Visualizing the Program Execution Control Flow of OpenMP Applications *

Karl Fürlinger and Shirley Moore

Innovative Computing Laboratory,
EECS Department,
University of Tennessee, Knoxville
{karl, shirley}@eecs.utk.edu

Abstract. One important aspect of understanding the behavior of an application with respect to its performance, overhead, and scalability characteristics is knowledge of its control flow. In comparison to sequential applications the situation is more complicated in multithreaded parallel programs because each thread defines its own independent control flow. On the other hand, for the most common usage models of OpenMP the threads operate in a largely uniform way, synchronizing frequently at sequence points and diverging only to operate on different data items in worksharing constructs.
This paper presents an approach to capture and visualize the control flow of OpenMP applications in a compact way that does not require a full trace of program execution events but is instead based on a straightforward extension to the data collected by an existing profiling tool.

1 Introduction

An important aspect of understanding the behavior of a parallel application is knowledge about its control flow. In the context of this paper we define the *control flow* as the sequence in which an application executes blocks of code, where a block of code might be as big as a function body or as small as individual statements. Typically, as we will discuss later, in our approach the individual elements of the control flow representations are the source code regions corresponding to whole OpenMP constructs such as parallel regions, critical sections, functions, or user-defined regions. A user can add individual statements to the control flow representation by manually instrumenting them, but typically the user-defined regions would be larger and at least contain a couple of statements.

To motivate the benefit of knowing the control flow of an application, consider the following simple example. Assume our application calls two functions `foo()` and `bar()` as show in Fig. 1a. The `gprof` output corresponding to an execution of this application is shown in Fig. 1c. Now consider the alternative version in Fig. 1b. Analyzing these two applications with `gprof` gives exactly the same profile, even though the control flow with respect to the functions `foo()` and `bar()` is different. In the first example `bar()` is always called after `foo()` (20 times) while in the second case `foo()` is the predecessor of `bar()`

* This work was partially supported by US DOE SCIDAC grant #DE-FC02-06ER25761 (PERI) and NSF grant #07075433 (SDCI).

in the control flow only once (at the beginning of the loop), while it is its own predecessor 19 times. This is visualized in Figs. 1d and 1e, respectively.

```
void main() {
  int i;
  for( i=0; i<20; i++ ) {
    foo();
    /* ... */
    bar();
  }
}
```

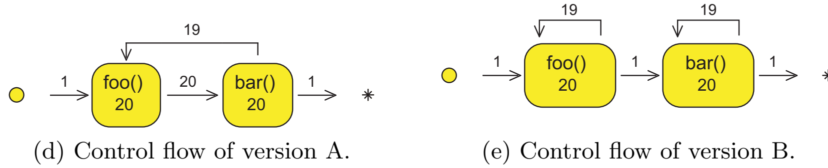
(a) Version A.

```
void main() {
  int i;
  for( i=0; i<20; i++ ) {
    foo();
  }
  /* ... */
  for( i=0; i<20; i++ ) {
    bar();
  }
}
```

(b) Version B.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    9.77                 main [1]
                4.94    0.00      20/20          foo [2]
                4.83    0.00      20/20          bar [3]
-----------------------------------------------
                4.94    0.00      20/20          main [1]
[2]     50.6    4.94    0.00      20         foo [2]
-----------------------------------------------
                4.83    0.00      20/20          main [1]
[3]     49.4    4.83    0.00      20         bar [3]
-----------------------------------------------
```

(c) `gprof` profile, versions A and B.



(d) Control flow of version A. (e) Control flow of version B.

Fig. 1: A simple example demonstrating that differences in the control flow are not reflected in runtime profiles.

Knowledge about the control flow can be important with respect to performance considerations related to data locality and reuse. If `foo()` and `bar()` work on the same data items, version A keeps data in cache which can be beneficial over version B, which iterates over all data items twice. Evidently the control flow information is not retained in the `gprof` profiles, as in both cases the functions have been called the same number of times and in both cases `bar()` as well as `foo()` have been called from `main()`. Hence, analyzing the callgraph cannot uncover the control flow information.

One approach to recover the control flow is of course to do a full trace of all enter and exit events of all interesting functions, constructs or other source code regions and to visually analyze this trace with tools like Vampir [9], Intel Trace Analyzer [4] or Paraver [10]. However, with raw trace visualization it can be cumbersome to visualize the essential parts of the control flow as the number of events is often overwhelming. In this paper, we discuss an approach that

shows that full tracing is not necessary and that the control flow information can be uncovered using a simple extension of a profiling tool.

The rest of this paper is organized as follows: the next section introduces the profiling tool that we extended to extract the control flow information and describes the necessary extensions. Sect. 3 then discusses the visualization of the control flow for OpenMP constructs and presents an example control flow of an application from the NAS parallel benchmark suite. In Sect. 4 we describe related work and in Sect. 5 we conclude and outline directions for future work.

2 The OpenMP Profiler `ompP`

`ompP` is a profiling tool for OpenMP applications designed for Unix-like systems. Since it is independent of the OpenMP compiler and runtime system, it works with any OS/compiler combination. `ompP` differs from other profiling tools like `gprof` or OProfile [7] in primarily two ways. First, `ompP` is a measurement based profiler and does not use program counter sampling. The instrumented application invokes `ompP` monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). The direct measurement approach can potentially lead to higher overheads when events are generated very frequently, but this can be avoided by instrumenting such constructs selectively. An advantage of the direct approach is that the results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts.

The second difference lies in the way of data collection and representation. While general profilers work on the level of functions, `ompP` collects and displays performance data in the user model of the execution of OpenMP events [5]. For example, the data reported for critical section contain not only the execution time but also list the time to enter and exit the critical construct (`enterT` and `exitT`, respectively) as well as the accumulated time each threads spends inside the critical construct (`bodyT`) and the number of times each thread enters the construct (`execC`). An example profile for a critical section is given in Fig. 2

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT
   0       1.00          1       1.00       0.00       0.00
   1       3.01          1       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       0.00
   3       4.01          1       1.00       3.01       0.00
 SUM      10.02          4       4.01       6.01       0.00
```

