Code-Centric Communication Graphs for Shared-Memory Multithreaded Programs

Benjamin P. Wood Joseph Devietti Luis Ceze Dan Grossman
University of Washington
{bpw,devietti,luisceze,djg}@cs.washington.edu

Abstract

With more and more complex pieces of software using explicit multithreading, tools to extract the structure of such software are increasingly important. We present a novel tool that builds graphs describing how threads in shared-memory parallel programs communicate. Compared to prior work, our communication graphs are *code-centric*: nodes represent units of code (*e.g.*, functions) and edges represent inter-thread shared-memory communication via these units of code. Our approach is dynamic, using actual executions to build graphs, and exploits binary-code instrumentation to work for large, real-world applications. The graphs are useful for understanding software structure and computing program properties, such as the effect of nondeterministic thread-scheduling on the communication pattern.

1. Introduction

Multithreaded desktop and server applications are becoming unavoidable in order to reap the performance benefits of multicore architectures. Unfortunately, writing, debugging, and understanding multithreaded code is much more difficult than for single-threaded code. Automatic and interactive development tools can ameliorate this pain. While many tools exist and are commonly used in industry (*e.g.*, Intel's ThreadChecker [1]), new tools that complement them — such as the one in this work — are sorely needed.

While there are many models for parallel and concurrent programming, we focus on shared-memory programs because this is the prevailing paradigm for multicore machines. We begin by reviewing the advantages and disadvantages of shared memory. We then consider how existing tools for helping with shared-memory computing take a data-centric approach rather than our code-centric approach.

Shared-Memory Benefits and Drawbacks Shared memory makes inter-thread communication *simple*: One thread writes to a shared location that another thread later reads. On multicore architectures it is also *efficient*: We can communicate an entire data structure by communicating just a pointer to it. The hardware moves parts of shared data structures to the processors using them. Compared to message-passing, there is less focus on the complexities of serializing objects and developing explicit communication protocols.

The rub is that shared memory is *too easy*: Typically communication is implicit; any memory access might or might not be part of inter-thread communication. Understanding the structure an application requires inferring its pattern of inter-thread communication, which is exactly what shared memory does not make manifest. Ideally we would like to recover the communication patterns that message passing makes explicit without sacrificing convenience or performance.

Although shared-memory *allows* inter-thread communication almost everywhere, well-written applications actually communicate among threads rarely in the code. Therefore, tools that extract communication patterns from shared-memory programs can reveal program structure that is invaluable for debugging, software understanding, testing, etc.

The Data-Centric Approach Much research over the last decade has tamed shared memory by determining what *memory locations* are shared among threads and what *synchronization idioms*, particularly locks protecting memory locations, are used. Static and dynamic analyses have been developed to identify and enforce "good" programming patterns such as thread-local data and data consistently guarded by a lock. More recently this line of work has been extended to identify atomicity violations, i.e., when some piece of code does not appear to execute all at once to other threads.

Such work takes a *data-centric* view of shared-memory communication. For each memory location, it determines how it is used. For example, it might be accessed only while holding a specific lock. Even atomicity detectors work by considering what data is accessed in some critical section and then whether the invariants for the data are such that the critical section is in fact atomic.

Our Code-Centric Approach Our work takes a fundamentally complementary view of an application's communication structure. Instead of presenting information in terms of memory locations, we present only which units of code (*e.g.*, functions) participate in inter-thread communication with which other units of code.

Our approach produces a *communication graph* in which the nodes are code units and the edges indicate that during program execution the edge's source wrote data to one or more locations and the edge's target then read that data in another thread. (Section 2.2 modifies this basic definition, but the core idea remains this simple.) In this work, we compute such graphs via a dynamic analysis. Our focus is on the graphs and their usefulness. We have built graphs for the Parsec [4] multithreaded benchmark suite and for large widely used parallel applications including MySQL and the Apache web server. Our tool uses dynamic binary instrumentation to support any program using C or C++ with POSIX Threads without needing the source code.

Qualitatively, the generated graphs are valuable documentation, especially after simple and principled manual pruning of a few uninteresting nodes that have many edges. Anecdotally, for every program we have considered, we can learn key aspects of the program's structure from its graph even before we have ever looked at the source code. More quantitatively, the graphs have enabled several experiments and analyses that characterize large multithreaded applications, including:

- The distribution of node-degrees in a graph indicates how centralized or diffuse shared-memory communication is in the code base.
- By comparing graphs for program runs with the same inputs, we can measure the effect of nondeterministic thread-scheduling on communication patterns.
- By comparing graphs for program runs with different inputs, we can measure the effect of inputs on communication patterns, which helps assess test coverage.

Contributions and Outline We introduce communication graphs as a new, code-centric way to describe the structure of shared-memory programs. We describe a dynamic tool to build the graphs automatically by observing an instrumented program execution. Our tool works for large and sophisticated programs. We consider important variations of the graph, particularly a notion of function "communication inlining" that lets us produce different graphs that correspond to different levels of software abstraction. The graphs enable us to compute relevant communication metrics for multithreaded programs.

