John von Neumann Institute for Computing

Towards an Implementation of the OpenMP Collector API

Van Bui, Oscar Hernandez, Barbara Chapman,
Rick Kufrin, Danesh Tafti, Pradeep Gopalkrishnan

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 573-580, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

Towards an Implementation of the OpenMP Collector API

Van Bui[1], Oscar Hernandez[1], Barbara Chapman[1], Rick Kufrin[2], Danesh Tafti[3], and Pradeep Gopalkrishnan[3]

[1] Department of Computer Science
University of Houston
Houston, TX 77089
E-mail: {*vtbui, oscar, chapman*}*@cs.uh.edu*

[2] National Center for Supercomputing Applications
University of Illinois
Urbana, IL, 61801
E-mail: rkufrin@ncsa.uiuc.edu

[3] Department of Mechanical Engineering,
Virginia Tech
Blacksburg, VA, 24061
E-mail: {*dtafti, pradeepg*}*@vt.edu*

Parallel programming languages/libraries including OpenMP, MPI, and UPC are either in the process of defining or have already established standard performance profiling interfaces. The OpenMP Architecture Review Board (ARB) recently sanctioned an interface specification for profiling/tracing tools that defines a protocol for two-way communications and control between the OpenMP runtime library and performance tools, known as the collector API. Reference implementations of the collector are sparse and are primarily *closed-source*. We provide a description of our efforts towards a full implementation of an *open-source* performance monitoring tool for OpenMP based on the collector API. This effort[a] evaluates the collector's approach to performance measurement, assesses what is necessary to implement a performance tool based on the collector interface, and also provides information useful to performance tool developers interested in interfacing with the collector for performance measurements.

1 Introduction

Many scientific computing applications are written using parallel programming languages/libraries. Examples include applications in mechanical engineering, neuroscience, biology, and other domains[1–3]. Software tools, such as integrated development environments (IDEs), performance tools, and debugging tools are needed that can support and facilitate the development and tuning of these applications. Parallel programming libraries/languages such as OpenMP[4], MPI[5], and UPC[6] are in the process of drafting or have already standardized/sanctioned a performance monitoring interface[7–10]. Performance tools must be able to present performance information in a form that captures the user model of the parallel language/library and a performance monitoring interface enables this. The task of relating performance data in terms of the user model can be challenging for some languages. In the case of OpenMP, the compiler translates OpenMP directives inserted into the source code into more explicitly multi-threaded code using the OpenMP runtime library. Measuring the performance of an OpenMP application is complicated

[a]This work is supported by the National Science Foundation under grant No. 0444468, 0444319, and 0444407.

by this translation process, resulting in performance data collected in terms of the implementation model of OpenMP. Performance tool interfaces have emerged for OpenMP to provide a *portable* solution to implementing tools to support presentation of performance data in the user model of the language[10,9].

The performance monitoring API for OpenMP, known as the collector interface, is an event based interface requiring *bi-directional* communications between the OpenMP runtime library and performance tools[11]. The OpenMP Architecture Review Board (ARB) recently sanctioned the collector interface specifications. Reference implementations to support development of the collector are sparse. The only known *closed-source* prototype implementation of the collector is provided by the Sun Studio Performance Tools[10]. To the best of our knowledge, there are currently no *open-source* implementations of the collector API. Our goal is to provide such a reference implementation for the research community. We have designed and implemented an API that leverages the OpenMP runtime library to support performance monitoring of OpenMP applications. Our performance monitoring system consists of an OpenMP runtime library, a low-level performance collector that captures hardware performance counters, and a compiler to support mapping performance back to the source level.

In the remainder sections of this paper, we provide a more implementation focused discussion of the OpenMP collector API, detail our own experiences in implementing and evaluating a performance monitoring system that leverages the OpenMP runtime library, present a performance case study to demonstrate the usefulness of our performance monitoring system, and finally end with our conclusions from this study.