Fig. 2: Profiling data delivered by `ompP` for a critical section.

Profiling data in a similar style is delivered for each OpenMP construct, the columns (execution times and counts) depend on the particular construct. Fur-

thermore, ompP supports the query of hardware performance counters through PAPI [1] and the measured counter values appear as additional columns in the profiles. In addition to OpenMP constructs that are instrumented automatically using Opari [8], a user can mark arbitrary source code regions such as functions or program phases using a manual instrumentation mechanism. Function calls are automatically instrumented on compilers that support this feature (e.g., -finstrument-functions) for the GNU compilers

Profiling data are displayed by ompP both as flat profiles and as callgraph profiles, giving both inclusive and exclusive times in the latter case. The callgraph profiles are based on the callgraph that is recorded by ompP. An example callgraph is shown in Fig. 3. The callgraph is largely similar to the callgraphs given by other tools, such as callgrind [11], with the exception that the nodes are not only functions but also OpenMP constructs and user-defined regions, and the (runtime) nesting of those constructs is shown in the callgraph view. The callgraph that ompP records is the union of the callgraph of each thread. That is, each node reported has been executed by at least one thread.

```
    ROOT  [critical.i686.ompp: 4 threads]
  REGION  +-R00004 main.c (40-51) ('main')
PARALLEL     +-R00005 main.c (44-48)
  REGION        |-R00001 main.c (20-22) ('foo')
  REGION        |  +-R00002 main.c (27-32) ('bar')
CRITICAL        |     +-R00003 main.c (28-31) (unnamed)
  REGION        +-R00002 main.c (27-32) ('bar')
CRITICAL           +-R00003 main.c (28-31) (unnamed)
```

Fig. 3: Example callgraph view of ompP.

2.1 Data Collection to Reconstruct the Control Flow Graph (CFG)

As discussed in the introduction, the callgraph does not contain enough information to reconstruct the CFG. However, a full trace is not necessary either. It is sufficient to keep a record that lists all predecessor nodes and how often the predecessors have been executed for each callgraph node. A predecessor node is either the parent node in the callgraph or a sibling node on the same level. A child node is not considered a predecessor node because the parent–child relationship is already covered by the callgraph representation. An example of this is shown in Fig. 4. The callgraph (lower part of Fig. 4) shows all possible predecessor nodes of node A in the CFG. They are the siblings B and C, and the parent node P. The numbers next to the nodes in Fig. 4 indicate the predecessor nodes and counts after one iteration of the outer loop (left hand side) and at the end of the program execution (right hand side), respectively.