Section 2 defines several variations of communication graphs. Section 3 describes how to dynamically generate communication graphs as well as key optimizations to reduce instrumentation overhead. Section 4 presents salient details of our dynamic-analysis tool. Section 5 presents our experimental evaluation, including case studies and summary statistics across applications. Sections 6, 7, and 8 discuss related work, future work, and conclusions, respectively.

2. Communication Graphs

This section describes several flavors of communication graphs and how to generate them. The discussion assumes

```
void enqueue(Queue* q, int val) {
  ... // synchronization of q
  q->buf[q->tail++ % BUF_SZ] = val;
  ... // synchronization of q
}
int dequeue(Queue* q) {
  ... // synchronization of q
  int ans = q->buf[q->head++ % BUF_SZ];
  ... // synchronization of q
  return ans;
}
Queue pipeA = INITIAL_QUEUE;
Queue pipeB = INITIAL_QUEUE;
void stage1() { /* put items in pipeA */ }
void stage2() { /* remove items from pipeA;
                   put items in pipeB */ }
void stage3() { /* remove items from pipeB */ }
```

Thread 1:	Thread 2:	Thread 3:
`stage1();`	`stage2();`	`stage3();`

Figure 1: Outline of a simple pipeline program.

nodes in the graphs are functions in the code. Other granularities, either finer such as individual lines or coarser such as entire files, also work well, which often helps to distinguish inter-file edges from intra-file edges.

2.1 Basic Graphs

Communication graphs are directed graphs where the nodes are functions in the source code. When a thread T_1 executing a function f_1 reads a memory location, the last write to the location must have been performed by some thread T_2 executing some function f_2. If $T_1 \neq T_2$, then a directed edge from f_2 to f_1 is included in the graph. (Note $f_1 = f_2$ is possible.) The graph for a program execution includes all such edges. They describe exactly the pairs of functions that participated in inter-thread communication.

To build the graph for a dynamic execution, we interpose instrumentation code on all memory reads and writes. On a write to memory address m, we record the executing thread T and executing function f as the most-recent write to m in an in-memory table that maps memory addresses to most-recent writes. On a read of m, we consult the table and add an edge to an in-memory representation of the graph unless the edge is already present. The edge is not added for intra-thread communication (i.e., the thread in the table is the thread doing the read), which is the common case. When the program terminates, the graph is written to disk. Subsequently, a graph-visualization tool we wrote using the Prefuse visualization toolkit [7] is used to view the graph. The tool lets users move or remove graph nodes interactively.

As an example, consider the program outlined in Figure 1 (the `Queue` type definition and corresponding synchronization is elided). This program implements a simple pipeline, where a separate thread performs each pipeline stage, communicat-

```
Queue buf = INITIAL_QUEUE;
void produce() { /* put items in buf */ }
void print() { /* remove items from buf
                  and print them */ }
void discard() { /* remove items from buf
                    silently */ }
```

Thread 1:	Thread 2:	Thread 3:
`produce();`	`print();`	`discard();`

Figure 2: A producers-consumers program with a high-level communication pattern different than Figure 1's program.

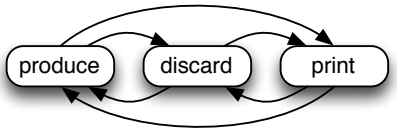


Figure 3: Communication graph for the program in Figure 2, with `enqueue()` and `dequeue()` "inlined."

ing with adjacent stages via synchronized bounded-buffer queues. Figure 4a shows the trivial communication graph recorded for an execution of this program.

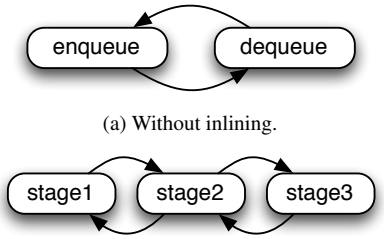
Functions that do not participate in inter-thread communication do not appear in the graph. For example, the functions `stage1`, `stage2`, and `stage3`, while arguably involved in communication, did not directly perform any read or write operations that caused communication. In a full application, most functions are clearly not involved in inter-thread communication and so do not appear in the graph.

2.2 Function "Communication Inlining"

A useful way to extend the basic communication graph just describes is to ascribe communication to a function's caller rather than to the function actually doing the read or write. For example, reconsider the pipeline program in Figure 1. If we "inline" the calls to `enqueue()` and `dequeue()`, then we get the graph shown in Figure 4b.

Neither the graph in Figure 4a nor the graph in Figure 4b subsumes the other in usefulness. They present complementary information by representing inter-thread communication at different levels of abstraction. The lower-level graph in Figure 4a is useful for ensuring all communication occurs through shared queues, as evidenced by the absence of other edges. The higher-level graph in Figure 4b displays the overall pipeline in the application. Indeed, if we consider the program shown in Figure 2, it has exactly the same lower-level graph as Figure 4a, but inlining the calls to `dequeue()` and `enqueue()` leads to the graph shown in Figure 3.

