Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Recording the control flow of parallel applications to determine iterative and phase-based behavior[☆]

Karl Fürlinger[a,*], Shirley Moore[b]

[a] Computer Science Division, EECS Department, University of California at Berkeley, Soda Hall 593, Berkeley, CA 94720, USA

[b] Innovative Computing Laboratory, EECS Department, University of Tennessee, Claxton Complex, Knoxville, TN 37996, USA

ARTICLE INFO

Article history:
Received 16 October 2008
Received in revised form
5 March 2009
Accepted 13 May 2009
Available online 20 May 2009

Keywords:
Performance tools
Execution control flow visualization and analysis
Performance profiling
Iterative and phase-based behavior of applications

ABSTRACT

Many applications exhibit iterative and phase-based behavior. We present an approach to detect and analyze iteration phases in applications by recording the execution control flow graph of the application and analyzing it for loops that represent iterations. Phases are then manually marked and performance profiles are captured in alignment with the iterations. By analyzing how profiles change between capture points the differences in execution behavior between iterations can be highlighted.

Published by Elsevier B.V.

1. Introduction

Many applications exhibit iterative and phase-based behavior. Typical examples are the time steps in a simulation and iteration until convergence in solvers of linear systems of equations. With respect to performance analysis phase knowledge can be exploited in several ways. First, repetitive phases offer the opportunity to restrict data collection to a representative subset of program execution. This is especially beneficial when tracing is used due to the large amounts of performance data and the challenges involved with capturing, storing, and analyzing it. Conversely, it can be interesting to see how the iterations differ and change over time to expose effects such as cache pollution, operating system jitter and other sources that can cause fluctuations in the execution time of otherwise similar iterations.

In this paper we present an approach to detection and analysis of phases in threaded scientific applications. Our approach assists in the detection of the phases based on the control flow graph (CFG) of the application if the developer is not already familiar with the structure of the code. To analyze phase-based performance data we modified an existing profiling tool for OpenMP applications. Based on markups in the code that denote the start and end of phases, the profiling data is dumped into a file during the execution of the application (and not only at the end of the program run) and can thus be correlated to the application phases.

The rest of this paper is organized as follows. Section 2 describes the technique we used to assist the developer in detecting iterative application phases. In Section 3 we describe the analysis of performance data based on phases using the existing profiling tool called ompP. In Section 4 we describe an example of applying our technique to a benchmark application and in Section 5 we describe related work. We conclude the paper in Section 6.

2. Iterative phase detection

Our approach to identify iterative phases in threaded applications is based on the monitoring and analysis of the control flow graph of the application. For this, we extended our profiling tool ompP.

ompP [1] is a profiling tool for OpenMP applications that supports the instrumentation and analysis of OpenMP constructs. For sequential and MPI applications it can also be used for profiling on the function level, and the phase detection described here is similarly applicable. ompP keeps profiling data and records a callgraph of an application on a per-thread basis and reports the (merged) callgraph in the profiling report.

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

* Corresponding author. Tel.: +1 510 984 2526.
E-mail addresses: fuerling@eecs.berkeley.edu (K. Fürlinger), shirley@eecs.utk.edu (S. Moore).

0167-739X/$ – see front matter. Published by Elsevier B.V.
doi:10.1016/j.future.2009.05.008

```
P() {                           A() {         C() {
  for(i=1; i<5; i++ ) {           X();          Z();
    A();                          Y();        }
    B();                        }
    C();
  }
}

          predecessor                     predecessor
          list                            list
 P                               P
  +-A     (P:1)                   +-A     (P:1,C:4)
  |  +-X  (A:1)                   |  +-X  (A:5)
  |  +-Y  (X:1)                   |  +-Y  (X:5)
  +-B     (A:1)                   +-B     (A:5)
  +-C     (B:1)                   +-C     (B:5)
     +-Z  (C:1)                      +-Z  (C:5)
```

Fig. 1. Illustration of the data collection process to reconstruct the control flow graph.

