Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning

Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata,
and Michael Bedford Taylor
Department of Computer Science & Engineering
University of California, San Diego
http://parallel.ucsd.edu/pyrprof

Abstract

Multicore processors have forced mainstream programmers to rethink the way they design software. Parallelism will be the avenue for performance gains in these multicore processors but will require new tools and methodologies to gain full acceptance by everyday programmers. As a step towards improved parallelization tools, we propose a parallelization taxonomy that categorizes tools based on which of five fundamental stages of parallelization they assist with. Based on this taxonomy, we find that many popular parallelization tools focus on the final stages, leaving the programmer to perform the initial stages without assistance. In this paper we provide a preliminary description of `pyrprof`, a tool that helps the programmer locate parallel regions of code and decide which regions to parallelize first. `pyrprof` performs dynamic critical path analysis and utilizes the structure of programs to highlight exploitable forms of parallelism. A case study based on MPEG encoding is used to demonstrate `pyrprof`'s effectiveness.

1 Introduction

With the promise of mainstream multiprocessors comes the formidable challenge of programming them. Programming these multicore processors will require the programmer to perform the notoriously difficult task of exploiting parallelism. These parallelism requirements as well as the ubiquity of multicore processors have forced parallelization into the consciousness of the mainstream programming community. To address the rising prevalence of parallel programming, many tools have been developed to aid the programmer in this difficult task. Unfortunately, these tools do not form a complete flow that addresses all parts of the parallelization task. This absence of tools for several key parallelization stages has created a "parallelization gap" that the programmer must bridge on their own. This gap hinders programmers' effectiveness in exploiting the parallel resources of multicore processors and threatens to limit the potential of these chips.

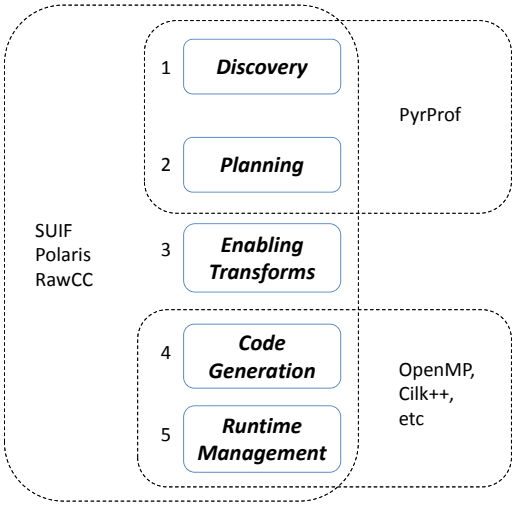

Figure 1: **A Taxonomy of Parallelization** The taxonomy categorizes parallelization tools based on which of five fundamental parallelization stages they assist with. Automatically parallelizing compilers like Polaris [1] and SUIF [3] attempt to perform all five without programmer assistance, while tools like OpenMP, Cilk++ [6], and X10 [8] focus on the last two. The paper's tool, `pyrprof`, targets the first two stages.

The parallelization gap is partially a result of the lack of a well-established terminology to describe the stages involved in parallelization. As shown in Figure 1, we have developed a taxonomy for common parallelization stages. Parallelization begins with *Parallelism Discovery*, which is the process of identifying regions of a program that have exploitable parallelism. Locating these regions of the program is especially onerous for large, complex programs or when —as is often the case—the parallel programmer is not the original author of the pro-

gram. The next stage, *Parallelism Planning*, determines which subset of these parallel regions should be parallelized. Ideally, this plan would consider many factors including the parallel resources available and the runtime environment's ability to effectively utilize each type of parallelism. The third step is *Enabling Transforms*. These are source-level transformations that the user performs in order to enhance the parallelizability of the code used in subsequent stages. The parallelization process concludes with the final two steps of *Parallel Code Generation* and *Runtime Management*. During these two stages, parallel/threaded code is generated and runtime systems provide an environment that allows for parallel execution of the program.

As shown in Figure 1, automatic parallelizing compilers such as Polaris [1], [5], and SUIF [3] automate all of the stages in the parallelization taxonomy. Although they eliminate the need for manual intervention, the performance of code generated by these compilers often pales in comparison with code generated manually. This lack of performance stems from the difficulty of many of these steps; the compiler cannot effectively perform them without the benefit of runtime information or without otherwise unsound semantic changes by the user. While additional information could potentially improve their performance, early parallelizing compilers did not have each mechanism for the user to influence their behavior. Without this additional input, these compilers had an all-or-nothing feel: the compiler either did all of the work or none of it.