Implementing this scheme in ompP was straightforward. ompP already keeps a pointer to the *current* node of the callgraph (for each thread) and this scheme

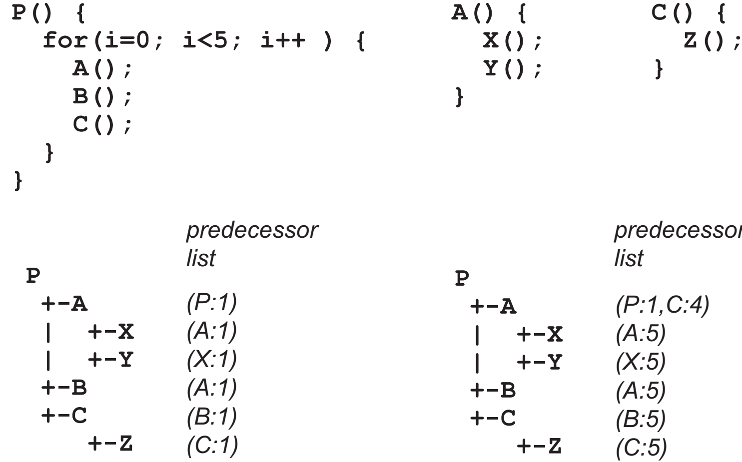


Fig. 4: Illustration of the data collection process to reconstruct the control flow graph.

is extended by keeping a *previous* node pointer as indicated above. Again this information is kept on a per-thread basis, since each thread can have its own independent callgraph as well as flow of control.

The previous pointer always lags the current pointer one transition. Prior to a parent → child transition, the current pointer points to the parent while the previous pointer either points to the parent's parent or to a child of the parent. The latter case happens when in the previous step a child was entered and exited. In the first case, after the parent → child transition the current pointer points to the child and the previous pointer points to the parent. In the latter case the current pointer is similarly updated, while the prior pointer remains unchanged. This ensures that the previous nodes of siblings are correctly handled.

With current and previous pointers in place, upon entering a node, information about the previous node is added to the list of previous nodes with an execution count of 1, or, if the node is already present in the predecessor list, its count is incremented.

3 Visualizing the CFG of OpenMP Applications

The data generated by `ompP`'s control flow analysis can be displayed in two forms. The first form visualizes the control flow of the whole application, the second is a layer-by-layer approach. The full CFG is useful for smaller applications, but for larger codes it can quickly become too large to comprehend and cause problems for automatic layout mechanisms. An example of an application's full control flow is shown in Fig. 5. The code corresponds to the callgraph of Fig. 3 where the critical section's body contains work for exactly one second.

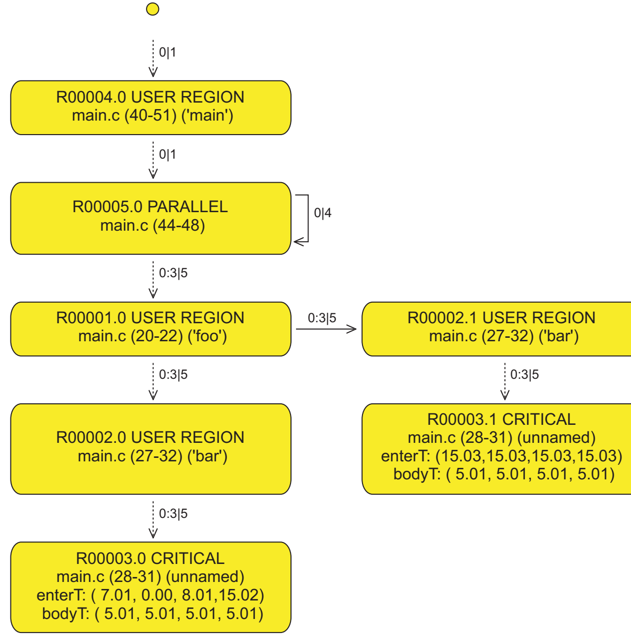


Fig. 5: An example for a full control flow display of an application.

Rounded boxes represent source code regions. That is, regions corresponding to OpenMP constructs, user-defined regions or automatically instrumented functions. Solid horizontal edges represent the control flow. An edge label like $i|n$ is interpreted as thread i has executed that edge n times. Instead of drawing each thread's control flow separately, threads with similar behavior are grouped together. For example the edge label 0–3|5 means that threads $0, 1, 2$, and 3 combined executed that edge 5 times in total. This greatly reduces the complexity of the control flow graph and makes it easier to understand.

For each node the box contains the most important information. This includes the type of the region (such as `CRITICAL`), the source code location (file name and line number) and performance data. Due to space limitations the included performance data do not list the full profile but only the most important aspects for the particular construct. This information includes the overall execution time as well as the most likely cause for a potential bottleneck. For critical sections this is the time required to enter the construct (`enterT`) and for parallel loops it is the waiting time at the implicit barrier, for example.

Dotted vertical lines represent control flow edges from parent to child (with respect to the callgraph). The important difference in interpreting these two types of edges is that a solid edge from A to B means that B was executed after A finished execution while a dotted line from C to D means that D is executed (or called) in the context of C (i.e., C is still "active").