Unfortunately, the callgraph of an application (recording the caller–callee relationships and also the nesting of OpenMP regions) does not contain enough information to reconstruct the control flow graph. However, a full trace of function execution is not necessary either. It is sufficient that for each callgraph node a record is kept that lists all predecessor nodes and how often the predecessors have been executed. 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. 1. The callgraph (lower part of Fig. 1) shows all possible predecessor nodes of node *A* in the CFG (control flow graph). They are the siblings *B* and *C*, and the parent node *P*. The numbers next to the nodes in Fig. 1 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 a mechanism to record the execution control flow according to this scheme in ompP was straightforward. ompP already keeps a pointer to the *current* node of the callgraph (for each thread) and this scheme 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 previous 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.

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 graph layout mechanisms. An example of an application's full control flow is shown in Fig. 2 along with the corresponding (pseudo-) source code.

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 to be 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 executed that edge 5 times. This greatly reduces the complexity of the control flow graph and makes it easier to understand.

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").

Based on the control flow graph, the user has to manually mark the start and end of iterative phases. To allow a user to perform this markup we extended the OpenMP instrumenter OPARI [2]. OPARI (OpenMP Pragma And Region instrumenter) performs an automated instrumentation of OpenMP constructs and allows the user to specify source code regions of interest using pragmas in C/C++ and comments in FORTRAN. To mark the start of the phase the user adds a directive in the form `phase start`, and to mark the end, `phase end`.

Automating the process of marking the phase boundaries is an area for future work. This automation step will require graph algorithms to analyze the generated flow graphs and potentially the usage of heuristics to determine the iterations in cases with more complex flow graphs. As of now, our approach assists the user in determining and analyzing the iterative structure of their code.

3. Iterative phase analysis

The phase-based performance data analysis implemented in ompP captures profiling snapshots that are aligned with the start and end of program phases and iterations. Instead of dumping a profiling report only at the end of the program execution, the reports are aligned with the phases and the change between capture points can be correlated to the activity in the phase. This technique is a modification of the incremental profiling approach

```
main(){
 init();
 for(i=0;i<10; i++) {
  #pragma omp parallel
  {
    #pragma omp critical
    {
      /* do something */
    }
  }
 }
 fini();
}
```

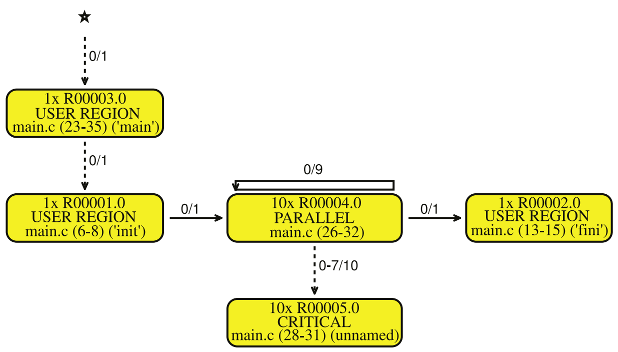

Fig. 2. An example for a full control flow display of an application.

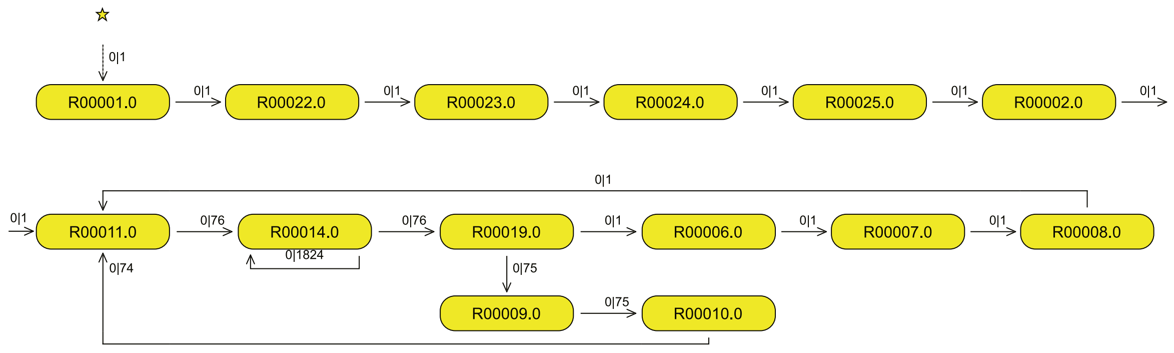

Fig. 3. (Partial) control flow graph of the CG application (size C).

described in [3] where profiles are captured in regular intervals such as 1 s.