Therefore, we leave choosing what functions to inline to developers. The choice will depend on the level of abstraction at which they wish to view the program. Given a list of inlined functions, the tool is modified as follows: The instrumentation for a read or write checks if the executing function f is on



(a) Without inlining.

(b) With `enqueue()` and `dequeue()` "inlined."

Figure 4: Communication graph for the program in Figure 1.

the list. If so, the instrumentation code inspects the stack to determine f's caller, and then uses it as the node in the graph. Repeated inlining is no problem; the instrumentation continues inspecting the stack to find the shallowest non-inlined function.

Inlining affects graph generation, so the program must be re-run with different lists of inlined functions. Therefore, producing the graphs in Figures 4a and 4b requires separate program runs. The alternative is to record the whole call stack for each memory access, which we deem too expensive.

2.3 Usage Scenarios

Developers can use communication graphs to investigate communication properties. High-degree nodes are producers or consumers of data for many other functions; they are often key to understanding the application's structure. More generally, a graph often visually decomposes into loosely coupled pieces that developers can peruse separately. The graphs could also be used to detect bugs if anomalous nodes or edges are discovered. While it is difficult to convey the invaluable experience of interactively exploring a communication graph, Section 5 describes case studies we performed.

Combining and comparing different graphs is also valuable. Different program runs can yield different graphs due to different inputs or nondeterministic thread-scheduling. The graph-union of graphs across multiple runs of an entire test suite describes the communication covered by the suite, while the graph-intersection describes the communication so core to the application that it is independent of the input. Edges present in the union but not in the intersection (*i.e.*, the symmetric difference) describe communication that occurs during only some executions. Section 5 quantitatively characterizes these properties.

3. Graph-Building Optimizations

Instrumenting every read and write and maintaining a table indexed by memory address is expensive, potentially slowing down execution by two orders of magnitude or more. While significant slowdown is acceptable for a code-understanding tool (consider how long it would take to derive an application's communication graph by hand), some care is warranted. In particular, real applications often have built-in timeouts to respond to slow connections and communications. We use two simple optimizations to achieve slowdown small

enough to avoid such problems. This section describes these optimizations and their effect on graph generation, i.e., what edges are potentially missed as a result.

Ignore Stack Accesses Our tool has an option to not instrument memory accesses that are on the stack. For a binary-instrumention tool, this means skipping accesses indexed from the stack pointer or frame pointer registers. While it is possible in C/C++ to use pointers to local variables for inter-thread communication, it is rare. Where it does occur, it is often for simple fork-join idioms that are easily identified in the source code. Nonetheless, removing instrumentation of stack accesses can omit communication edges. Though it is optional to ignore stack accesses, in practice we do so.

Permit Data Races Our run-time system lets multiple threads access distinct memory-table entries in parallel. A completely accurate graph would still require synchronizing on each access to a memory location so that the table update and actual memory access occur atomically. However, removing this synchronization has only a small effect on graph correctness — and none for data-race free applications.

We justify this claim via the example in Figure 5. Thread 3 reads `a` after it was written by either Thread 1 or Thread 2. If the former, as in the interleaving shown in the figure, then Thread 3 also reads `b` written by Thread 1 so the communication graph should have the single edge from `f` to `h`. By allowing a race with Thread 2 between the instrumentation of Thread 1's write to `a` and Thread 1's actual write to `a`, the instrumentation records that Thread 2 last wrote to `a`. So the graph produced has edges from `f` to `h` and `g` to `h`.

Such a graph is false: there is no execution where `f` and `g` both communicate to `h`. Is it worth risking an impossible graph in order to avoid the performance cost of synchronizing on every memory access? Two reasons justify an affirmative answer. First, impossible graphs can arise only from data races in the original application, which are rare, worth finding with data-race detection tools (not our focus), and forbidden by the emerging C++ standard [5]. Second, while the *overall graph* is impossible, *every edge* in the graph is possible on some execution. That is, every edge is possible on some interleaving. This argument holds generally: no race can lead to an impossible edge, only impossible graphs.

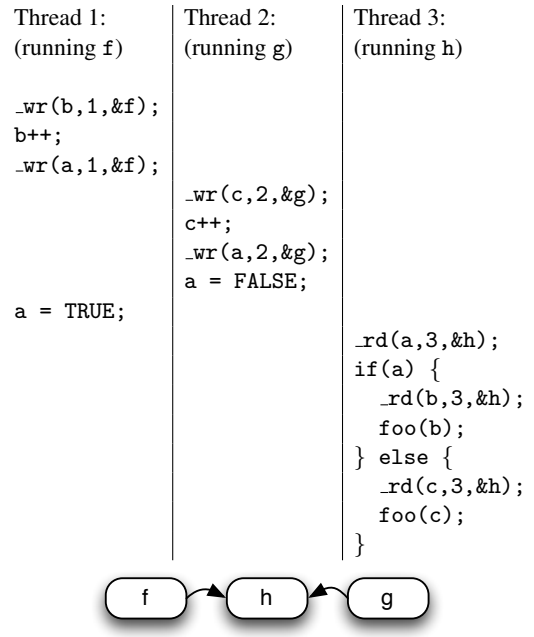


Figure 5: An execution to demonstrate that data races can lead to an impossible communication graph but not impossible edges. Time progresses downward. Functions `_wr` and `_rd` are the instrumentation for writes and reads.

