomputers, the higher-level communication abstractions eventually resolve to some DMAPP calls for RDMA operations. By link-time redefinition of DMAPP RDMA operations, we can intercept several configurations of NWChem such as its Global Arrays abstraction running on ARMCI, ComEx, or MPI. We have also audited the InfiniBand verbs API and believe our approach trivially ports to that system API.

To correctly elide synchronization, the analysis needs to precisely detect accesses to local memory that bypass the standard RDMA calls for efficiency. We provide a plugin functionality that the developers can use to mark regions of code that perform such bypassing. As overhead and precision are a concern, the art is to identify the right level of granularity. With the cooperation of NWChem developers we have audited the code, identified operations at different levels of abstraction and understood their side-effects. We then manually inserted the instrumentation at only two places in the NWChem code to recognize all local accesses that have global visibility. These are the `comex_get_nbi()` and `comex_put_nbi()` calls, which resolve in either remote-access DMAPP calls or local memory accesses.

Another way by which local accesses can have global effects is by applications gaining a local pointer to the shared address space. Global Arrays have a well-defined interface for gaining a pointer access (`ga_access`) and releasing the access (`ga_release`) to the local shared regions. By keeping track of the number of outstanding accesses, we conservatively disable the execution regions where it is unsafe to elide barriers. Several levels of global memory abstraction and the fact that "casting" sometimes occurs in an unprincipled manner based on code knowledge have complicated this process. Augmenting our offline analysis with a full-fledged data race detector would simplify this process. A principled approach to creating

[3] When callers and callees are defined in the same file, link-time approaches need to be augmented with source modifications.

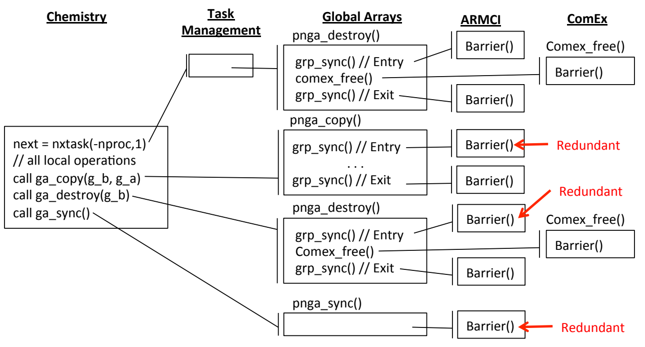


Figure 5. Barrier synchronization in Hartree-Fock solver. Nine barriers, across five modules written in Fortran and C.

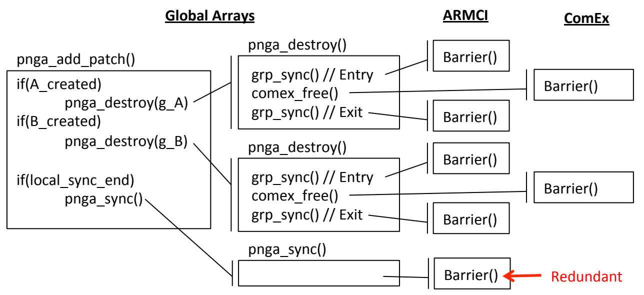


Figure 6. Context- and flow-sensitive redundant barriers: each pnga_destroy performs 3 barriers.

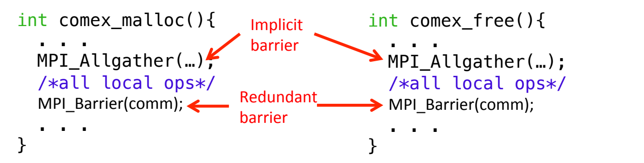
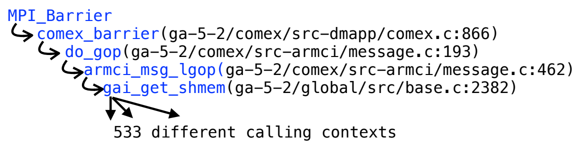

```
int comex_malloc(){        int comex_free(){
  . . .                      . . .
  MPI_Allgather(…);          MPI_Allgather(…);
  /*all local ops*/          /*all local ops*/
  MPI_Barrier(comm);         MPI_Barrier(comm);
  . . .                      . . .
}                          }
```