An alternative to the fully-automated approach is to separate the parallelization process into the stages shown in Figure 1 and make use of parallel programming tools to automate as much of the work as possible. These tools would relieve most of the burden on the programmer and allow them to focus on the *Enabling Transforms* stage where automated tools are most limited. Recent tools such OpenMP, Cilk++ [6], and X10 [8] are examples of how user-centered tools can improve the productivity of parallel programmers. These tools provide extensions to standardized languages that allow a programmer to explicitly specify parallel regions and synchronization points and automatically handle the final two stages of the parallelization process. By the time a programmer can utilize tools such as OpenMP and Cilk++, they must have already performed the first three stages of parallelization without assistance. We refer to this lack of assistance as the parallelization gap.

To help the programmer bridge the parallelization gap, we have developed `pyrprof`, which performs both *Parallelism Discovery* and *Parallelism Planning*. `pyrprof` profiles a sequential program to determine the amount of parallelism in each region, typically a loop or function, of a program. We call the tool `pyrprof` because it analyzes the program at every level of region nesting, similar to multi-scale pyramid representations in image processing. This is an important feature for choosing the right granularity at which to exploit parallelism. A visual representation of `pyrprof`'s discovery stage can also provide insight during parallelization.

To help determine what regions to parallelize first, `pyrprof`'s planning phase takes the output of the parallelism discovery phase and creates a sequence in which the regions should be parallelized. It is designed to be as easy to use as `gprof` but with additional capabilities needed by the parallel programmer. If the user wishes to deviate from the presented plan, `pyrprof` offers an interface that allows the programmer to specify regions which they cannot or will not parallelize. This input is used by `pyrprof` to revise its recommendations.

The rest of the paper is structured as follows. We start by providing a case study of `pyrprof` based on an MPEG encoding program in Section 2. In Section 3 we discuss the implementation of `pyrprof` including its two main phases: discovery and planning. Finally, we summarize our work in Section 4.

2 Case Study: MPEG encoder

Before we describe how `pyrprof` is implemented, we will motivate the need for such a tool as well as demonstrate its capabilities through the use of a case study. For this case study, we looked at the MPEG encoder applications from the ALP benchmark suite [9]. This program provides ample opportunities for parallelization and includes a reference parallelized implementation which employed pthreads.

To create a simple yet effective tool for discovering and planning for parallelism, we took inspiration from `gprof` [2]. `gprof` is a profiling tool that presents the user with a list of functions ordered by the amount of time the program spent executing them. A programmer typically visits functions in order down this list, looking for optimization opportunities. Implicitly, `gprof`'s ordering addresses Amdahl's Law-style limitations by focusing the programmer on the regions of the program that apply to the largest percentage of the program's dynamic execution. Typically, the user stops when potential speedups become overly limited by Amdahl's Law.

While both `pyrprof` and `gprof` seek to provide an ordering based on potential for optimization, only `pyrprof` takes into account both the time spent in each region and the potential speedup benefits from that region's inherent parallelism. These benefits require not only an assessment of the amount of available parallelism, but also an evaluation of the effect of parallelization of a region on overall execution time. These effects can be quite subtle, because the performance benefit de-

```
$> make CC=pyrprof-cc
$> ./mpeg_enc -i input.data -o output.mpg
$> pyrprof mpeg_enc --exclude=exclude.txt -n 7

The following regions will be excluded from recommendations:  D, E

           GNTES
   ID  Cum.   Incr.  File        Lines      Function              Type
1  A   3.14   3.14   motion.c    208 - 220  ptmotion_estimation   loop
2  B   4.40   1.40   motion.c    211 - 220  ptmotion_estimation   loop
3  G   5.50   1.25   transfrm.c  176 - 233  pttransform           loop
4  H   7.17   1.30   transfrm.c  249 - 305  ptitransform          loop
5  C   9.60   1.34   putpic.c    376 - 612  ptputpict             loop
6  F   13.50  1.41   quantize.c  105 - 137  ptquant               loop

Parallelize these regions in the exact order shown.  If you decide not
to parallelize a region, add it to the exclude list and rerun pyrprof.
```

Figure 2: Output of `pyrprof`. `pyrprof-cc` transparently inserts instrumentation code into the program. After running the program, `pyrprof` uses the profiling results to produce ordered recommendations, with the option to exclude specific regions that the user has decided not to parallelize. `pyrprof` includes an approximate upper bound on expected speedup (GNTES).

