Detection and Analysis of Iterative Behavior in Parallel Applications

Karl Fürlinger and Shirley Moore

Innovative Computing Laboratory,
Department of Electrical Engineering and Computer Science,
University of Tennessee
{karl, shirley}@eecs.utk.edu

Abstract. Many applications exhibit iterative and phase based behavior. We present an approach to detect and analyze iteration phases in applications by recording the 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, differences in execution behavior between iterations can be uncovered.
Key words: Phase detection, control flow graph, continuous profiling

1 Introduction

Many applications exhibit iterative and phase based behavior. Typical examples are the time steps in a simulation and iteration until convergence in a linear solver. 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 execution time of otherwise identical 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 of the application if the developer is not already familiar with the code's structure. 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 to a file during the execution of the application (and not only at the end of the program run) and can 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 Sect. 3 we describe the analysis of performance data based on phases using the existing profiling tool called `ompP`. In Sect. 4 we describe an example of applying our technique to a benchmark applications, in Sect. 5 we describe related work and conclude in Sect. 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 call graph of an application on a per-thread basis and reports the (merged) callgraph in the profiling report.

Unfortunately, the callgraph of an application (recording 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. 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.

```
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.

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 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.

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. 2 along with the corresponding (pseudo-) source code.

```
main(){
  init();
  for(...) {
    #pragma omp parallel
  }
  fini();
}
```

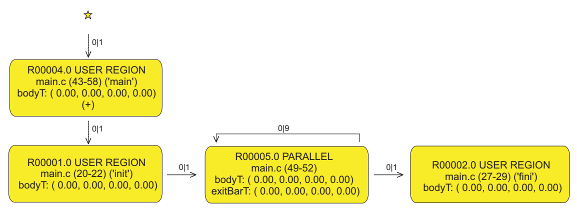

Fig. 2. An example for a full control flow display of an application.

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 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.

Based on the control flow graph, the user has to manually mark the start and end of iterative phases. To mark the start of the phase the user adds the directive `phase start`, to mark the end `phase end`.

3 Iterative Phase Analysis

The phase based performance data analysis implemented in `ompP` works by capturing profiling snapshots that are aligned with the start and end of program phases. 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 described in [2] where profiles are captured in regular intervals such as 1 second.

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 two-dimensional graphs, where the x-axis correlates to execution time and the y-axis displays overheads in terms of percentage of execution time lost. The overheads can be displayed both for the whole application or for each parallel region separately. An example is show 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 Similar 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 [3]. Users selects 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 show in Fig. 4.

4 Example

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

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.

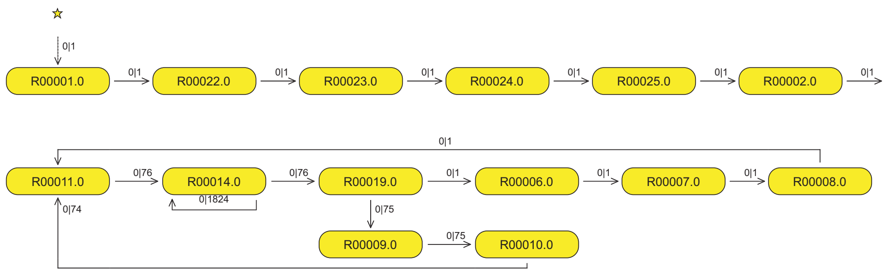

Fig. 3. (Partial) control flow graph of the CG application (size C).

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 one iteration takes another path than 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 and end of each iteration with an `phase start` directive and each end the ends with a `phase end` directive. Using directives (compiler pragmas in C/C++ and special style comments in FORTRAN) similar to OpenMP directives has the advantage that the normal (non-performance analysis) build process is not interrupted while 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 of 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

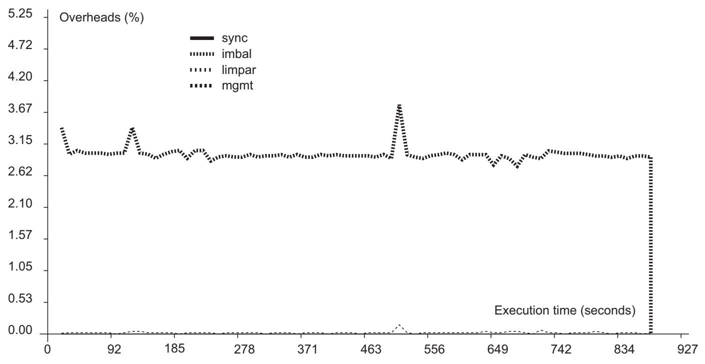

Fig. 4. Overheads of the iterations of the CG application. X-axis is wallclock execution time in seconds, while th y-axis represents the percentage of execution time lost due to overheads.

likely reason for these two peaks is operating system jitter, since the iterations are otherwise identical in this example.

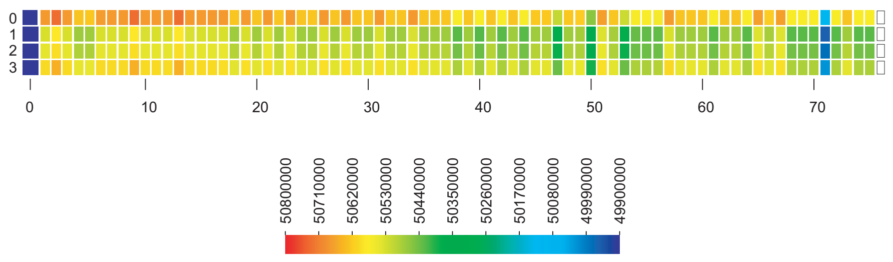

Fig. 5. Performance counter heatmap of the CG application. X-axis is phase or iteration number, the y-axis corresponds to the thread ID.

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 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 floating point rate of execution.

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 to 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. [4] 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.

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. Fürlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. In: Proceedings of the First International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA (2005)
2. Fürlinger, K., Dongarra, J.: 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). (2007) accepted for publication.
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.J.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14** (2000) 189–204
4. Casas-Guix, M., Badia, R.M., Labarta, J.: Automatic phase detection of MPI applications. In: Proceedings of the 14th Conference on Parallel Computing (ParCo 2007), Aachen and Juelich, Germany (2007)