A Framework for Enabling OpenMP Autotuning

Vinu Sreenivasan[1], Rajath Javali[1], Mary Hall[1](✉), Prasanna Balaprakash[2], Thomas R. W. Scogland[3], and Bronis R. de Supinski[3]

[1] University of Utah, Salt Lake City, UT 84103, USA
mhall@cs.utah.edu
[2] Argonne National Laboratory, Argonne, IL 60439, USA
[3] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

Abstract. This paper describes a lightweight framework that enables autotuning of OpenMP pragmas to ease performance tuning of OpenMP codes across platforms. This paper describes a prototype of the framework and demonstrates its use in identifying best-performing parallel loop schedules and number of threads for five codes from the PolyBench benchmark suite. This process is facilitated by a tool for taking a compact search-space description of pragmas to apply to the loop nest and chooses the best solution using model-based search. This tool offers the potential to achieve performance portability of OpenMP across platforms without burdening the programmer with exploring this search space manually. Performance results show that the tool identifies different selections for schedule and thread count applied to parallel loops across benchmarks, data set sizes and architectures. Performance gain over the baseline with default settings of up to 1.17×, but slowdowns of 0.5× show the importance of preserving default settings. More importantly, this experiment sets the stage for more elaborate experiments to map new OpenMP features such as GPU offloading and the new `loop` pragma.

Keywords: Autotuning · Loop scheduling · Performance portability

1 Introduction

OpenMP is an API which is used to explicitly direct thread-level, shared memory parallelism. By design, OpenMP programmers express parallelism with only modest changes to a sequential code through the addition of pragmas that are used by the compiler to map the code to a parallel platform. As all widely-used

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under the RAPIDS Subcontract Award Number 4000159989.

© Springer Nature Switzerland AG 2019
X. Fan et al. (Eds.): IWOMP 2019, LNCS 11718, pp. 50–60, 2019.
https://doi.org/10.1007/978-3-030-28596-8_4

compilers understand OpenMP pragmas and can generate parallel code, such an approach allows for a single source code that is *portable* across systems.

Achieving high parallel efficiency with OpenMP usually requires *prescriptive* pragmas that explicitly define the program behavior, specifying, for example, parallel schedules and number of threads to use. As pragmas become increasingly prescriptive, the advantage of cross-architecture portability decreases. Descriptive directives pass information about code semantics to the compiler to allow it to optimize without specifying how it might choose to do that. By leaving degrees of freedom in the mapping of OpenMP code, an application code can more readily adapt to different data sets and architectures.

We achieve this goal through the use of autotuning. Autotuning relies on empirical measurement to explore alternative implementations of a computation, and has been used in the HPC community to achieve performance portability across hardware platforms. In this work, we develop a tool we call a *pragma autotuner*, as the alternative implementations it evaluates involve alternative OpenMP pragmas. To manage the large search spaces that arise even with the limited experiment in this paper, our approach incorporates the Search using Random Forests (SuRF) framework, which creates a statistical model of the search space and constrains the time required for empirical measurement [8].

For this paper, we apply the pragma autotuner to the problem of scheduling parallel loops, designated as `#pragma omp parallel for` and equivalent. Even for such a limited experiment, the search space consists of how many threads to use, whether to use static or dynamic scheduling of loop iterations, and the chunk size which selects the granularity of the scheduling. For architectures with large numbers of cores, this search space can be quite large. Moreover, we envision such a tool will be much more necessary as recent features of OpenMP gain wider use, including GPU offload and the prescriptive `loop` construct which leaves the compiler significant freedom in mapping the code.

Related Work and Contribution. Autotuning on high-performance computing has been demonstrated as an important strategy for achieving performance portability across different architectures, starting with BLAS libraries PhiPAC [3] and ATLAS [11], early autotuning compilers [4] and generalizations to other scientific computing motifs [12]. A survey of autotuning for HPC can be found here [2]. The concept of autotuning OpenMP code is well-established and the most prevalent of these employ tuning to go beyond loop schedules, to look at parallel tasks, function inlining, and tuning for energy [5–7,10]. Most closely related to our paper, the work of Liao et al. performed autotuning of OpenMP loop schedules on SMG2000, examining a larger search space and achieving a speedup of more than 5× on 6 threads due to autotuning. However, prior work on autotuning OpenMP requires the use of specialized libraries or specific compilers, and would require more extensive adaptation as new OpenMP constructs are added. In contrast, this paper contributes a general framework that can be used to explore user-directed search spaces of any pragmas, even beyond OpenMP. The centerpiece of this work, a *pragma autotuner*, works with the C preprocessor

to update the pragmas at marked locations in the code. In future work, such an approach could be fully automated using rewrite rules.