Region	`gprof`	`pyrprof` initial	`pyrprof` interactive
A	7	1	1
B	8	3	2
G	27	41	3
H	31	29	4
C	35	35	5
F	42	13	6
D	3	4	exclude
E	5	2	exclude

Table 1: For each region in the original serial version of MPEG encode that improved program performance after parallelization, the corresponding rank given by `gprof`, non-interactive `pyrprof`, and interactive `pyrprof`. `gprof` was extended with region support. Region B was not parallelized by the ALP reference parallel implementation; based on `pyrprof`'s results, we were able to improve the performance.

pends non-linearly on what future and previous regions in the program have been or will be parallelized. For example, only parts of the program that are on the critical path will affect execution time when parallelized. `gprof` does not capture either of these accurately in its relative execution time-based ordering. `pyrprof` takes both of these into account by modeling the execution time of the program using information such as the parallelism in each region, the structure of the program, and the parallelization status of each region. Section 3.2 discusses these factors in more detail.

Figure 2 shows the output of running `pyrprof` on the original serial version of the MPEG encoder benchmark and displays the steps required to use `pyrprof`. `pyrprof` integrates seemlessly into current compilation frameworks by providing a drop-in replacement for `gcc`: `pyrprof-cc`. `pyrprof-cc` inserts all the necessary instrumentation code into the program so `pyrprof`'s parallelism analysis is performed during the normal execution of the program. A special file is created after running the compiled program which `pyrprof` uses to provide an ordered set of recommendations to the user. The user will examine these recommendations, in order, and if they are unable to effectively exploit the parallelism in a particular region, they should add it to an exclusion list and rerun `pyrprof` so that it can adjust the plan. `pyrprof`'s output provides not only basic region information (source file, line numbers, and type) but also the somewhat facetiously named *Guaranteed Not To Exceed Speedup*, or GNTES. The GNTES provides an approximate upper bound on the speedup that can be obtained without speculation or dependence-altering transformations. It comes in a cumulative form, including all regions above it, and an incremental form, indicating the marginal benefit of that region. The GNTES is calculated based on `pyrprof`'s region time estimation model. Our experience suggests that GNTES is well correlated with speedup across a diverse collection of multicore architectures and programmer skill levels.

Table 1 provides a comparison of `pyrprof` with

gprof. The first six regions were the ones that resulted in the best parallel execution time for the benchmark. The last two regions are two regions that were recommended by pyrprof but were excluded after examination by the user. The first column shows how gprof ranked these regions. The second column shows the initial ranking of pyrprof, and the third column shows the final output of pyrprof after the user had iterated on the recommendations and excluded two regions. The results show that only one of the parallelized regions in the optimized parallel implementation (region A) appeared in the top ten list of gprof while the others fell between 27 and 42 out of total 189 regions. Interestingly, pyrprof recommended a region, B, not exploited by the original authors of the ALP benchmark, and we were able to get additional speedup. Overall, the numbers show that pyrprof is significantly more effective at prioritizing the important regions than gprof.

2.1 Eyeballing pyrprof's Discovery Stage

While pyrprof provides an order in which regions should be parallelized, the user may wish to understand graphically the structure of parallelism in these and other regions. Like with turn-by-turn directions from Google, the graphical overview can be used to sanity check the step-by-step directions. Figure 3 visualizes pyrprof's analysis of the serial version of the MPEG encoder program. The x-axis charts the serial execution time of the program from start to finish, and is further subdivided into boxes whose left and right sides indicate the start and finish times for each (possibly nested) program region. The relative widths of the boxes can be used to infer their relative importance with respect to the serial execution time (e.g. region A takes much more time than region G, F, or H). The y-axis of this chart plots the ideal parallelism based on pyrprof's analysis.