The following performance data items can be extracted from phase-aligned profiles and displayed to the user in the form of 2D graphs.

Overheads ompP classifies wait states in the execution of the OpenMP application into four overhead classes: synchronization, limited parallelism, thread management and work imbalance. Instead of reporting overall, aggregated overhead statistics, ompP's phase analysis allows the correlation of overheads that occur in each iteration. This type of data can be displayed as 2D graphs, where the x-axis correlates to execution time and the y-axis displays overheads in terms of the percentage of execution time lost. The overheads can be displayed both for the whole application and for each parallel region separately. An example is shown in Fig. 5.

Execution Time The amount of time a program spends executing a certain function or OpenMP construct can be displayed over time. Again, this display shows line graphs where the x-axis represents (wall clock) execution time of the whole application while the y-axis shows the execution time of a particular function or construct. In most cases it is most useful to plot the execution time sum over all threads, while it is also possible to plot a particular thread's time, the minimum, maximum or average of times.

Execution Count Similarly to the execution time display, this view shows when certain constructs or functions got executed, but instead of showing the execution time spent, the number of invocations or executions is displayed in this case.

Hardware Counters ompP is able to capture hardware performance counters through PAPI [4]. Users select a counter they want to measure and ompP records this counter on a per-thread and per-region basis. Hardware counter data can best be visualized in the form of heatmaps, where the x-axis displays the time and the y-axis corresponds to the thread ID. Tiles display the normalized counter values with a color gradient or gray scale coding. An example is shown in Fig. 5.

4. Example

In this example we apply the phase detection and analysis technique to a benchmark code from the OpenMP version (3.2) of the NAS parallel benchmark suite. All experiments have been conducted on a four processor AMD Opteron SMP system. The application we chose to demonstrate our technique is the CG application which implements the conjugate gradient technique.

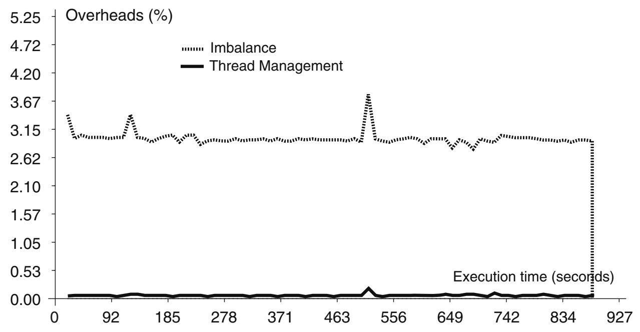

Fig. 4. Overheads of the iterations of the CG application. The x-axis is the wallclock execution time in seconds, while the y-axis represents the percentage of execution time lost due to overheads.

The CG code performs several iterations of an inverse power method to find the smallest eigenvalue of a sparse, symmetric, positive definite matrix. For each iteration a linear system $Ax = y$ is solved with the conjugate gradient method.

Fig. 3 shows the control flow graph of the CG application (size C). To save space, only the region identification numbers Rxxxx are shown in this example; in reality, the control flow nodes show important information about the region such as region type, location (file name and line number) and execution statistics in addition. Evidently the control flow graph shows an iteration that is executed 76 times where each iteration takes a path different from the others. This is the outer iteration of the conjugate gradient solver which is executed 75 times in the main iteration and once for initialization.

Using this information (and the region location information) it is easy to identify the iterative phase in the source code. We marked the start of each iteration with a `phase start` directive and each end with a `phase end` directive. Using directives (compiler pragmas in C/C++ and special style comments in FORTRAN) has the advantage that the normal building process is not interrupted. The directives are translated into calls that cause ompP to capture profiles when performance analysis is done, and ompP's compiler wrapper script translates the directives into calls implemented by ompP's monitoring library.