2 Search Space for Loop Scheduling

We illustrate the approach with a simple example, the main computation from the `atax` benchmark from PolyBench [9]. This computation has two parallel loops, one for initialization of the output vector, and the other nested loop to compute the result `A*Ax`.

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < _PB_NY; i++)
    y[i] = 0;
  #pragma omp for private(j)
  for (i = 0; i < _PB_NX; i++) {
    tmp[i] = 0;
    for (j = 0; j < _PB_NY; j++)
      tmp[i] = tmp[i] + A[i][j] * x[j];
    for (j = 0; j < _PB_NY; j++)
      y[j] = y[j] + A[i][j] * tmp[i];
  }
}
```

The scheduling of the parallel loops uses default settings for the following three parameters:

- Number of threads to use
- Static vs. dynamic scheduling of loop iterations to threads
- Chunk size, which is the scheduling unit

Figure 1 shows the input to our framework that permits tuning based on these parameters for a 4-core desktop platform with a maximum of 8 threads. We use the Search using Random Forests framework to navigate the search space that arises from this specification.

3 Pragma Autotuner System Design

Figure 2 depicts the organization of the pragma autotuner, used to optimize OpenMP. It needs a configuration file which has the search space definition; for example, the loop scheduling parameters in Fig. 1(b). The original loop scheduling pragmas are replaced with the mapped pragmas. For each replacement pragma in the search space, a separate output OpenMP code file is generated.

```
#pragma omp parallel num_threads(#P2)
{
  #pragma omp for schedule(#P0, #P1)
  for (i = 0; i < _PB_NY; i++)
   y[i] = 0;
  #pragma omp for private (j) schedule(#P0, #P1)
  for (i = 0; i < _PB_NX; i++) {
    tmp[i] = 0;
    for (j = 0; j < _PB_NY; j++)
      tmp[i] = tmp[i] + A[i][j] * x[j];
    for (j = 0; j < _PB_NY; j++)
      y[j] = y[j] + A[i][j] * tmp[i];
   }
 }
```

(a) Code with markers for autotuner.