2 The Collector Runtime API for OpenMP

The collector interface specifies a protocol for communication between the OpenMP runtime library and performance tools. A prior proposal for a performance tool interface for OpenMP, known as POMP[9], was not approved by the committee. POMP is an interface that enables performance tools to detect OpenMP events. POMP supports measurements of OpenMP constructs and OpenMP API calls. The advantages of POMP includes the following: the interface does not constrain the implementation of OpenMP compilers or runtime systems, it is compatible with other performance monitoring interfaces such as PMPI, and it permits multiple instrumentation methods (e.g. source, compiler, or runtime). Unfortunately, if the compiler is unaware of these instrumentation routines, they can interfere with static compiler analysis and optimizations, which affects the accuracy of gathered performance data.

In contrast, the collector interface is independent of the compiler since it resides and is implemented inside the OpenMP runtime. The primary advantage of the collector interface is that the application source code remains unmodified since instrumentation is not required at the source or compiler level. Consequently, data collection will interfere much less with compiler analysis/optimizations and a more accurate picture of the performance of the application is possible. The design also allows for the collector and OpenMP runtime to evolve independently. However, overheads must be carefully controlled from both the side of the OpenMP runtime and from collecting the measured data in order to obtain the most accurate results. A second drawback is that the interface provides little support for distinguishing between different loops inside a parallel region and corresponding barrier regions. Given these drawbacks, much additional software support and efficient algorithms are required to properly address these concerns.

2.1 Rendezvous Between OpenMP Runtime and Collector

The OpenMP runtime and collector communicate via the collector interface. The collector initiates communications and also makes queries to the OpenMP runtime via the collector interface. The collector interface consists of a single routine that takes the following form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array containing one or more requests. For example, the collector can request that the OpenMP runtime notifies it when a specific event occurs. The input parameters sent in this case would include the name of the event and the callback routine to be invoked from inside the OpenMP runtime when this specific event occurs. Figure 1 depicts this request more precisely along with other requests that can be made by the collector. The collector may make requests for the current state of the OpenMP runtime, the current/parent parallel region ID (PRID), and to pause/resume generation of events.

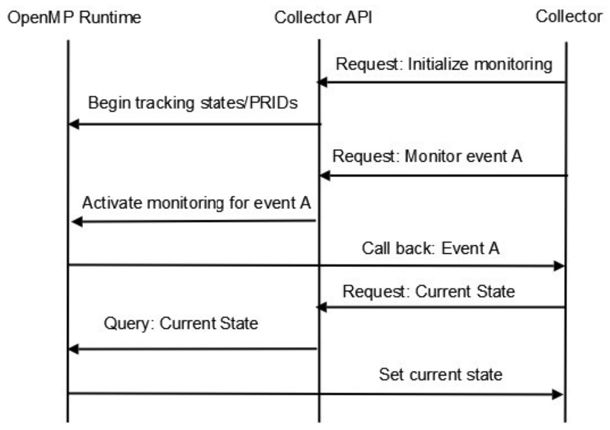

Figure 1. Example of sequence of requests made by collector to OpenMP runtime.

2.2 Performance Tool Support

Performance tools may use the collector interface to support performance measurements of OpenMP codes. The performance tool typically queries the OpenMP runtime for information (e.g. event notifications, thread state, etc), records performance information, and captures the runtime call stack. The call stack that is captured at runtime is based on the implementation model of OpenMP since the translated code includes calls to the OpenMP runtime library. Performance data should be presented in the user model of OpenMP since the OpenMP specification is written in terms of the user model. Using the support provided by the OpenMP runtime and realizing the support required on the tools side, the user model call stack can be reconstructed from the implementation model call stack for each thread.

There are certain expectations that must be met by supporting system and auxiliary user-level software to allow the approach we describe to provide the most accurate performance feedback. At the lowest level, the operating system should maintain the performance data of interest in a manner such that the data is virtualized on a per-thread basis (that is, it must be possible to extract performance measurements isolated to a single thread, not across an entire process, group, or the entire system). The optimal situation is one in which the underlying thread model is "one-to-one", where every user level thread is bound to a single kernel thread. In this case, the kernel can maintain the required virtualized performance data as part of the overall thread context and return this data as-is in response to a query from a performance tool. Similarly, it must be possible to retrieve at runtime the call stack associated with a single thread in an efficient manner.

