Binary Instrumentation Support for Measuring Performance in OpenMP Programs

Mustafa Elfituri
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
melfitur@nmsu.edu

Jeanine Cook
Department of Electrical and Computer Engineering
New Mexico State University
Las Cruces, NM 88003 USA
jcook@nmsu.edu

Jonathan Cook
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
joncook@nmsu.edu

Abstract—In parallel computations, evaluating the causes of poor speedup is an important development activity to reach the goal of creating the most efficient parallel computation possible. In our research on irregular parallel computations, especially graph algorithms, we had specific measurement needs for which a dearth of tools could be found. We created PGOMP, a small library-based profiling tool for the Gnu OpenMP implementation, and show its use here in discovering some of the causes of poor speedup in graph computations.

Index Terms—profiling, graph computation, measurement.

I. INTRODUCTION

It has long been known that different types of computations can require very different approaches to their parallelization, and that some computations are much easier to parallelize than others. In particular, irregular computations, such as those performed over large graph data structures, typically exhibit poor speedup relative to the resources available [1].

To understand what exactly is causing poor speedup for a particular program on a particular machine, we need to measure and evaluate the various overheads due to concurrent execution. Depending on the hardware resources available, this might be synchronization and lock waiting times, communication overhead, or shared memory resource contention and hierarchy overhead. Because performance is such a critical issue for computational science and engineering applications, software development environments for these parallel computations must include tools that give developers the ability to measure and analyze these aspects of the parallel performance of their applications. To this end, many tools exist, including commercial, open source, and ongoing experimental research tools that push the bounds of what we can observe and learn in our parallel software’s behavior.

We are beginning an investigation into the reasons that parallel graph computations run poorly on both shared memory and distributed architectures, with the goal of identifying characteristics that can be addressed by creating new and/or custom hardware support. Our expectation is that small hardware capability changes will have a significant impact on the performance of parallel graph computations, and hopefully on other irregularly parallel computations; others are evaluating irregular parallelism and its performance issues and are investigating similar avenues of solution, e.g., [1].

In pursuing this work we desire to have easy to use tools that measure various aspects of performance. In our initial work in measuring programs that use the OpenMP standard for parallelization, we saw the opportunity to make the process of measuring OpenMP parallel performance easier with a small tool that we call PGOMP, which measures overhead within the Gnu OpenMP implementation.

PGOMP is a useful addition to OpenMP toolsets because it does not interfere with source-processing tools, it is easy to use and self-contained, and it produces output that supports a variety of analyses. For example, PGOMP would be an ideal tool to slip into a nightly build and test process, giving performance feedback on the tests that can be tracked from build to build. PGOMP would also be an ideal tool in an educational environment, where simplicity, ease of use, and ease of installation and maintenance are at a premium.

This paper presents the motivation and initial construction of PGOMP, and initial results from using it to inspect the behavior of some parallel graph computations.

II. BACKGROUND

Many tools have been developed for analyzing parallel program performance, and many have adopted standard interfaces so that they can work together. Tau [2], VI-HPS [3], Periscope [4], OpenSpeedshop [5], VampirNG [6], and others have many similar capabilities or the ability to share information in order to use the others’ specialty capabilities.

Many of these toolsets were begun before OpenMP was introduced and are often focused on monitoring and measuring distributed message-passing parallel programs. However, most now do support integrating information about local shared-memory parallelism, including that enabled by OpenMP.

To do this, virtually all of them use the Opari OpenMP instrumentation package [2]. Opari (Opari2) is an OpenMP measurement tool that uses source-level instrumentation to add in the hooks needed to perform its measurements. It rewrites the OpenMP pragmas to include calls to its monitoring library API. Advantages of source instrumentation are that measurement information can often easily be related back to line numbers in the program and the approach is naturally cross platform at the source level, with only the backend needing rebuilt and perhaps customized for different platforms.
Major disadvantages of source instrumentation are that it disrupts the compilation tool chain and can introduce build problems as such, and either a full language parser is needed or one risks the possibility of an occasional instrumentation error that is wrong or perhaps does not even compile. Indeed these are the main reasons source instrumentation tools are often avoided in production environments. Tool chains can be somewhat fragile, and large code bases can often have features that the source instrumentor fails on. Indeed on one OpenMP forum a post indicated that Oparsi has some difficulties with some C++ syntax, and the Oparsi documents admit that the tool does “fuzzy” partial parsing.