Fig. 4 shows the overheads over time display for the iterations of the CG application with problem size C. Evidently, the only significant overhead identified by ompP is imbalance overhead and the overhead does not change much from iteration to iteration with the exception of two peaks. The most likely reason for these two peaks is operating system jitter, since the iterations are otherwise identical in this example.

Fig. 5 shows the heatmap display of the CG application with four threads. The measured counter is `PAPI_FP_OPS`. In order to visually compare values, absolute values are converted into rates. The first column of tiles corresponds to the initialization part of

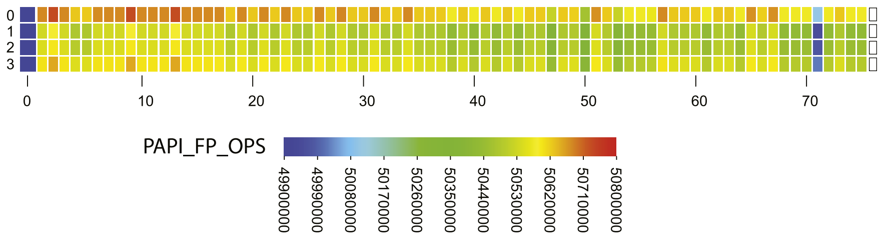

Fig. 5. Performance counter heatmap of the CG application. The horizontal axis represents execution time. There is a tile for each phase or iteration and the labels correspond to the iteration number. The vertical axis corresponds to the thread ID. The lower part of the figure is the legend showing which color corresponds to which hardware counter intensity value.

the code which features a relatively small number of floating point operations; the other iterations are of about equal size but show some difference in the floating point rate of execution.

The heatmap display is available for any counter that can be measured with PAPI. The images are created by post-processing ompP's profiling reports with a perl script that generates SVG (scalable vector graphics) images that can be opened with any modern web browser. Depending on the selected hardware counters, this display offers a very interesting insight into the behavior of the applications. Phenomena that we were able to identify with these performance displays in previous work [3] include:

- The homogeneity or heterogeneity of thread behavior. For example, in 32-thread runs, threads 16, 8, and 24 would frequently show markedly different behavior compared to other threads for many of the applications of the SPEC OpenMP suite we studied. We identified several possible reasons for this difference in behavior, either coming from the application itself (related to the algorithm) or from the machine organization and system software layer (mapping of threads to processors and their arrangement in the machine and its interconnect).
- Identification of temporary performance bottlenecks such as short-term bus-contention.

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

Detection of phases in parallel programs has previously been applied primarily in the context of message passing applications. The approach of Casas-Guix et al. [5] works by analyzing the autocorrelation of message passing activity in the application, while our approach works directly by analyzing the control flow graph of the application.

The work of Preissl et al. [6] tries to detect recurring patterns of communication events for optimization purposes. Events are recorded as an array of 32-bit integers (tracing is performed) and repeating sequences of events are searched for by either a convolution or suffix-tree-based method. The identified and matched repeating sequences, together with source code analysis using Rose [7], are the basis for source code transformations such as replacing a series of point to point operations with the corresponding collective. Compared to their method, our technique avoids the overhead of generating, storing, and analyzing traces. Instead our technique of recording the execution control flow directly exposes repetitive structures as loops in the flow graphs.

Earlier work by Knuepfer et al. [8,9] has focused on exploiting repeating patterns in traces to reduce trace size. The resulting data structures are called compressed complete callgraphs and they can be used for lossy storage of traces in trace analysis tools. In contrast, our approach never stores full traces but directly derives the flow information from profiling. The earlier work of Kranzlmüller [10,11] was primarily concerned with the detection of repeating patterns in recorded traces for debugging purposes.

Szebenyi et al. [12] investigate the merits of iteration phase based performance analysis for the MPI-parallel tree code PEPC (Pretty Efficient Parallel Coulomb-solver) using the Scalasca [13] toolset. Between iterations they find gradually varying performance characteristics (such as the computational load of individual ranks) that accumulate to big load imbalances over time. Their case study highlights the benefits of phase-based performance analysis because the effects visible in individual iterations would be hard to discern based on performance data for the whole program execution.