2.3 Support Inside the OpenMP Runtime

The OpenMP runtime is primarily responsible for implementation of the collector interface. Performance tools may make several requests to the OpenMP runtime library via the collector interface. To satisfy these requests, the OpenMP runtime library needs to implement support to (1) initiate/pause/resume/stop event generation, (2) respond to queries for the ID of the current/parent parallel region, and (3) respond to queries for the current state of the calling thread.

The collector initiates interactions with the OpenMP runtime by making a request to the OpenMP runtime to create and initiate the data structures required to store the state of each thread and also to track the current/parent parallel region ID. Updates and accesses to these data structures by each of the threads must be properly managed to minimize overhead.

When the collector makes a request for notification of a specified event (s), the OpenMP runtime will activate monitoring for this event inside its environment. The collector may also make requests to pause, resume, or stop event generation. To make a request for notification of a specific event, the collector passes along the name of the event to monitor as well as a callback routine to be invoked by the OpenMP runtime to notify the collector each time the event occurs. Examples of events include fork, join, entry/exit into a barrier, entry/exit into a master region, entry/exit into an idle state, and several others. The collector interface specification requires that the OpenMP runtime provide support for generating the fork event and specifies support for the other events as optional to support tracing. Some issues to consider regarding implementing this functionality includes the overhead associated with activating/inactivating event generation and overheads from invoking the call back routine. These overheads should be minimized in order to obtain the most accurate performance measurements.

3 Building a Performance Tool Inside the OpenMP Runtime

The process of building a performance tool based on the collector interface consists of a series of steps. We first select the software components that will form the infrastructure for our tool. Second, we assess whether any changes or extensions are needed in order for the software components to communicate with one another to perform the specified tasks of the end tool. Third, we build a prototype tool to assess the overheads associated with data collection and to also evaluate the accuracy of the generated data for performance tuning. Finally, we move forward to build the end tool.

3.1 Software Components

The set of software components needed to implement a performance profiling tool based on the collector API includes a compiler's OpenMP runtime library, a performance monitoring interface for OpenMP, and a performance collection tool. We use the OpenUH[12] compiler's OpenMP runtime library and PerfSuite's[13] measurement libraries as the collector for our profiling tool. Extensions are needed for both the OpenMP runtime library and the collector in order to realize our final prototype tool. The software components that we employ in this development effort are open-source and freely available to the research community.

To support building a prototype performance tool based on the collector interface, we designed a performance monitoring interface for OpenMP, called PerfOMP[14]. PerfOMP was designed to monitor the runtime execution behavior of an OpenMP application with the help of the OpenMP runtime library. The current PerfOMP event based interface includes support for monitoring fork, join, loop, and barrier regions. Each interface routine takes only a few parameters including the thread number, thread count, and parallel region ID. PerfOMP maps OpenMP events with identifiers such as the thread number and parallel region number. PerfOMP supports the development of performance profiling and tracing tools for OpenMP applications in a manner that is transparent to the user.

3.2 Instrumentation

The compiler translates OpenMP directives with the help of its OpenMP runtime library. An example of this transformation process in the OpenUH compiler is shown in Fig. 2a and Fig. 2b. Figure 2c shows an example of PerfOMP instrumentation points inside the OpenMP runtime library of OpenUH. Figure 2c shows instrumentation points inside the *ompc_fork* operation that captures the following events: fork, join, parallel begin/end for the master thread, and the implicit barrier entry/exit for the master thread. To capture events for the slave threads requires instrumentation of a separate routine in the runtime library. The entire instrumentation of the OpenMP runtime library of the OpenUH compiler required minimal programming effort. This instrumentation enables collecting measurements of several events including the overall runtime of the OpenMP library, of individual parallel regions and several events inside a parallel region for each thread.