Thus, while Oparsi is of obvious benefit in present environments, binary instrumentation tools are to be preferred. Indeed in the original Oparsi paper the authors’ main purpose was not to introduce a source instrumentation tool but rather to propose a profiling API standard, POMP, for OpenMP. If OpenMP defined a profiling API standard, then cross platform binary instrumentation tools could be created. This is what MPI has long done with its PMPI function interception (profiling) standard. As it is, commercial OpenMP compilers and toolsets often do provide binary instrumentors (e.g., Sun and IBM do). Developers at Sun even proposed their own OpenMP profiling API as a standard [7].

Since commercial compiler providers deem it useful to include binary OpenMP instrumentation tools in their toolsets, we think it would also be useful to have such tools for open-source compiler toolchains. The most common one is the Gnu compiler set, which has its own implementation of OpenMP, embodied in compiler support for the OpenMP pragmas, and in the libgomp library support for runtime control.

We looked at the symbol definitions exported by libgomp, and looked at the executable code emitted by the gcc compiler to see what calls into libgomp were being created where, and decided that it would be fairly straightforward to create a binary instrumentor for Gnu-compiled OpenMP programs. While we have only been working with C programs, the Gnu toolchain generally shares implementations and we see no reason our instrumentor would not work with Gnu-compiled C++ or Fortran programs.

III. METHODS

In desiring a binary instrumentor in our toolset, we created a lightweight instrumentation tool called PGOMP that profiles the Gnu implementation of OpenMP. As with many other tools, it is based on the library preloading mechanism, whereby a shared library can be preloaded into a program’s address space and dynamically linked name resolutions will then first match this preloaded library before being searched for in any other library. In this way we can interpose our own measurement functions in between the Gnu libgomp OpenMP library and thus collect timing statistics for various OpenMP overheads.

For each function $f$ that we interpose, we capture two timing values, $T_{begin}(f)$ and $T_{end}(f)$. These are the beginning and ending times of the actual library function $f$. Each function invocation can be identified by three values: the thread identifier $t$, the call location $l$, and the execution occurrence count $i$. Since PGOMP is Gnu-specific we use the Gnu builtin function _builtin_return_address() to retrieve the return address and use it as the call location identifier; it can then be further interpreted with symbol resolution.

From these timing values we can calculate at least two useful pieces of information. One is the time between a pair of related library functions—e.g., one that begins a particular OpenMP construct and one that ends it. This duration captures the time an application or thread spends in a particular OpenMP construct. The other is the duration that one particular Gnu OpenMP function takes to execute. This is useful where the body of a Gnu OpenMP function might block and wait.

The exact functions and information captured are detailed below. Functions unique to the Gnu implementation begin with the prefix “GOMP”, while functions that are part of the OpenMP API begin with “omp”.

A. Critical Section Overhead

A critical section is a section of code that only one thread is allowed to execute at any time, typically used to update or access a shared resource. Internally, the Gnu implementation uses the functions GOMP_critical_start and GOMP_critical_end. We calculate the duration of each execution of a critical section as

$$T_{crit}(t, l, i) = T_{begin,t,l,i}(GOMP\_critical\_end) - T_{end,t,l,i}(GOMP\_critical\_start)$$

and the time spent blocking to enter a critical section as

$$T_{critwait}(t, l, i) = T_{end,t,l,i}(GOMP\_critical\_start) - T_{begin,t,l,i}(GOMP\_critical\_start).$$

By experimentation we determined that GOMP_critical_start is where a thread blocks while waiting to get into a critical section, and the execution time of GOMP_critical_end is negligible.

B. Barrier Synchronization

Barriers are used to synchronize OpenMP threads at a point of execution, and cause overhead because earlier threads must wait for later threads. The Gnu OpenMP library has one function, GOMP_barrier, that implements the barrier capability. The individual thread barrier waiting time is

$$T_{barrier}(t, l, i) = T_{end,t,l,i}(GOMP\_barrier) - T_{begin,t,l,i}(GOMP\_barrier).$$