6. Conclusion

We have presented an approach for detecting and analyzing the iterative and phase-based behavior in threaded applications. The approach works by recording the control flow graph of the application and analyzing it for loops that represent iterations. This help of the control flow graph is necessary and useful if the person optimizing the code is not the code developer and does not have intimate knowledge.

With identified phase boundaries, the user marks the start and end of phases using directives. We have extended a profiling tool to support the capturing of profiles aligned with phases. In analyzing how profiles change between capture points, differences in execution behavior between iterations can be uncovered.

References

[1] Karl Fürlinger, Michael Gerndt, ompP: A profiling tool for OpenMP, in: Proceedings of the First International Workshop on OpenMP, IWOMP 2005, Eugene, Oregon, USA, May 2005.
[2] Bernd Mohr, Allen D. Malony, Sameer S. Shende, 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.
[3] Karl Fürlinger, Jack Dongarra, On using incremental profiling for the performance analysis of shared memory parallel applications, in: Proceedings of the 13th International Euro-Par Conference on Parallel Processing, Euro-Par '07, August 2007 (in press).
[4] Shirley Browne, Jack Dongarra, N. Garner, G. Ho, Philip J. Mucci, A portable programming interface for performance evaluation on modern processors, Int. J. High Perform. Comput. Appl. 14 (3) (2000) 189–204.
[5] Marc Casas-Guix, Rosa M. Badia, Jesus Labarta, Automatic phase detection of MPI applications, in: Proceedings of the 14th Conference on Parallel Comput., ParCo 2007, Aachen and Juelich, Germany, 2007.
[6] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, Daniel J. Quinlan, Using MPI communication patterns to guide source code transformations, in: ICCS '08: Proceedings of the 8th International Conference on Computational Science, Part III, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 253–260.

[7] Daniel J. Quinlan, ROSE: Compiler support for object-oriented frameworks, Parallel Process. Lett. 10 (2/3) (2000) 215–226.

[8] Andreas Knüpfer, Wolfgang E. Nagel, New algorithms for performance trace analysis based on compressed complete call graphs, in: International Conference on Computational Science (2), 2005, pp. 116–123.

[9] Andreas Knüpfer, Wolfgang E. Nagel, Construction and compression of complete call graphs for post-mortem program trace analysis, in: Proceedings of the 2005 International Conference on Parallel Processing, ICPP-05, Oslo, Norway, June 2005, pp. 165–172.

[10] Dieter Kranzlmüller, Communication pattern analysis in parallel and distributed programs, in: Proceedings of the 20th IASTED International Multi-Conference Applied Informatics, AI 2000, ACTA Press, 2002, pp. 153–158.

[11] Dieter Kranzlmüller, Event Graph Analysis for Debugging Massively Parallel Programs, Ph.D. Thesis, GUP Linz, Joh. Kepler University Linz, Altenbergerstr. 69, A-4040 Linz, Austria, September 2000.

[12] Zoltan Szebenyi, Brian J.N. Wylie, Felix Wolf, Scalasca parallel performance analyses of PEPC, in: Proceedings of the Workshop on Productivity and Performance, PROPER 2008 at EuroPar 2008, Las Palmas de Gran Canaria, Spain, 2006.

[13] Markus Geimer, Felix Wolf, Brian J.N. Wylie, Bernd Mohr, Scalable parallel trace-based performance analysis, in: Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2006, Bonn, Germany, 2006, pp. 303–312.

Karl Fuerlinger is a postdoctoral researcher at the University of California at Berkeley. He earned his Ph.D. at the University of Technology in Munich and was a Senior Research Associate at the Innovative Computing Laboratory (ICL) in Knoxville TN before joining UC Berkeley. His research interests include parallel computing and machine learning.

Shirley Moore is Associate Director of Research at the Innovative Computing Laboratory (ICL) in the Electrical Engineering and Computer Science Department at the University of Tennessee. Her research interests are in performance analysis and optimization of parallel software, software reuse in high performance computing, and distributed databases.