3.3 Mapping Back to Source

Mapping the performance data back to the source code was accomplished with the help of the OpenUH compiler. Modifications were made in the OpenMP translation by the OpenUH compiler so that it dumps to an XML file source level information. Each parallel region is identified by a unique ID. Unique IDs can also be generated for other parallel constructs. This required changes in the OpenMP compiler translation phase to add an additional integer parameter variable to the fork operation to hold and pass along the parallel region ID to the OpenMP runtime. A map file containing the source level mappings is generated for each program file and is created statically at compile time.

```
F90 Parallel Region
-------------------------

program test

  ...

  !omp parallel do
  do i=0,N,1
    a(i)=b(i)*c(i)
  enddo

  ...

end program
```

(a)

```
Compiler Translated OpenMP Code
-----------------------------------------------------

program test
  ...
  ompc_fork(ompdo_test_1, ...)
  ....

  !** Translated Parallel Region
  subroutine
    ...
    !** Signifies beginning of a loop
    ompc_static_init_4(...)
    ...
    !** Determine upper and lower work bounds
    do  __locali = do_lower_0, do_upper_0, 1
      A(__locali) = B(__locali)*C_0(__locali)
    enddo
    !** Signifies end of loop
    ompc_static_fini(...)
    ...
    return
  end subroutine

...
end program
```

(b)

```
OpenMP Runtime Library Routine Using PerfOMP
-----------------------------------------------------

void ompc_fork(...) {

  // Library initialization
  perfomp_parallel_fork(thread_num, id,
                        n_threads);

  // Thread team initializations
  perfomp_parallel_begin(thread_num, id);

    // execute microtask for master thread
    microstask(0, frame_pointer);

    perfomp_parallel_end(thread_num, id);

    perfomp_barrier_enter(thread_num, id);

      ompc_level_barrier(0);

    perfomp_barrier_exit(thread_num, id);

  perfomp_parallel_join(thread_num, id);

} // end ompc_fork
```

(c)

Figure 2. (a) OpenMP parallel region in Fortran. (b) Compiler translation of the piece of OpenMP code from part (a). (c) A PerfOMP instrumented OpenMP runtime library routine.

As an alternative approach to support mapping back to source, we have extended PerfSuite's library routines to also return the runtime call stack. The advantage of obtaining the source level mappings via the compiler is that we can get a direct mapping between the performance data and the compiler's representation of the program. This will enable the compiler to more easily utilize the generated data to improve its optimizations. Although this mapping is not context sensitive in that it does not distinguish between different calling contexts of a parallel region. Gathering source level mappings at runtime using the call stack will allow us to distinguish the different calling contexts and this additional capability motivates our use of runtime stack information gathered through PerfSuite. A performance tool should be able to reconstruct the user model call stack using the retrieved implementation model call stack. The extensions being made to PerfSuite will provide a parameter variable allowing the caller to specify the number of stack frames to skip or ignore to facilitate this reconstruction. To control the overheads associated with retrieving large stack frames, PerfSuite will also provide a parameter for specifying the maximum number of instruction pointers to retrieve. These extensions to PerfSuite will be used in our implementation of a performance tool based on the collector interface.

4 Experimental Results

To demonstrate the usefulness of the data gathered from our prototype measurement system, we present a case study where we analyze the performance of GenIDLEST[1]. GenIDLEST uses a multiblock structured-unstructured grid topology to solve the time-dependent Navier-Stokes and energy equations. We use its 45rib input data set. All the experiments were performed on an SGI Altix 3700 distributed shared memory system and using eight threads. The OpenMP version of GenIDLEST is compiled with the OpenUH compiler with optimization level 3 and OpenMP enabled.

4.1 Case Study: Bottleneck Analysis of GenIDLEST

Several different methodologies exist for utilizing performance hardware counters for performance analysis. We apply the methodology for detecting bottlenecks using the Itanium 2 performance counters as described by Jarp[15]. This technique applies a drill down approach, where the user starts counting the most general events and drills down to more fine grained events. Applying this measurement technique, GenIDLEST shows an estimated 8% of additional overheads when we activate performance measurements with PerfOMP for all OpenMP parallel regions in the code.

First, we measure the percentage of stall cycles and find that stall cycles account for $\sim$70% of the total cycles. The data also shows that the parallel region inside the **diff_coeff** subroutine is taking up about 20% of the execution time and spending about 3.34% of time waiting inside the barrier at the end of a parallel region. We collect additional performance counters for the **diff_coeff** subroutine to identify and resolve the bottleneck for that region of code.