Figure 7. Context-insensitive redundant barriers in ComEx.


```
MPI_Barrier
↪comex_barrier(ga-5-2/comex/src-dmapp/comex.c:866)
  ↪do_gop(ga-5-2/comex/src-armci/message.c:193)
    ↪armci_msg_lgop(ga-5-2/comex/src-armci/message.c:462)
      ↪gai_get_shmem(ga-5-2/global/src/base.c:2382)

          533 different calling contexts
```

Figure 8. Common subcontext of a top redundant barrier in NWChem.

aliases between global and local pointers or using smart pointers can ease the task of a dynamic analysis frameworks in Partitioned Global Address Space programs.

6.4 Managing Execution Contexts

In our implementation, we identify contexts by unwinding the stack and computing a hash value based on the frame return address. We used `libunwind` [23] for unwinding and Google dense hash tables [35] for fast searching of contexts. We compiled the application by enabling frame pointers, and the overhead of unwinding was negligible in our experiments.

The online algorithm requires the calling contexts to be contextually aligned, i.e. the barriers do not need to be textually aligned [20], but their calling contexts always match the same way. At a barrier's calling context for a process, if the the calling contexts of other processes participating in the same barrier do not remain stable, it will hinder the learning process. Conservatively, we do not elide any barriers with contextual misalignment. To detect contextually misaligned barriers, we pass the context hash of each barrier in the reduction operation during the learning process. We drop contexts from eliding if we detect misalignment during learning. We note that the context hash can be different on different process for dynamically loaded binaries. By canonicalizing the instruction pointer as a <load-module:instruction-offset> pair, we can obtain system-wide consistent context hashes. One can also use program debugging information to construct canonical context hashes.

6.5 Portability to Other Applications

Our framework is organized into a core analysis component and a data race detection component. The analysis component consists of stack unwinding, training, online and offline analysis, and runtime elision. The analysis component can be used out of the box by any other application. The data race detection component has two subcomponents: a) application-agnostic RDMA instrumentation and b) application-specific instrumentation. The RDMA instrumentation is done once per specific system API (e.g., DMAPP). The application specific instrumentation needs to be neither too coarse grained, nor too fine grained and needs retargeting for each application. It took us about three months to get the application-specific instrumentation working correctly in NWChem.

7. NWChem Application Insights

Figure 5 presents the dynamic call graph of a routine inside the Hartree-Fock solver which destroys an atomic task counter, copies the data from global memory to local memory, and destroys the global memory. As shown, the code causes execution of nine barriers (three are redundant) across five levels of software abstraction.

By examining barriers in conjunction with their dynamic behavior, we uncovered context-sensitive and context-insensitive redundant barriers. Most of the redundant barriers during NWChem execution are context sensitive. Figures 6 shows dynamic callgraphs from the `nxtask` and `pnga_add_patch` routines, where `pnga_destroy` is sometimes called in sequence based on the input arguments. Internally, `pnga_destroy` calls barriers at entry and exit. In this case the programmer adds an additional `pnga_sync` call that causes a redundant barrier when the previous calls to `pnga_destroy` already enforce barriers.

We also uncovered context-insensitive barriers, some new, some already known to the NWChem developers. As we continue to analyze traces we expect to find more of these situations. Figure 7 shows how inside `comex_malloc` and `comex_free` a barrier operation is dominated by another collective operation, in this case an `Allgather`. As probably this is highly likely to happen in a context-sensitive manner, we have started extending the analysis to perform elision based on knowledge about all collective operations in the program.

Figure 8 shows one of the top subcontexts found by our offline analysis. The redundant barrier is common to a group operation function (`do_gop`). 8% of redundant barriers (11186 out of 138072) happen in this subcontext. 7% (553 out of the total 7959) of the unique redundant barrier contexts have this suffix in their calling contexts. On investigation, we found that in the `do_gop` routine, redundant barriers are intentionally introduced for portability, but not needed in most production system software configurations. These occur to quiesce the caller runtime upon any transition into a callee runtime, e.g. when transitioning from ComEx into an explicit independent MPI call inside the application. Our analysis can be configured to keep these barriers when needed.