4. Tool Details

Our tool consists of a runtime monitor to collect communication graphs and a visualizer to examine them.

4.1 Runtime Monitor

The monitor handles arbitrary C/C++ binaries and is implemented in the Pin binary instrumentation framework [10]. Using an industrial-strength binary-instrumentation framework lets us analyze large real-world applications with no modification of the original source code. After the tool was developed and debugged, it required zero modifications to handle large applications such as MySQL and Apache.

The runtime monitor maintains a memory map that records the last thread and function to write to each memory location, as discussed in Section 2.1. The tool instruments memory accesses with hooks to update the map and the communication graph, stored as an adjacency matrix.

Inlining To handle function inlining, we could mark each call frame on the stack as inlined or non-inlined and walk the stack at each meory access to attribute the operation to the most recent non-inlined function. To improve performance, the monitor maintains a shadow stack instead. When a non-inlined function is called, we push its identifier onto the shadow stack and pop it when the function returns. Calls and returns of inlined functions are ignored. Load and store hooks use the top of the shadow stack as the current function. Exceptions could cause shadow stack misalignments and result in graph errors, but we have not observed this problem in programs we tested.

The monitor supports flexible configuration of inlining at the binary image, source file, and function granularities. The tool instruments all memory accesses, but by default it inlines all functions not in the specified binary image.

Memory Allocators Memory allocators change the meaning of a memory address by reusing it for logically different pieces of data. In well-behaved programs, the data stored in a piece of memory has reached the end of its meaningful life when that memory is freed. To distinguish the data stored at a given memory address that is re-allocated, the tool erases the memory map entries for a chunk of memory when

Benchmark	Total LOC	Instrumented Functions	Initial Graph		After Inlining			After Inlining and Pruning		
			Functions	Edges	Inlined	Functions	Edges	Pruned	Functions	Edges
blackscholes	421	13	4 (31%)	8	-	-	-	-	-	-
bodytrack	11,804	233	82 (35%)	296	14	77 (33%)	336	9	68 (29%)	279
canneal	4,085	42	10 (24%)	23	-	-	-	-	-	-
dedup	3,683	122	39 (32%)	143	14	28 (23%)	131	5	23 (19%)	95
facesim	29,355	1454	77 (5%)	259	22	59 (4%)	231	1	58 (4%)	226
ferret	15,035	1861	79 (4%)	181	19	77 (4%)	153	43	34 (2%)	60
fluidanimate	941	27	12 (44%)	45	-	-	-	-	-	-
freqmine	2296	74	34 (46%)	140	-	-	-	-	-	-
raytrace	12,878	5589	19 (<1%)	35	4	6 (<1%)	7	-	-	-
streamcluster	2,333	37	14 (38%)	60	-	-	-	-	-	-
swaptions	1,165	39	14 (36%)	63	-	-	-	-	-	-
vips	174,151	5064	135 (3%)	722	22	126 (2%)	778	26	100 (2%)	548
x264	37,413	408	140 (34%)	538	5	135 (33%)	524	12	123 (30%)	435
httpd	335,914	2072	401 (19%)	1269	1108	258 (12%)	1146	37	221 (11%)	876
mysqld	941,021	11215	847 (8%)	2293	575	811 (7%)	2254	388	423 (4%)	802

Table 1: Graph metrics for Parsec, `httpd`, and `mysqld`. At each stage of graph processing, "Functions" denotes the number (and percentage) of functions that communicate.

it is allocated by any of the standard C allocator functions (`malloc`, `calloc`, etc.), effectively giving each allocation a fresh logical address. Furthermore, the tool ignores all memory accesses inside the malloc/free library.

Performance Our runtime monitor induces application slowdowns of 200x to 450x; overheads are 25-50% higher without our optimization of ignoring stack accesses. Though inconvenient, these overheads have not proved prohibitive in our tests. Subsequent tools targeting the Java language have total overheads 1-2 orders of magnitude lower, but are beyond the scope of this paper.

4.2 Visualizer

The visualizer, based on the Prefuse visualization toolkit [7], lets users interactively zoom, rearrange, and filter nodes from the communication graph. The graph is layed out via an n-body physics simulation wherein nodes repulse one another and edges are springs. In practice this provides a helpful visual layout, moving communicating nodes closer together, and non-communicating nodes farther apart. The user can also move nodes around manually with or without the physics in action.