In addition to providing the user with a sense of the relative importance of the regions in a program, the chart in Figure 3 can quickly reveal the structure of the program and the relationships between regions. For example, Figure 3 shows that region A encompasses 8 instances of region B while region E contains instances of A, C, F, G, and H. In the first case, region A has a higher ideal parallelism than the regions it encloses. This indicates that there is more parallelism to be found by executing these enclosed regions in parallel. In the case of region E, the ideal parallelism is below that of most of its enclosed regions. This is likely because region C is having an Amdahl's Law effect on the speedup of region E.

3 pyrprof Implementation

As described in the introduction, pyrprof addresses the first two stages of the parallelization process: *Parallelism Discovery* and *Parallelism Planning*. During the discovery stage, pyrprof profiles the application to estimate the parallelism in each region. The planning stage then uses these estimates to order the regions based on their potential parallelization benefit. We now examine pyrprof's implementation of these two stages.

3.1 Discovery Stage

pyrprof's discovery stage analyzes every region in the target application to determine its suitability for parallelization. pyrprof currently gathers two important classes of information as part of this task. First, it establishes the amount of work in each region by analyzing the dynamic instruction counts. Second, it estimates the amount of parallelism in each region by analyzing runtime control and data dependences, including those through memory.

pyrprof finds the dependencies and estimates the parallelism available in a region using a dynamic analysis. Using a dynamic analysis instead of a static analysis allows pyrprof's planning stage to make use of runtime information, including control-flow transfers and run-time memory dependences. pyrprof uses LLVM [4] to insert instrumentation directly into the source code before it is compiled. We opt for this approach over a dynamic binary instrumentation infrastructure such as Valgrind [7] for two key reasons. First, it allows us to perform a deeper analysis of the underlying source code that would be hard or otherwise impossible to discover from the binary. Our experience in working with binary instrumentation has suggested that it often requires relatively complex and sometimes approximate heuristics to uncover critical information such as the location of region boundaries, control dependencies, and false dependencies caused by loop induction variables. Source-level instrumentation provides easy access to all of these. Second, using source-level input allows pyrprof to heavily optimize the instrumentation code against the original source code, resulting in better combined performance.

During profiling, pyrprof estimates the parallelism in each region of code based on two factors: the amount of work in the region, and the critical path time of the region. pyrprof's profiling infrastructure determines the earliest time that an instruction will be available based on its data and control dependencies as well as the time required to execute that instruction. pyrprof tracks the latest availability time in each region using a shadow memory infrastructure to determine the earliest possible time that the region can complete. This time is referred to as the critical path time. pyrprof divides the critical path time into the total amount of work in the region to determine the average *ideal parallelism* of the region. We define ideal parallelism as the number of instructions the region can execute in parallel at any given time, on