```
problem.spec_dim(p_id=0, p_space=["static", "dynamic"], default="static")
problem.spec_dim(p_id=1, p_space=[1, 8, 16], default=1) # chunksize
problem.spec_dim(p_id=2, p_space=[1, 2, 4, 8], default=4) # threads
```

(b) Excerpt of parameter specification for `atax` example.

Fig. 1. Modified code to permit pragma autotuning (top) and search space specification (bottom).

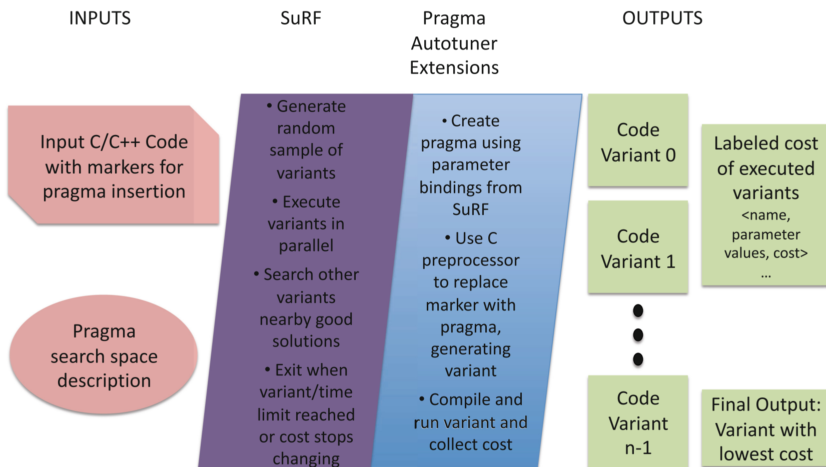


Fig. 2. System design of current pragma autotuner.

The tool also takes in a parameter list which maps different replacement policies to loops marked in the code.

One requirement for code modification of the autotuner is that it requires manual tagging of the beginning of a loop, which is used by the tool to parse the code and generate multiple code files with all combinations of pragmas. The C preprocessor then replaces this mark with the selection of pragmas identified through autotuning search. Then all the output files are executed to record the execution times of the modified loop. Based on the times a suggestion is made by the autotuner software regarding which pragma performs well.

The autotuner has a configuration file through which we can specify the path of the benchmark we want to run. The benchmark source file should have proper markers placed at the corresponding positions where we want to optimize the loops. Later, in the problem definition, we need to define the possible options for those markers using pragmas. We need to pass this problem definition with parameters, their possible values and default values to SuRF, which will return individual points in the search space to examine next.

The parser method in the autotuner then replaces the markers in the source file with the corresponding values received from the search tool and generates a new source file that will be saved in a temporary location in the experiment directory. Later, the generated source files are compiled and run with the options from the configuration file. Once the run has been completed, the execution time will be passed to SuRF as a cost measurement. Based on the execution time, SuRF will return the best combination suitable for the benchmark to run efficiently. To limit the overhead associated with autotuning, the system limits the time of the search, in the case of this paper to 10,000 s.

Sometimes we need an empty string for a parameter to indicate that the default values or no parallelization should be used. Therefore the autotuner supports the empty string parameter value. Whenever the value "None" has been returned from the search tool, the parser will replace it as an empty string in the final code generation.

The generated source is compiled with standard OpenMP compilers; we have tested clang and gcc compilers, and gcc is used in this paper.

4 Experiment

In this section, we describe a simple experiment to demonstrate the capability of the pragma autotuner and its ease of use. We revisit the loop scheduling problem from Sect. 2.

4.1 Methodology

Our goal is to determine via autotuning an optimized schedule (static or dynamic), a chunksize (1, 8 or 16 to coincide with a fully dynamic schedule or a cache line), and number of threads (1, 2, 4 or 8). We execute this experiment on a desktop platform, an Intel CORE i7-4770 with 4 Cores and 8 threads

due to hyperthreading. We apply the system to five benchmarks from PolyBench shown in Table 1. This subset of benchmarks were chosen as representative of 1D, 2D and 3D loop nests, and all have OpenMP parallel for loops without reductions. We used two inputs to test adaptability, Default and Large. For each input, Table 1 provides the settings for Schedule, Chunk and Threads identified by the framework.

We have recently ported the system to a local cluster and are performing multi-node experiments where the evaluations can execute in parallel across nodes. This cluster has dual-socket, 28-core Intel Xeon Broadwell nodes. For this experiment, we show results for just `atax` and, use only the Large dataset, and set the default to 4 threads.