The key insight is that redundancy is determined by clustering of calls at several levels of abstraction removed from the actual operation. In the NWChem case, this spans different programming

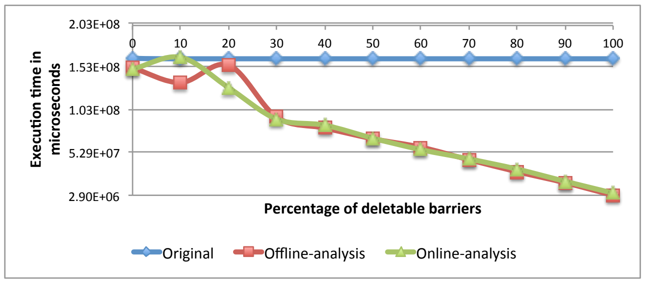


Figure 9. Impact of variation in fraction of deletable barriers on barrier elision. As more fraction of barriers become redundant, execution time reduces due to barrier elision.

languages and runtime implementations. Understanding the code by mere visual inspection is probably beyond most humans, and our analysis techniques provide useful insight to developers.

Most redundant barriers are context sensitive, but our sub-context based classification indicates that there are a few contexts that contribute to large fraction of the redundancy. In these cases a synchronized/non-synchronized dual implementation is feasible at the application level.

8. Performance Evaluation

We evaluate performance on a Cray XC30 MPP installed at the National Energy Research Scientific Computing Center (NERSC). Each of its 5200 nodes contains two 12-core Ivy Bridge processors running at 2.4 GHz. Each processor has four DDR3-1866 memory controllers which can sustain a stream bandwidth in excess of 50 GB/s. Every four nodes are connected to one Aries network interface chip. The Aries chips form a 3-rank dragonfly network. Note that depending on the placement within the system, traffic can traverse either electrical or optical links.

8.1 Microbenchmarks

In Figure 9, we present results from a microbenchmark written to asses the runtime overhead of our implementation. The code performs 500,000 barrier operations. We vary the degree of barrier redundancy from 0% to 100% in 1,000 different contexts. To indicate that a barrier is necessary, the micro benchmark calls a dummy function before it, which sets a flag in our instrumentation library. The performance is presented for a run using 9,600 MPI ranks and a stack depth of 16. The figure indicates that the analysis adds a negligible runtime overhead per barrier operation and it is able to improve performance as the degree of barrier redundancy increases.

Overall, the performance improvements are determined by the scalability of barrier operations and the scalability of the analysis. Barrier latency grows with the number of cores. For example, we observe 30μs and 130μs at 2400 and 19200 processors, respectively. Overall the analysis overhead is independent of the number of cores, but increases with the number of barrier contexts during execution and with the depth of the program stack. The offline analysis seems to have a slightly lower overhead than the online analysis. For example, the overhead per barrier operation varies from 4.7μs to roughly 8μs when increasing stack depth from 4 to 64 and providing thousands of contexts. The maximum stack depth for any barrier during the NWChem runs is 21.

8.2 NWChem

Figure 5 presents the end-to-end performance improvements observed for two aforementioned science production runs of NWChem. As shown, both methods are able to uncover a large number of redundant barriers, up to 45% and 63% for online and offline analyses respectively. This translates into application speedup up to 13.3% and 13.9% on 2048 cores, with online analysis and offline analysis respectively. Note that we report end-to-end speedup, which in NWChem includes a significant I/O portion.

Since the online algorithm learns behavior for a tunable number of repetitions of the same context, it may miss barriers that are redundant in a large number of independent contexts. The algorithm learns only once and it misses the cases where barriers become redundant later in the execution. To asses optimality, we compare the decisions of our online algorithm with the “true” barrier redundancy extracted using the data race information for an execution.