C. Explicit OpenMP Locks

OpenMP has an API function pair omp_set_lock and omp_unset_lock that an application can use to create their own synchronization. Time spent holding a lock is calculated as

$$T_{lock}(t, l, i) = T_{begin,t,l,i}(omp\_unset\_lock) - T_{end,t,l,i}(omp\_set\_lock).$$
and the time spent waiting to acquire a lock as
\[ T_{\text{lockwait}}(t, l, i) = T_{\text{end}, t, l, i}(\text{omp\_set\_lock}) - T_{\text{begin}, t, l, i}(\text{omp\_set\_lock}). \]

We have not yet instrumented the few other OpenMP lock functions as our current applications do not use them; for full coverage we will need to. The function \text{omp\_set\_lock} is the one that blocks, although an application could use \text{omp\_test\_lock} and avoid blocking in the library, which would make it hard to capture the performance loss due to not acquiring the lock. At the least we could provide a count of the number of times a lock was tested, indicating a measure of the cost of acquiring the lock.

\[ D. \text{ Data Accumulation} \]

The timing data described above can be used to elucidate many different aspects of parallel computation performance, and in the future we will be exploring using the data to its greatest extent. Currently we have only made use of per-thread and overall summations of waiting times. For example, for barrier synchronization we calculate the per-thread and overall barrier overhead as
\[ T_{\text{barrier}}(t) = \sum_l \sum_i T_{\text{barrier}}(t, l, i) \]
and for the application
\[ T_{\text{barrier}} = \sum_l T_{\text{barrier}}(t). \]
The other metrics’ summations are similarly accumulated. In the future we plan on investigating other timing metrics that can be calculated from the raw data; for example, the periodicity of a thread acquiring a lock would be calculated as
\[ T_{\text{lockperiod}}(t) = \frac{1}{N} \sum_i \sum_l T_{\text{begin}, t, l, i}(\text{omp\_set\_lock}) - T_{\text{begin}, t, l, i+1}(\text{omp\_set\_lock}) \]
where \( N \) is the number of intervals over all occurrences and locations.

\[ IV. \text{ Results} \]

We have begun using \textit{PGOMP} to experiment with assessing the performance of some graph algorithms, and present a small initial study here that shows the usefulness of \textit{PGOMP} along with some initial insights into performance issues.

In particular we are interested in examining the nature of parallel performance problems in graph-based computations. Thus, we began with two graph benchmark programs, the OpenMP version of the Graph500 benchmark [8] and the SSCA2 GraphAnalysis benchmark [9]. To compare these with a more traditional regular-arithmetic parallel computation we used a common OpenMP example program that computes the heating of a metal plate [10].

We used our \textit{PGOMP} measurement tool and ran examples from 1 to 32 threads on a 20-core 2-way hyperthreaded computer. The machine had two 10-core Intel Xeon CPU’s, each of which supports 2 hardware threads. Thus all threads in all runs can each be allocated their own (virtual) hardware core.

We used a problem size of 23 for the Graph500 program, a problem size of 23 for SSCA2, and a problem size of 2000 by 2000 for the heated plate program. Each configuration of each program was run five times and the mean of the 5 executions is what is reported in all of the tables and graphs below.

Figure 1 shows the speedup for all three applications. As expected, the regular parallelism of Heated Plate has the best performance, and at 16 threads it is performing significantly better than the graph computations. However, at 32 threads Graph500 comes close to the speedup of Heated Plate, which tails off of its earlier performance curve. We suspect the reason for this is the CPU hyperthreading—at 16 threads each thread can have its own physical core, while at 32 threads some must share physical cores through CPU hyperthreading. It would make sense that this would affect a more compute-bound computation like Heated Plate, while the graph algorithms are still constrained by other resources. As such, hyperthreading might be a useful and cost-effective feature for achieving parallel graph computation performance, and more study is warranted to explore this possibility.

Figure 2 shows the overall time the applications’ threads spend blocked and waiting. The first thing to notice is that it is fairly low for all applications, relative to their speedup performance. For example, at 16 threads the Graph500 benchmark has 6.7% of its total thread execution time being spent blocked and waiting, but its speedup of about 9 indicates that roughly 44% of its execution time is lost in some kind of

\[ \begin{array}{|c|c|c|c|}
\hline
\text{Application} & \text{Barrier Wait} & \text{Critical Sec Wait} & \text{Lock Wait} \\
\hline
\text{Graph500} & 6.7 & 0 & 0 \\
\text{SSCA2} & 5.0 & 0 & 0.64 \\
\text{Heated Plate} & 0.9 & 0.0002 & 0 \\
\hline
\end{array} \]
overhead. Table I shows the overall waiting overhead broken down into the three different types of overhead that PGOMP measures. The Graph500 benchmark is a read-only benchmark and so only uses synchronization barriers, while the other two have some lock or critical section overhead. SSCA2 uses explicit OpenMP locks while Heated Plate uses critical section pragmas. In SSCA2 and Heated Plate the synchronization barriers are still the majority of overhead, indicating that lock contention is not a major issue in these applications.