5. Evaluation

We evaluated our tool on the Parsec multithreaded benchmark suite [4], the Apache web server, and the MySQL database server. After describing the applications, we present case studies of the use of our tool on two applications, followed by observations on the principled use of inlining and pruning. Finally, we summarize all the applications with quantitative metrics computed over their communication graphs, analyzing graph complexity and graph stability across multiple program runs.

5.1 Applications Investigated

The Parsec benchmark suite encompasses a wide variety of parallel programming models, including pipelines, fork/join concurrency, task queues, and work-stealing. The workloads themselves are also diverse, including such tasks as file compression, ray-tracing, data mining and physics simulation.

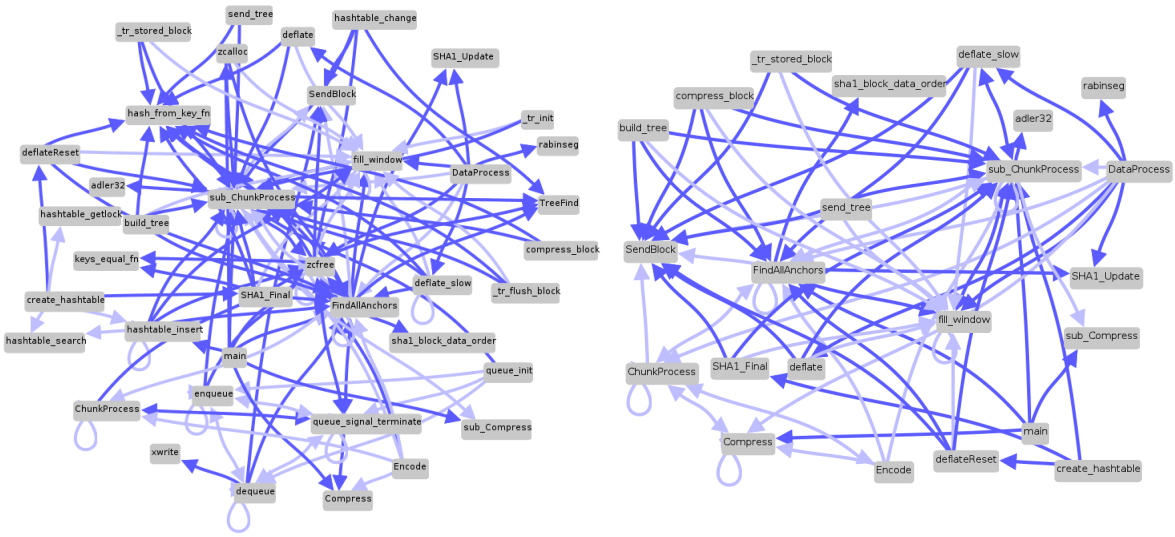
Apache is a widely-deployed industrial-strength web server. We configured version 2.2.11 to use worker threads and used the `httperf` web workload generator to run several clients, simultaneously requesting documents from a single `httpd` server that was running under our runtime monitor.

The popular MySQL database server allows concurrent access by large numbers of clients. We configured version 5.1.32 with defaults suggested for small installations. To exercise inter-thread communication in the `mysqld` server, we used the accompanying `mysqlslap` utility to execute workloads with several clients simultaneously querying a single server.

Table 1 shows information about the applications. The second column shows their size in lines of code. The third column shows how many functions were not implicitly inlined (*i.e.*, functions in the application, not in shared libraries).

5.2 Case Study: `dedup`

`dedup` is a compression program that pushes data through a 5-stage pipeline. Each stage uses multiple threads for load-balancing and higher throughput. The initial communication graph (Figure 6a) does not immediately reveal this pipeline structure. Communication often occurs via synchronization functions (*e.g.*, `queue_signal_terminate`), utility functions (*e.g.*, `hash_from_key_fn`), and custom-memory-allocater functions (*e.g.*, `zcfree`). To reveal `dedup`'s high-level structure, we inlined three simple kinds of functions:



(a) Before inlining and pruning

(b) After inlining and pruning

Figure 6: Communication graphs for `dedup`. Dark edges indicate inter-file, and light edges intra-file, communication. Only communicating functions (nodes with degree greater than zero) are shown.

(1) synchronization functions used by the shared queues separating the pipeline stages, (2) functions used to access a shared hash table, and (3) functions found in the source file `util.c`. We also pruned `dedup`'s custom memory-allocation functions. In total, we inlined 14 functions and pruned 5.

Inlining and pruning make `dedup`'s high-level structure much more evident (Figure 6b). The most-highly connected nodes — `DataProcess`, `FindAllAnchors`, `ChunkProcess` (and its helper function `sub_ChunkProcess`), `Compress` and `SendBlock` — are precisely the stages in the pipeline. Pipeline structure is evident from the edge directions: `DataProcess`, the first pipeline stage, is only a source of edges while `SendBlock`, the last stage, is only a sink. The darker edges show communication across file boundaries, while the lighter edges show intra-file communication. The pipeline stages clearly reside in the same file, but use helper functions in other files.