For the dichlorine oxide input, the code executes a total of 138,072 barriers in 7959 unique contexts. Only 45,614 barriers are required, according to our data race detection. The online algorithm learns for 31,750 barriers and elides 57,238 barriers in 2359 contexts, succeeding in deleting 41% of the redundant barriers. During the speculation phase 5238 more barriers become redundant during execution, but are not skipped by our implementation. Similar trends are observable for other inputs. Overall, these results indicate that our techniques are able to skip a significant fraction of the redundant barriers. Extending the algorithms to re-learn redundancy is easy and may further improve performance.

Other performance improvements may be possible by specializing the learning process to exploit module information and dynamically tailoring the stabilization threshold. For example, for our inputs we set 10 as the learning threshold, where as maximum threshold required was seven, and the vast majority of contexts stabilized by their 3^{rd} learning iteration.

Memory Overhead: A key contributor to the memory overhead is the size of the hash table used to memorize the contexts. In both DCO and OCT scientific runs, the size of the hash tables were only 456KB per process. This size is insignificant compared to tens of Giga bytes of scientific data resident per process. Our instrumentation added an additional 1MB binary code to an already 180MB NWChem’s statically-linked executable.

Higher Impact of Barrier Elision: At the time of writing this paper, we have identified a redundant initialization in NWChem for the dichlorine oxide input. This finding was based on insights from NWChem experts and further corroborated by a dead write detection tool [9]. Due to contention for the on-chip memory controllers, this redundancy impacts the execution time from 15% up to 50%, depending on the number of processes per node. Eliminating the redundant initialization reduces the overall running time. This optimization increases the percentage contribution of the communication in the overall execution. We have not been able to collect the new data with barrier elision with this modification due to time constraints and the cost of large-scale executions. We expect speedup due to our offline elision technique to increase from 13.9% to as high as 28%. Similarly, the online elision technique would show application speedup of up to 27%.

9. Discussion

The analyses exploit some inherent characteristics of the NWChem code base, which composes multiple iterative solvers written using SPMD parallelism. As the online method learns behavior, iterative algorithms present more optimization opportunity. Fortunately, this is the case with many scientific codes, which eschew direct solvers in favor of indirect iterative solvers. In order to make the overhead of classification palatable for the multi-pass approaches it is desired that behavior learned at low concurrency applies to high concurrency. This is the case for most existing SPMD codes. For the few scientific codes that have been tuned to switch solvers based on concurrency, training at the appropriate concurrency or writing synthetic tests for solvers is required. Overall, we believe that this

Table 5. End-to-end performance results for the dichlorine oxide (DCO) and oxidized cytosine (OCT) simulations.

Input	Number of cores	Time(s)	Total Barriers	Speedup / Barriers elided Offline	Speedup / Barriers elided Online	Unique Contexts
DCO	512	731	138072	0.7% / 63%	0.3% / 41.7%	7959
	1024	1084	138072	7.6% / 63%	0.2% / 41.4%	7959
	2048	1362	138072	13.9% / 63%	13.3% / 41.4%	7959
OCT	512	570	72188	3.4% / 63%	1.7% / 44.5%	4702
	1024	586	72188	6.6% / 63%	4.4% / 44.6%	4702
	2048	624	72188	6.0% / 63%	6.5% / 44.6%	4702

type of context sensitive dynamic analysis is applicable to many other scientific codes.

The combination of an ad-hoc lightweight data race detector with context-sensitive voting in synchronization operations (barriers) enables even more powerful synchronization optimizations. We are already considering extensions for reasoning about barriers in conjunction with other collectives, for transforming blocking collective calls into non-blocking calls, and for relaxing conservative communication synchronization operations such as fences. We believe these optimizations are useful for the Molecular Mechanics solvers in NWChem, whose scalability is limited by collective operations. Other code bases such as Cyclops Tensor Framework [34] will directly benefit from similar optimizations.

Mining the context information generated by our analyses already provided insight into the code characteristics, which can be used for manual transformations. We believe there is an opportunity to refine the notions of context (e.g. to clusters of variables) and to extend the classification methods in order to develop useful program understanding tools for large scale codes.