Figures 3, 4, and 5 show barrier wait times for all threads for each of the Graph500, SSCA2, and Heated Plate applications. Noting the graph scales, SSCA2 has by far the highest wait times on average, with Heated Plate having the lowest. Graph500 at 16 threads shows a wide distribution of waiting costs per thread, while at 32 its threads cluster very well at a low overhead. Both SSCA2 and Heated Plate the synchronization barriers are still the majority of overhead, indicating that lock contention is not a major issue in these applications.

Figures 3, 4, and 5 show barrier wait times for all threads for each of the Graph500, SSCA2, and Heated Plate applications. Noting the graph scales, SSCA2 has by far the highest wait times on average, with Heated Plate having the lowest. Graph500 at 16 threads shows a wide distribution of waiting costs per thread, while at 32 its threads cluster very well at a low overhead. Both SSCA2 and Heated Plate the synchronization barriers are still the majority of overhead, indicating that lock contention is not a major issue in these applications.

Except one low-waiting outlier; this one thread could be a significant cause of slowdown, since all other threads must wait for it to reach the barrier, and should be investigated further to see if there is a programmatic cause for it that can be fixed.

Figure 6 shows lock waiting times for SSCA2. The graph shows a nice uniform cost behavior for locking access to shared-data updates, and expected decreasing contention per thread as more threads share the work.

Figure 7 shows, similar to the lock overhead, the critical section overhead for Heated Plate. Though the figure shows less uniformity among thread overhead costs, the graph Y axis scale shows that these are very small overheads in comparison to other performance costs.

Evaluating the results, because the measured overhead for the OpenMP synchronization and lock primitives is a low fraction of the overall speedup deficiency of the graph algorithms, we surmise that the main component of performance degradation is in hardware resource contention, mainly in the memory hierarchy, since the parallel computing model here is shared memory parallelism. We are currently progressing...
in this investigation, and are evaluating choices in how to investigate memory hierarchy performance issues, especially shared data contention.

This is one area that has long been a difficult one to investigate. Hardware cache performance counters do exist, but most CPUs do not implement cache coherency counters, and other cache counters give generic hit/miss counts that offer little insight into what is happening in terms of core contention. Memory traces can be created, but such tracing overhead greatly perturbs the actual application and thus the parallel memory accesses that end up in the trace are not necessarily related much to the application’s full speed performance. Full cycle-accurate simulations of hardware, including the memory hierarchy, are extremely time intensive and can only be done on small problem sizes, and have their own issues in relating the results to actual hardware.

V. CONCLUSION AND FUTURE WORK

We presented PGOMP, a binary instrumentor for performance measurement of Gnu-compiled OpenMP programs. The Gnu toolchain is widely-enough used that it is useful to create a specific binary instrumentor for it, as commercial OpenMP toolchain providers have done. Binary instrumentation avoids the issues of interfering with the development build tools that source instrumentors such as Opapi do, and offer a clean, easy-to-use mechanism to provide performance analysis feedback to the development process.

The basic information that PGOMP produces can be reduced to many other metrics besides the ones we used here in the small example reported in this paper, and we are continuing to investigate other metrics based on the data that PGOMP produces. Such metrics include the number of occurrences of blocking, key locks that get blocked more, periodicity of blocking, duration of lock holding, and many others.

An important and very useful extension to our tool will be to enable it to produce data in the open formats that other performance tools support, such as the trace format OTF that Vampir can use. In this case PGOMP could act as a replacement to Opapi, in case developers wanted to (or needed to) avoid source instrumentation. We are already working on such usability extensions.

ACKNOWLEDGEMENTS AND AVAILABILITY

We thank the anonymous reviewers of this paper for their helpful comments and ideas. This material is based upon work supported in part by the National Science Foundation under Grant No. CCF-1111798.

PGOMP is available at www.cs.nmsu.edu/please/pgomp.

REFERENCES