5.3 Case Study: MySQL Server

At 941,021 lines of code, `mysqld` is a large application. As a higher-level shared memory system itself, allowing concurrent database access by multiple clients, it contains significant inter-thread communication. Although only 8% of functions in `mysqld` communicate (see Table 1), the raw communication graph for any run of `mysqld` remains large, with almost 850 communicating functions and 2300 edges in larger graphs. Simply put, the unmodified graph is visually overwhelming. However, straightforward inlining and conservative pruning of the graph makes it quite tractable. Since we were unfamiliar with the code, we used the graph to explore the code, starting with the highest degree nodes. The very highest degree nodes included initialization functions such as `mysql_init_variables`, run at server boot time, `my_thread_init`, used to initialize thread data structures, and `THD::THD`, a thread metadata constructor. Pruning these nodes interactively revealed more structure in the graph.

Function pruning can remove false communication from the graph. Functions such as custom memory allocators, constructors, and destructors can obfuscate otherwise simple communication graphs by introducing false edges via global metadata. Communication through metadata channels such as the run queue of a thread pool or the free list of a custom allocator are not meaningful at the application level. Inlining custom allocators would induce communication between a function that frees an object and a subsequent function that allocates one. Instead, we eliminate this false communication path entirely by pruning the relevant functions.

In the `mysqld` graph, other high-degree nodes with suspicious function names like `my_vsnprintf` and `my_hash_insert` led us to source code for generic data structures and utility functions, which the source distribution fortuitously groups in two directories. Running `mysqld` again with all these utility functions inlined yielded a greatly simplified graph. We also pruned several custom memory allocator functions and all C++ object destructors. The resulting graph is much more manageable; simple inlining and pruning abstracts enough low-level communication to reveal communication patterns in `mysqld`. For example, functions for opening and closing tables, such as `open_table` and `close_open_tables`, are tightly connected, as shown on the right in Figure 7. Similarly, database and table locking operations are tightly connected on the left of Figure 7. A screenshot does not do the visualizer justice; we have found interactive graph exploration far more helpful than any static graph representation.

5.4 Function Inlining and Pruning

The simple heuristics for inlining and pruning presented above – inlining synchronization, data structure, and helper functions, and pruning custom allocators and initialization and destruction functions – tamed the complexity of many graphs by abstracting low-level communication.

Quantitative results, shown in Table 1 as the number of communicating functions (*i.e.* nodes with nonzero degree) and edges in a graph before and after applying inlining and pruning, support our qualitative observations of the usefulness of inlining and pruning. For large applications in our tests, it was not uncommon for inlining and pruning to remove 20-50% of communicating functions and edges. (Graph metrics are discussed further in Section 5.5.)

Inlining and pruning are relatively easy, even for those without prior knowledge of an application's source code. Since we were unfamiliar with the applications we targeted, we were conservative in our approach to inlining and pruning. However, we believe that developers working with familiar code will have good intuitions for what functions to inline or prune, enabling them to inline more application-specific functions that we did not.

In our experiments, candidates for inlining were often self-evident from their distinctive names, and graph topology highlighted high-degree nodes to prune, suggesting that the graphs are indeed powerful tools for program understanding. After identifying candidates in the graphs, we examined their source code to verify they should be inlined. Since function inlining can reveal new communication edges (as shown previously in Figure 4a), it was sometimes helpful to iterate on inlining a small number of times.

Finally, our experience with Apache and MySQL suggests that inlining and pruning allow our communication graphs to scale well to large applications. Furthermore, source-code organization in large applications often makes inlining and pruning simple, so the effort to use these techniques also scales well with application size. For example, the Apache Runtime library (APR) provides several low-level operations. Inlining just the APR and pruning functions for handling memory pools and allocation simplified the graph enough to show higher level communication between server components.

5.5 Graph Characterization

Representing inter-thread communication as a graph lets us quantify communication properties. We now show a quantitative analysis of the graphs from applications we studied as case studies for our tool. In summary, we find:

1. Node degree distributions are heavily biased towards zero.
2. At most 10% of nodes have degree above 10 (<7% for most applications).
3. Fewer than half of functions communicate at all.
4. In many programs, communication structure is relatively stable over multiple runs on a single input.
5. Across different inputs, communication differs by <15% for most applications, but reaches 50% in some cases.

Node Degree One intuitive way of understanding the complexity of a program's communication graph is by examining the degree distribution of the graph's nodes. The more functions that a given function communicates with, the higher the degree of its node in our graphs. Graphs whose nodes have lower degree generally indicate programs that perform inter-thread communication via a more limited interface.

Figure 8 shows the cumulative distribution function of node degrees in graphs that are the union of unpruned graphs observed across several runs of each benchmark. In all applications, nearly 100% of functions have degree less than 25; 90% have degree less than 8 for most applications. Fewer than half of all functions communicate at all, which supports the view that inter-thread communication is restricted to a subset of an application's functions. For large applications, this percentage is even lower, *e.g.* under 10% of the functions in `mysqld` participate in inter-thread communication. Small programs are more highly connected, but larger programs are well-partitioned and limit inter-thread communication.