Online barrier elision requires a low analysis runtime overhead, which we achieved by using a simplified and conservative race detector. The multi-pass approach can use precise race detectors in the offline training, and thus has the potential of uncovering a larger number of redundant barrier contexts. Note that as the race detector generates seeds for offline classification using testing, it can be also replaced by some other search procedure based on choosing contexts without any race information.

Our approaches are not sound and use dynamic program analysis and speculation. They are guaranteed to catch bad speculation at runtime and abort execution, but this may be undesirable at large scale. Bad speculation has not happened in our productions runs and as any other program transformation tool, developer confidence in their applicability has to be gained through testing coverage.

10. Related Work

Concurrency analysis can be traced back to the work of Shasha and Snir [32]. Static program analysis that examines the synchronization in parallel programs has been applied for parallelization purposes as well as performance optimizations. Jeremiassen and Eggers [19] present a static barrier analysis for SPMD codes (Stanford SPLASH) used to eliminate false sharing on shared memory machines. Zhang et al. [38, 39] present concurrency analyses for shared (OpenMP) and distributed (MPI) programming models with textually unaligned barriers. Kamil et al. [20] describe concurrency analyses for a programming language textually aligned barriers. They use the analysis for static data race detection in Titanium. Agarwal et al. [2] present a may-happen-in-parallel analysis for X10 programs with unstructured parallelism.

Runtime elision of synchronization operations has received a fair share of attention in both software and hardware. Many techniques have been developed for lock elision in Java, `glibc`, or the Linux kernel. At the hardware level, speculative lock elision has been described by Rajwar et al. [30] and it is available now in hardware implementations such as Intel TSX.

Sharma et al. [31] describe a dynamic program analysis to detect functionally irrelevant barriers in MPI programs. In their definition, a barrier is irrelevant if its removal does not alter the overall MPI communication structure of the program. They use a model checker to try program interleavings. Their technique uncovers irrelevant barriers in many small benchmarks containing a few hundred lines of code. Our notion of redundancy takes into account data races and we automatically elide the redundant operations.

Lightweight instrumentation of parallel programs is a well explored areas. Library interposing and link-time function wrapping are standard techniques to intercept function calls. These techniques are extensively used in performance analysis tools such as HPCTOOLKIT [36]. Our on-demand call stack unwinding technique on each barrier bears similarity with HPCTOOLKIT's call stack sampling. HPCTOOLKIT maintains an unwind recipe for each range of instructions by analyzing the binary code at runtime. Our technique deviates from this since we compile the code with frame pointers to ensure perfect stack unwinds. Binary analysis is directly applicable to improve our technique. An alternative context collection technique is to instrument every call and return instruction and eagerly compute the call path [8]. Eager call path collection techniques, while suitable for frequent unwinds, are unsuitable when the call path is needed infrequently. Context-sensitive dynamic analyses have been explored in the computer security also. Most of the approaches use stack walking for context identification. Recent work by Bond and McKinley [5, 6] refines the notion of a context to avoid stack unwinding overhead. Their probabilistic calling contexts are directly usable in our analyses.

11. Conclusions and Future Work

In this paper we present context-sensitive dynamic program analyses able to detect and eliminate redundant barrier operations in the NWChem computational chemistry code. By applying techniques of learning and speculation we are able to eliminate up to 63% of the barriers executed during science production runs. This translated into end-to-end speedup as high as 14% at 2048 cores.

We believe that our paper describes both a problem and solutions common to many other scientific codes. NWChem combines multiple programming languages (Fortran, C, C++) and runtimes (MPI, ComEx, GA) into a modular code base that provides solvers, memory management, tasking and load balancing, as well as fault tolerance mechanisms. NWChem uses modern programming concepts such as PGAS and RDMA and already contains internal optimizations to eliminate redundant synchronization.

In the NWChem case, modular software engineering introduced a very high number of context-sensitive redundant barrier operations. The magnitude was surprising to everybody, including NWChem experts.

As scientific codes are evolving towards multiphysics multiscale simulations and increasingly start using PGAS or one-sided communication, it is likely that redundant synchronization will become more prevalent. Our idea of running an online data race detector to detect redundant operations can be easily incorporated into performance analysis tools to provide application tuning insight.