The performance data for the parallel region inside **diff_coeff** shows $\sim$35% data cache stalls, $\sim$23% instruction miss stalls, and $\sim$28% of the stalls are occurring in the floating point unit (FLP). Relating the memory counters to the stall cycles, we found that $\sim$60% of the memory stalls result from L2 hits and $\sim$38% from L3 misses.

We apply transformations to the parallel region inside **diff_coeff** to better utilize the memory hierarchy. We apply modifications of variable scope from shared to private to this subroutine. Privatizing the data should relieve the bottlenecks associated with remote memory accesses since privatization ensures that each thread will have its own private local copy of the data.

After privatization of the data in the **diff_coeff** subroutine, we observe a $\sim$25% improvement in overall runtime of GenIDLEST. The parallel region inside the **diff_coeff** subroutine now only takes up $\sim$5.5% of the execution time compared to the previous estimated 20%. Furthermore, the wait time spent inside the barrier region also significantly improved by a factor of 10X. Performance counting data from the parallel region inside **diff_coeff** also shows a significant drop in stall cycles ($\sim$80-90% decrease).

5 Conclusions and Future Work

An implementation-focused discussion of the OpenMP ARB sanctioned collector interface is presented to support tool developers interested in implementing support for the collector API. We describe the necessary software components and extensions for implementing a performance tool based on the collector API. A prototype performance profiling tool that emulates the functionality of a tool based on the collector API is also presented and shown to generate useful and accurate performance data for performance tuning purposes.

Building on top of the current software infrastructure we have laid out, plans are currently underway to provide a full implementation of an open-source performance tool based on the collector interface. The remaining tasks include finalizing the extensions being made to PerfSuite that enable runtime call stack access, implementing the collector interface inside the OpenMP runtime library of OpenUH, and designing/implementing efficient algorithms for mitigating overheads that can incur both inside the OpenMP runtime library and from the collector. Our goal is to encourage more compiler developers to implement this support inside their respective OpenMP runtime libraries, thereby enabling more performance tool support for OpenMP users.

References

1. D. K. Tafti, *GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows*, in: Proc. ASME Fluids Engineering Division, (2001).
2. H. Markram, *Biology—The blue brain project*, in: SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing, p. 53, (ACM Press, NY, 2006).
3. C. Chen and B. Schmidt, *An adaptive grid implementation of DNA sequence alignment*, Future Gener. Comput. Syst., **21**, 988–1003, (2005).
4. L. Dagum and R. Menon, *OpenMP: an industry-standard API for shared-memory programming*, IEEE Comput. Sci. Eng., **5**, 46–55, (1998).
5. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Tech. Rep. UT-CS-94-230, (1994).
6. T. El-Ghazawi, W. Carlson, T. Sterling and K. Yelick, *UPC: Distributed Shared Memory Programming*, (John Wiley & Sons, 2005).
7. M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, (MIT Press, Cambridge, Mass., 1996).
8. H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley III and A. George, *GASP! A standardized performance analysis tool interface for global address space programming models*, Tech. Rep. LBNL-61659, Lawrence Berkeley National Lab, (2006).
9. B. Mohr, A. Malony, H. C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger and S. Shah, *A performance monitoring interface for OpenMP*, in: Proc. 4th European Workshop on OpenMP, (2002).
10. G. Jost, O. Mazurov and D. an Mey, *Adding new dimensions to performance analysis through user-defined objects*, in: IWOMP, (2006).
11. M. Itzkowitz, O. Mazurov, N. Copty and Y. Lin, *White paper: an OpenMP runtime API for profiling*, Tech. Rep., Sun Microsystems, (2007).
12. C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, *OpenUH: an optimizing, portable OpenMP compiler*, in: 12th Workshop on Compilers for Parallel Computers, (2006).
13. R. Kufrin, *PerfSuite: an accessible, open source performance analysis environment for Linux*, in: 6th International Conference on Linux Clusters: The HPC Revolution 2005, (2005).
14. V. Bui, *Perfomp: A runtime performance/event monitoring interface for OpenMP*, Master's thesis, University of Houston, Houston, TX, USA, (2007).
15. S. Jarp, *A Methodology for using the Itanium-2 Performance Counters for Bottleneck Analysis*, Tech. Rep., HP Labs, (2002).