The graphs shown in Figs. 5 and 6 are created with the `Graph::Easy` tool [2], which takes a textual description of the graph and generates the graph in HTML, SVG, or even ASCII format. For graphs that are not overly

complicated the automated layout engine of `Graph::Easy` does a very good job. However, for bigger graphs a full control flow graph can be unwieldy and it is advisable to do a layer-by-layer visualization in this case.

An example of the layer-by-layer visualization is shown in Fig. 6. Here each graph only shows a single layer of the callgraph, i.e., a parent node and all its child nodes. Since the predecessor nodes of each node are only its siblings or the parent node, this view is sufficient to cover the local view of the control flow graph. The horizontal and vertical edges have the same meaning as in the previous case. To indicate which nodes have child nodes, the text box contains a (+) sign. Clicking on such a node brings up the control flow graph of the child nodes to allow an interactive exploration of the CFG.

The example in Fig. 6 is derived from an execution of the CG benchmark of the OpenMP version of the NAS parallel benchmarks [6] (class C) on a 4-way AMD Opteron processor node (1.8 GHz, 3 GB of main memory). The application is automatically instrumented with Opari and the initialization phase and the iteration loop have been additionally instrumented manually.

As shown in Fig. 6a, the application spends 17.8 seconds in the initialization phase and then executes 75 iterations of the main iteration loop with a total of 702.6 seconds of execution time. Fig. 6b shows the control flow of the initialization phase, while Fig. 6c is the control flow of the main iteration loop. The initialization proceeds in a series of parallel constructs and parallel loops[1]. Significant time is only spent in the regions `R00017` and `R00027`.

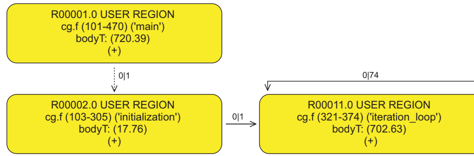
Fig. 6c shows the control flow of the iteration loop. We see a nested loop around the `R00017` parallel region which is executed 1875 times in total and represents by far the most time consuming region. Region `R00017` is called in the initialization as well as in the iteration phase. Drilling down to this parallel region in Fig. 6d, we see that it contains four loops (`R00018`, `R00019`, `R00020`, `R00021`) of which the first one is the most time consuming. The performance data include the waiting time at the end of worksharing regions (`exitBarT`). It is an indicator for load imbalance but does show any severe performance problems in this case.

Note that in Figs. 6a, 6b, and 6c the edges are only executed by the master thread (thread 0). Since the application executes sequentially in the phases outside of parallel regions (only the master thread is active). Only after a parallel region is entered, a thread team (with four threads in this case) is created and several threads show up in the control flow graph as in Fig. 6d.

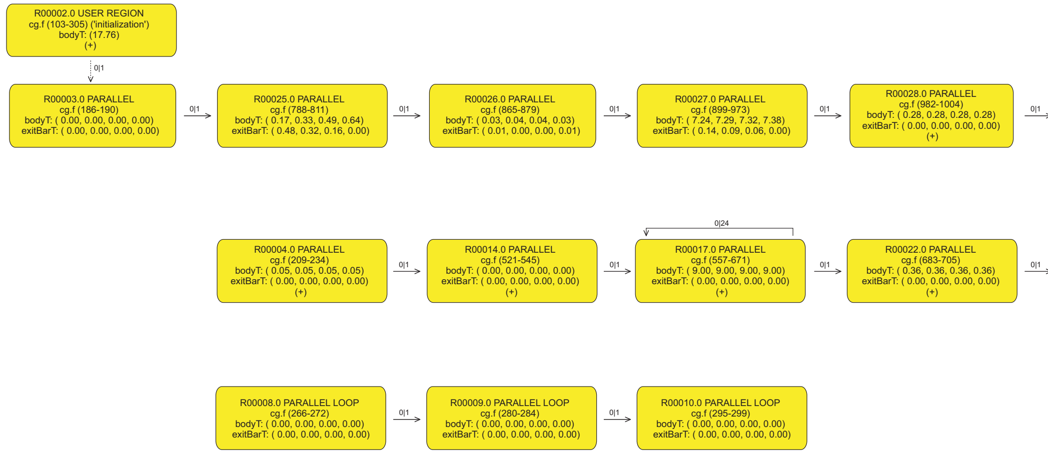
4 Related Work

Control flow graphs are an important topic in the area of code analysis, generation, and optimization. In that context, CFGs are usually constructed based on a compiler's intermediate representation (IR) and are defined as directed multi-graphs with nodes being basic blocks (single entry, single exit) and nodes