Analyses similar to ours, that examine applications alongside runtimes, can provide further software understanding support, as well as automated optimization tools.

Acknowledgments

Support for this work was provided in part through the X-Stack program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under collaborative agreement numbers DE-SC0008699 and DE-SC0010473.

References

[1] Cray Unified Parallel C. http://docs.cray.com/books/S-2179-50/html-S-2179-50/z1035483822pvl.html.

[2] S. Agarwal et al. May-happen-in-parallel analysis of X10 programs. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '07, 2007.

[3] C. Barton et al. Shared memory programming for large scale machines. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, 2006.

[4] L. S. Blackford et al. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, 1997.

[5] M. D. Bond et al. Efficient, context-sensitive detection of real-world semantic attacks. In *Proc. of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, 2010.

[6] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proc. of the 22Nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications*, OOPSLA '07, 2007.

[7] V. Cavé et al. Habanero-java: The new adventures of old X10. In *Proc. of the 9th Intl. Conf. on Principles and Practice of Programming in Java*, PPPJ '11, 2011.

[8] M. Chabbi, X. Liu, and J. Mellor-Crummey. Call paths for pin tools. In *Proc. of Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization*, CGO '14, pages 76:76–76:86, 2014.

[9] M. Chabbi and J. Mellor-Crummey. DeadSpy: a tool to pinpoint program inefficiencies. In *Proc. of the 10th Intl. Symp. on Code Generation and Optimization*, CGO '12, pages 124–134, 2012.

[10] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for GPU-accelerated supercomputers. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 43:1–43:12, 2013.

[11] B. Chamberlain et al. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3), Aug. 2007.

[12] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10), Oct. 2005.

[13] ComEx: Communications Runtime for Exascale. http://hpc.pnl.gov/comex/.

[14] A. Danalis. MPI and compiler technology: A love-hate relationship. In *Proc. of the 19th European Conf. on Recent Advances in the Message Passing Interface*, EuroMPI'12, 2012.

[15] P. C. Diniz and M. C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.*, 49, 1998.

[16] M. A. Heroux et al. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31, 2005.

[17] Compiled MPI. http://htor.inf.ethz.ch/research/compi/.

[18] P. Husbands et al. A performance analysis of the Berkeley UPC compiler. In *Proc. of the 17th Annual Intl. Conf. on Supercomputing*, ICS '03, 2003.

[19] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proc. of the IFIP WG10.3 Working Conf. on Parallel Architectures and Compilation Techniques*, PACT '94, 1994.

[20] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *In Proc. of the 18th Intl. Workshop on Languages and Compilers for Parallel Computing*, 2005.

[21] A. Karwande et al. CC-MPI: A compiled communication capable MPI prototype for ethernet switched clusters. In *Proc. of the Ninth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '03, 2003.

[22] P.-W. Lai et al. A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[23] The libunwind project. http://www.nongnu.org/libunwind/.

[24] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, 2002.

[25] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy libray for distributed array libraries and compiler run-time systems. In *Proc. of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th Intl. Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing*, 1999.

[26] J. Nieplocha et al. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2), May 2006.

[27] C. S. Park et al. Scaling data race detection for partitioned global address space programs. In *Proc. of the 27th Intl. ACM Conf. on Intl. Conf. on Supercomputing*, ICS '13, 2013.

[28] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[29] R. Preissl et al. Transforming MPI source code based on communication patterns. *Future Gener. Comput. Syst.*, 2010.

[30] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of the 34th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO 34, pages 294–305, 2001.

[31] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. In A. Lastovetsky et al., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 265–273. Springer Berlin Heidelberg, 2008.

[32] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2), 1988.

[33] J. G. Siek et al. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[34] E. Solomonik et al. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th Intl. Symp.*, pages 813–824. IEEE, 2013.

[35] Sparsehash. https://code.google.com/p/sparsehash/.

[36] N. R. Tallent et al. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '09, pages 441–452, 2009.

[37] M. Valiev et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[38] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *PPOPP*, 2007.

[39] Y. Zhang et al. Concurrency analysis for shared memory programs with textually unaligned barriers. In *LCPC*, pages 95–109, 2007.