Functions are often the right granularity for considering inter-thread communication, but communication between modules, classes, or source files is a good measure of the encapsulation of inter-thread communication among larger code units. Our experiments (omitted for space reasons) show that file degree in C programs behaves similarly to function degree, suggesting that applications limit inter-thread communication at multiple architectural layers.

Graph Stability We examined the stability of our communication graphs with respect to (1) changes in instruction interleaving due to nondeterminism across multiple runs of a program with the same input, and (2) changes due to running the program with a different input. For (1), the left part of Table 2 shows how many new edges, *i.e.* new communication paths, were discovered by monitoring an additional run of the program. For each benchmark, we chose the input that yielded the most new edges across runs; this differs from Table 1, which shows data aggregated across all inputs. The table gives the number of edges discovered that had not been seen in any previous run of the program. In general, only a modest number of runs, *e.g.* five runs total, is necessary to reach reasonable edge coverage for a single input. Across inputs (shown on the right part of Table 2), the edge variability can be much higher, so a diverse test suite is required to exercise all the communication in an application.

6. Related Work

Our work is most closely related to characterizations of shared-memory parallel programs, software-visualization tools, and tools for finding concurrency errors.

Characterizations of shared-memory parallel program benchmarks (*e.g.*, SPLASH [17] and Parsec [4]) normally

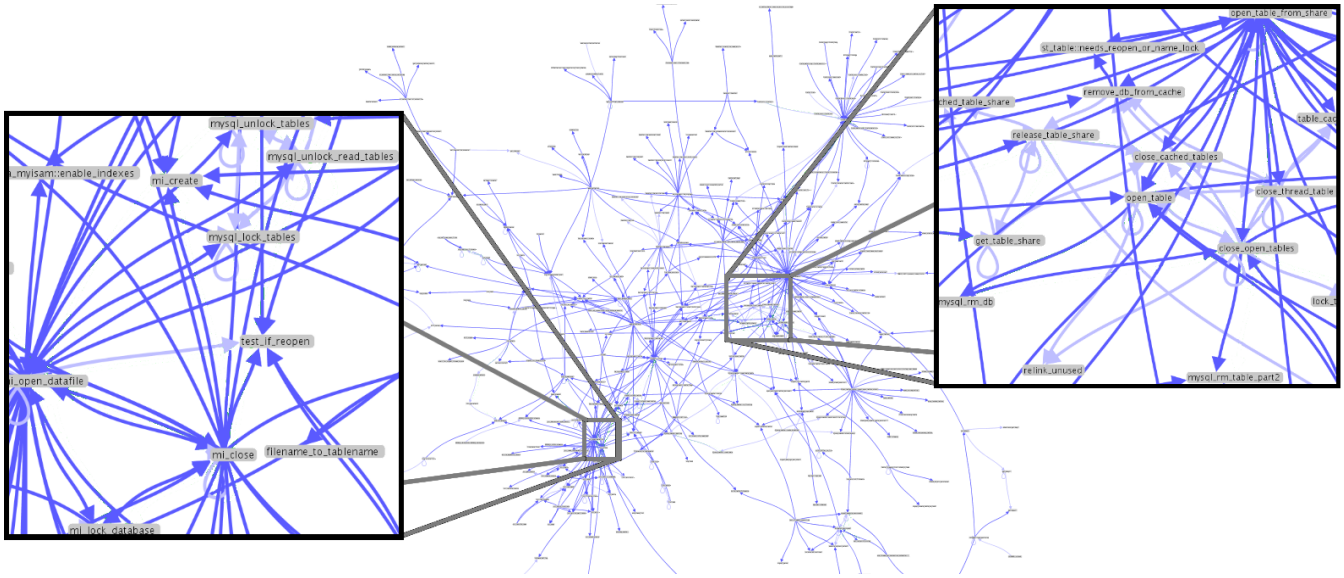


Figure 7: The main connected component in a communication graph for `mysqld` after inlining and pruning.

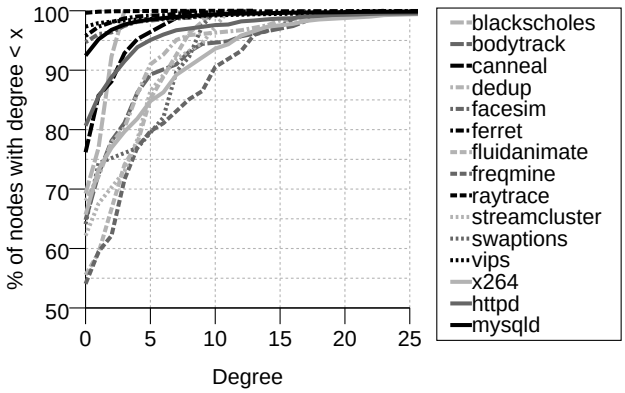


Figure 8: Cumulative distribution functions of node degree before inlining or pruning. Note the y-axis starts at 50%.

focus on the amount of shared data and how frequently memory accesses lead to a remote action (i.e., coherence protocol message). They provide a data-centric, low-level view of inter-processor communication. This is important when assessing behavior at the hardware level but does not describe the application's high-level communication structure.