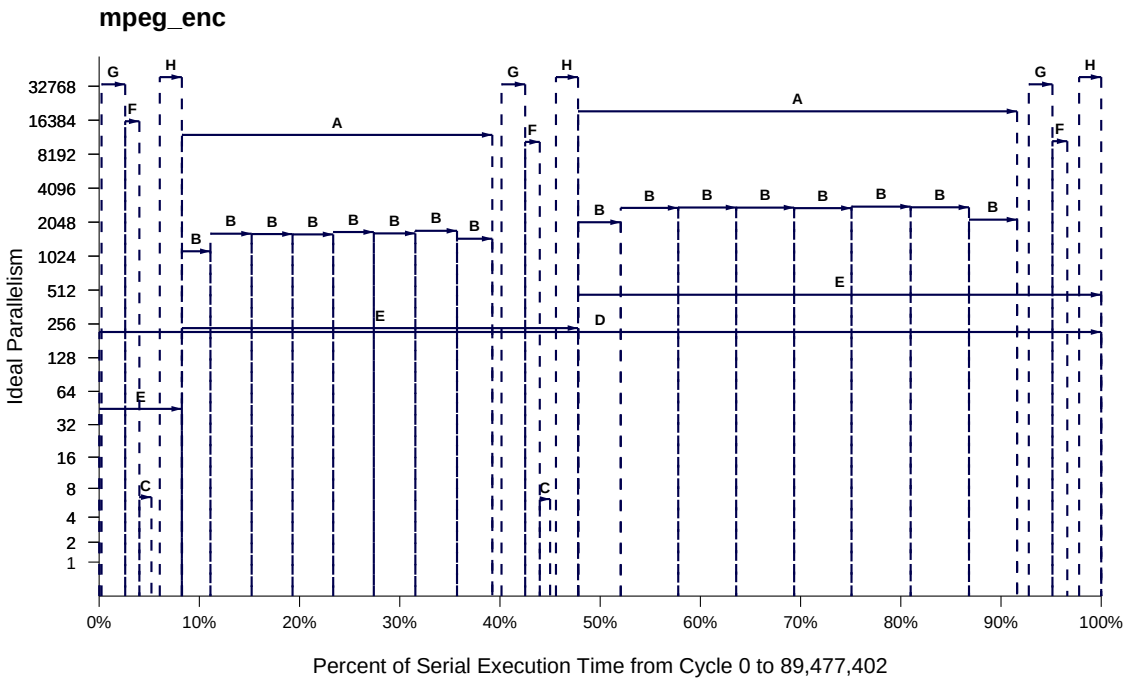

Figure 3: Visualization of `pyrprof`'s parallelism analysis for MPEG encoder benchmark.

an idealized parallel machine with infinite resources and zero-latency communication. For example, parallel regions such as DOALL loops with independent iterations will result in high ideal parallelism while a loop that performs pointer chasing will force this ratio towards one.

3.2 Planning Stage

The essential challenge of the planning stage is to accurately model the execution time of a parallelized program using only the information from the discovery stage. With this model in place, `pyrprof` can quantitatively compare potential parallelization strategies to determine the one with the best performance.

`pyrprof` uses a time estimation model to account for following factors: parallelism, execution time, program structure, and parallelization status. Parallelism and execution time of a region are basic information to calculate the program speedup as per Amdahl's Law. Program structure and parallelization status are also important because the estimation of parent region execution time requires that of children regions. For instance, in a doubly nested loop, if the inner loop is already parallelized, the benefit of outer loop parallelization is likely small because the nested loop's execution time is already reduced from the inner loop parallelization.

Simple greedy algorithms have been proven sufficient in many cases for providing `pyrprof` recommendations that offer good speedups. One challenge, however, is to avoid local minima cases, where speedup may only be realized after parallelizing a collection of regions. To overcome this issue, we have examine the use of a novel backwards search algorithm. This algorithm starts with the assumption that all regions have been parallelized and determines the region that will result in the smallest negative impact when serialized. This process iteratively repeats, each time adding one more region to the non-parallelized set of regions. `pyrprof` reverses this ordering to obtain the proper sequence. This algorithm has two significant advantages over a greedy approach. First, it groups regions that need to be parallelized together in order to achieve speedup: the first one removed will bear all the negative impact with the others free to quickly follow. Second, it avoids local plateaus because it starts with the best performance configuration and makes it way to being completely serial.

4 Conclusion

In this paper we have presented a new taxonomy for classifying the steps involved in parallelizing a program. We have seen that current popular tools focus on later steps. Motivated by the ease-of-use of `gprof`, we have created `pyrprof`, a tool that aims to bridge the parallelization gap by automating the first two stages of parallelization: parallelism discovery and planning. Our results suggest that `pyrprof` is effective at these tasks.

5 Availability

`pyrprof` is available for free download at: `http://parallel.ucsd.edu/pyrprof`.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. "Parallel programming with polaris." *IEEE Computer*, August 2002.

[2] S. L. Graham, P. B. Kessler, and M. K. McKusick. "gprof: a call graph execution profiler." *SIGPLAN Notices*, 2004.

[3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and E. Bu. "Maximizing multiprocessor performance with the suif compiler." *IEEE Computer*, August 1996.

[4] C. Lattner, and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, Mar 2004.

[5] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[6] C. E. Leiserson. "The cilk++ concurrency platform." In *Proceedings of the Design Automation Conference*, 2009.

[7] N. Nethercote, and J. Seward. "Valgrind: A framework for heavyweight dynamic binary instrumentation." *SIGPLAN Notices*, 2007.

[8] V. A. Saraswat, V. Sarkar, and C. von Praun. "X10: concurrent programming for modern architectures." In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[9] R. Sasanka, M. L. Li, S. V. Adve, Y. K. Chen, and E. Debes. "Alp: Efficient support for all levels of parallelism for complex media applications." *ACM Transactions on Architecture and Code Optimization*, 2007.