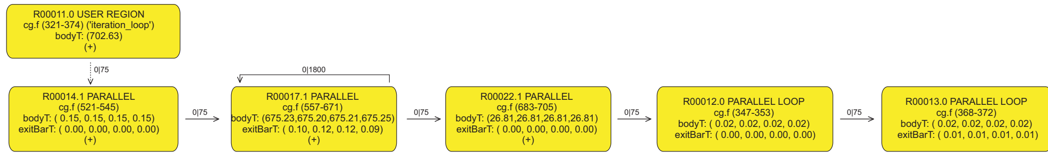
[1] A parallel loop is one of OpenMP's combined parallel-worksharing constructs.



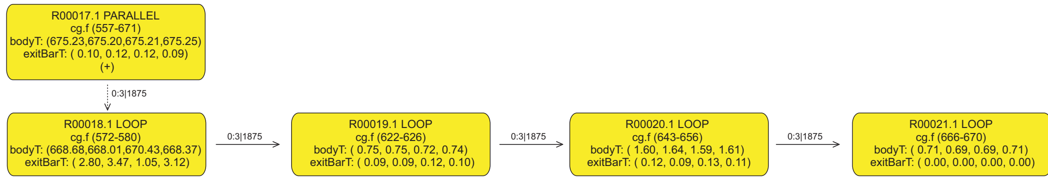
(a) Toplevel control flow.



(b) Control flow of the “initialization” phase.



(c) Control flow of the “iteration phase”.



(d) Most time is spent in the region `R00017`.

Fig. 6: Four layers of the control flow graph of the CG application of the NAS parallel benchmarks (class C).

representing branches that a program execution *may* take (multithreading is hence not directly an issue). The difference to the CFGs in our work is primarily twofold. First, the nodes in our graphs are generally not basic blocks but they are usually larger regions of code containing whole functions. Secondly, the nodes in our graphs record transitions that have actually happened during the execution and also contain a count that shows how often the transition occurred.

Dragon [3] is a performance tool from the OpenUH compiler suite. It can display static as well as dynamic performance data such as the callgraph and control flow graph. The static information is collected from OpenUH's analysis of the source code, while the dynamic information is based on the feedback guided optimization phase of the compiler. In contrast to our approach, the displays are based on the compiler's intermediate representation of source code. The elements of our visualization are the constructs of the user's model of execution to contribute to a high-level understanding of the program execution characteristics.

5 Conclusion

We have presented an approach to visualize the control flow graph of OpenMP applications. We have extended an existing profiling tool to collect the data required for the visualization and used a versatile automated layout tool to generate the graph images.

We believe that the CFG represents valuable information to anyone trying to understand the performance characteristics of an application. Naturally, the author of a code might be very well aware already of their application's control flow and benefit little from the insight `ompP`'s control flow graph can offer. For someone working on a foreign code and especially for big and unfamiliar applications, we believe the CFG view is very helpful to get an understanding of the application's behavior, to understand the observed performance behavior and to identify tuning opportunities.

Future work is planned in several directions. First, `ompP` cannot currently handle nested parallelism but adding support for this is planned for a future release. Visualizing nested parallelism will pose new challenges when displaying the control flow graph as well. Secondly, we plan to develop an integrated viewer for the profiling data delivered by `ompP`, eliminating the need for an external graph layout mechanism. Among other graphical displays such as overhead graphs this viewer will also be able to display the control flow graph. We plan to support both the full CFG display as well as the layered approach in an interactive way, i.e., navigating between the nodes of the control flow graph and call graph and linking this information to the detailed profiling data as well as the source code.

References

1. Shirley Browne, Jack Dongarra, N. Garner, G. Ho, and Philip J. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.
2. The graph::easy web page: `http://search.cpan.org/~tels/Graph-Easy/`.
3. Oscar Hernandez, Chunhua Liao, and Barbara Chapman. Dragon: A static and dynamic tool for OpenMP. In *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, pages 53–66, 2004.
4. Intel Trace Analyzer `http://www.intel.com/software/products/cluster/tanalyzer/`.
5. Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin. An OpenMP runtime API for profiling. Accepted by the OpenMP ARB as an official ARB White Paper available online at `http://www.compunity.org/futures/omp-api.html`.
6. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, 1999.
7. J. Levon. OProfile, A system-wide profiler for Linux systems. Homepage: `http://oprofile.sourceforge.net`.
8. Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.
9. Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–90, 1996.
10. V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and Occam Developments*, volume 44, pages 17–31, Amsterdam, 1995. IOS Press.
11. Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.