Table 1. PolyBench benchmarks used in this experiment.

Name	Selection (default)			Selection (large)		
	Sched	Chunk	Threads	Sched	Chunk	Threads
atax	dyn	8	4	stat	16	8
3 mm	stat	1	4	stat	1	8
convolution-2d	stat	16	4	stat	16	8
covariance	dyn	8	4	stat	1	8
correlation	dyn	8	4	stat	1	8

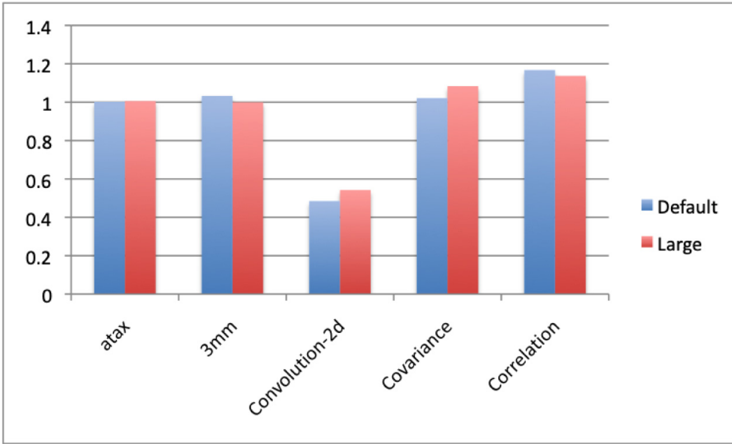
4.2 Performance Results

Figure 3(a) shows the results of the desktop system experiments, speedup over baseline for the five benchmarks and each of the two input data sets. We observe modest speedups for all benchmarks other than `convolution-2d`. The most significant speedups of 1.17× are for the long-running `correlation` benchmark. We believe the slowdown for `convolution-2d` is likely because we are not including the default chunksize in our search space.

Figure 3(b) shows speedups on the cluster system for just the large dataset and benchmark `atax`. As compared to a baseline using 4 threads, a speedup of over 3× is achieved, although as a result of 28 threads.

5 Future Work: From Descriptive to Prescriptive OpenMP

The above simple experiment shows modest performance gains, but we anticipate the true productivity advantage of the pragma autotuner will be to derive pragmas for more complex codes targeting the architectural diversity of current and future systems. This paper describes a work-in-progress as to applications



(a) Desktop system performance results.

Name	Selection (large)			Speedup
	Sched	Chunk	Threads	
atax	static	8	28	3.09×

(b) Cluster system initial performance results.

Fig. 3. Performance improvements over default Baseline schedule.

of the pragma autotuner. In this section, we detail an experiment we are designing to explore a search space for the `#pragma omp loop` that was introduced in OpenMP 5. This construct indicates to the OpenMP compiler that the loop's iterations are independent but leaves it to the discretion of the compiler writer to generate the most appropriate code. In an ongoing experiment, we wish to replace the descriptive `loop` pragma with prescriptive OpenMP pragmas that express how to optimize the loops. For example, we consider the following alternatives:

- A parallel for loop, with the scheduling parameters from the previous section.
- For multi-dimensional loops, we might augment the parallel for loop with a `collapse` clause to assign multiple loop dimensions to a single thread dimension.
- If our target architecture supports efficient `simd` execution, we might want to use the `pragma omp simd` directive.
- If our target architecture supports GPU offload, we might want to map coarse-grain loops to the GPU using the `pragma omp target` directive.

5.1 Case Study: 27-Point Stencil

Since `loop` is a new feature of OpenMP that is not even supported yet by the compilers used in our experiment, there do not currently exist benchmarks that

use this construct. However, we note that a similar descriptive construct in OpenACC is the `#pragma acc independent` pragma. We found an example use of this pragma in the 27 point stencil code from the EPCC OpenACC Benchmark Suite [1]. Figure 4 shows the input code (once converted to use `loop`), and the autotuning search space used for the desktop system in the previous section. The same approach can be used to derive the best mapping of the code.

This more complex experiment has a number of challenges. We plan to explore how to compactly describe the search space, but the example in Fig. 4 illustrates the bulleted items in the above list absent the GPU offload since there is no GPU on our target desktop system.

5.2 Handling Errors

As search spaces become more complex, as in the previous example, SuRF may generate invalid pragma combinations, such as the following example. Here, the middle loop has a `collapse` clause, which has the effect of making the `j` and `k` loops into a single 1D loop. After `collapse` is applied, there is therefore no longer a `k` loop to execute using the `simd` construct. The OpenMP compiler will throw an error when it encounters this kind of combination.