Software visualization is a very mature and active research topic. Researchers have explored visualization of many aspects of programs. These include execution steps [15], data-structure relationship [18], inter-class communication via method calls [3, 12], profiling data visualization [2], and parallel program executions [6, 11, 13], which is the closest to our work. Past work on parallel-program visualization has focused on performance of message-passing systems, *e.g.*, understanding execution imbalance, communication to computation ratio, etc. Our work focuses on understanding the structure of shared-memory communication.

We do not claim a large contribution in software visualization *per se*, but to the best of our knowledge, this is the first work to visualize shared-memory inter-thread communication. We focus on what to visualize (the communication graph) rather than how to visualize it. Prefuse [7] met all our needs, but integrating communication graphs in software-visualization frameworks [8, 16] could be valuable.

Dynamic concurrency bug-detection tools use program instrumentation to monitor synchronization and accesses to shared data. Eraser [14] and AVIO [9] are canonical examples. Eraser detects when shared data is accessed without holding an appropriate lock. AVIO determines when a pair of accesses to a given memory location were supposed to be atomic but ended up interleaved by an access from another thread. We see our tool as a complementary addition to the available tools. The lack of atomicity violations or locking discipline violations does not mean the program is free of concurrency bugs; using our tool, a programmer can identify unexpected communication between functions.

7. Future Work

Work on communication graphs can go well beyond automatic generation via dynamic instrumentation as discussed in this paper. First, we would like to consider checking an execution's communication against a pre-computed graph. This graph could come from developers or be the union of the graphs produced by a test suite. Either way, it provides a way to detect unexpected communications. More generally and speculatively, our code-centric view can be the core of a shared-memory programming model in which the allowable shared-memory communication is defined as a communica-

Benchmark	Additional Runs					Across Inputs
	Initial	1	2	3	4	
blackscholes	6	0	1	0	0	0
bodytrack	253	14	18	1	1	46
canneal	19	3	0	0	0	0
dedup	76	2	6	8	3	12
facesim	168	7	4	6	3	12
ferret	145	0	1	0	0	2
fluidanimate	38	0	0	0	0	3
freqmine	124	6	0	0	3	14
raytrace	6	0	0	0	0	0
streamcluster	50	0	0	0	0	1
swaptions	46	7	1	2	0	12
vips	424	48	20	16	15	109
x264	510	1	1	0	0	255
httpd	1164	19	5	9	9	102
mysqld	1879	43	5	2	0	159

Table 2: Number of new edges discovered with additional runs of the same input. The last column gives the maximum number of new edges discovered by changing inputs.

tion graph. Language primitives or annotations could allow programmers to specify an acceptable graph — and what to do when communication exceeds the bounds of the graph.

8. Conclusion

We have developed a tool that monitors shared-memory multithreaded programs and builds a communication graph to describe which functions communicate data across threads. We have applied our tool to real programs. The graphs are of reasonable size to gain real insight into the structure of the applications. Indeed, the greatest benefit of our tool may be the invaluable but difficult-to-quantify issue of program understanding. However, the communication graphs also enable new metrics for describing the behavior of program executions such as the difference in communication graphs across different runs.

References

[1] Intel Thread Checker. http://software.intel.com/en-us/intel-thread-checker/.

[2] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/.

[3] R. Bertuli, S. Ducasse, and M. Lanza. Run-time information visualization for understanding object-oriented systems. In *International Workshop on Object-Oriented Reengineering*, 2003.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *ACM Conference on Programming Language Design and Implementation*, 2008.

[6] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, Sep./Oct. 1991.

[7] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A Toolkit for Interactive Information Visualization. In *SIGCHI Conference on Human Factors in Computing Systems*, 2005.

[8] M. Lanza and S. Ducasse. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 29(9), 2003.

[9] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, 2005.

[11] W. Nagel and A. Arnold. Performance visualization of parallel programs – the PARvis environment. In *Intel Supercomputer Users Group Conference*, 1994.

[12] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Conference on Object-Oriented Technologies and Systems*, 1998.

[13] L. D. Rose, Y. Zhang, and D. A. Reed. SvPablo: A Multi-language Performance Analysis System. In *Computer Performance Evaluation (Tools)*, 1998.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[15] J. Stasko. Animating algorithms with XTANGO. *SIGACT News*, Spring 1992.

[16] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. SHriMP Views: An Interactive Environment for Information Visualization and Navigation. In *SIGCHI Conference on Human Factors in Computing Systems*, 2002.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.

[18] A. Zeller and D. Lütkehaus. DDD – a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, Jan. 1996.