```
for (iter = 0; iter < ITERATIONS; iter++) {
  #pragma omp parallel for
  for (i = 1; i < n+1; i++) {
    #pragma omp for collapse(2)
    for (j = 1; j < n+1; j++) {
      #pragma omp simd
      for (k = 1; k < n+1; k++) {
         <27pt stencil calculation goes here>
}}}}
```

For erroneous configurations, the tool must minimally check the exit code from the compiler and report to SuRF an execution time of `MAX_DBL` so that such configurations are avoided by the search. Ideally, we prefer to build configuration rules into the system to detect errors before generating the code and attempting the compilation. This encoding of OpenMP domain knowledge will increase the complexity of the tool implementation, but reduce the tuning time, and is the subject of future work.

5.3 Automation for Unmodified OpenMP Code

Because OpenMP has a fixed and limited collection of pragmas, we believe it is possible to derive a collection of standard rewrite rules for generating search spaces for pragmas that could automatically be explored for unmodified OpenMP codes. In this way, the user of the tool need not add the markers for the C preprocessor, but rather the tool parses the pragmas in the code to identify rewrite rules that may apply. If possible, this would greatly expand the users for OpenMP autotuning, and is an important area of future work.

```
for (iter = 0; iter < ITERATIONS; iter++) {
  // P0
  #pragma omp loop
  for (i = 1; i < n+1; i++) {
    // P1
    #pragma omp loop
    for (j = 1; j < n+1; j++) {
      // P2
      #pragma omp loop
      for (k = 1; k < n+1; k++) {
      a1[i*sz*sz+j*sz+k] = (
            a0[i*sz*sz+(j-1)*sz+k] + a0[i*sz*sz+(j+1)*sz+k] +
            a0[(i-1)*sz*sz+j*sz+k] + a0[(i+1)*sz*sz+j*sz+k] +
            a0[(i-1)*sz*sz+(j-1)*sz+k] + a0[(i-1)*sz*sz+(j+1)*sz+k] +
            a0[(i+1)*sz*sz+(j-1)*sz+k] + a0[(i+1)*sz*sz+(j+1)*sz+k] +
            a0[i*sz*sz+(j-1)*sz+(k-1)] + a0[i*sz*sz+(j+1)*sz+(k-1)] +
            a0[(i-1)*sz*sz+j*sz+(k-1)] + a0[(i+1)*sz*sz+j*sz+(k-1)] +
            a0[(i-1)*sz*sz+(j-1)*sz+(k-1)] +
            a0[(i-1)*sz*sz+(j+1)*sz+(k-1)] +
            a0[(i+1)*sz*sz+(j-1)*sz+(k-1)] +
            a0[(i+1)*sz*sz+(j+1)*sz+(k-1)] +
            a0[i*sz*sz+(j-1)*sz+(k+1)] + a0[i*sz*sz+(j+1)*sz+(k+1)] +
            a0[(i-1)*sz*sz+j*sz+(k+1)] + a0[(i+1)*sz*sz+j*sz+(k+1)] +
            a0[(i-1)*sz*sz+(j-1)*sz+(k+1)] +
            a0[(i-1)*sz*sz+(j+1)*sz+(k+1)] +
            a0[(i+1)*sz*sz+(j-1)*sz+(k+1)] +
            a0[(i+1)*sz*sz+(j+1)*sz+(k+1)] +
            a0[i*sz*sz+j*sz+(k-1)] + a0[i*sz*sz+j*sz+(k+1)]) * fac;
}}}}
```

(a) 27 point stencil input code.

```
problem.spec_dim(p_id=0, p_space=["None",
        "#pragma omp for schedule(#P3, #P4) nthreads(#P5)",
        "#pragma omp for schedule(#P3, #P4) collapse(#P6) nthreads(#P5)",
       ], default="#pragma omp for schedule(#P3, #P4) nthreads(#P5)")
problem.spec_dim(p_id=1, p_space=["None",
        "#pragma omp for schedule(#P3, #P4) nthreads(#P5)",
        "#pragma omp for schedule(#P3, #P4) collapse(#P6) nthreads(#P5)",
       ], default="#pragma omp for schedule(#P3, #P4) nthreads(#P5)")
problem.spec_dim(p_id=2, p_space=["None",
        "#pragma omp for schedule(#P3, #P4) nthreads(#P5)",
        "#pragma omp simd",
        ], default="#pragma omp simd")
problem.spec_dim(p_id=3, p_space=["static", "dynamic"], default="static")
problem.spec_dim(p_id=4, p_space=[1, 8, 16], default=1)
problem.spec_dim(p_id=5, p_space=[1, 2, 4, 8], default=1)
problem.spec_dim(p_id=6, p_space=[2,3], default=1)
```

(b) Customized search space for this code.

Fig. 4. 27-point stencil code input (top), and associated search space (bottom).

6 Conclusion

This paper has described a pragma autotuner that we have developed to ease the performance portability of OpenMP applications and reduce the programmer's burden of tuning their code as they migrate to the increasingly diverse hardware platforms, and support complex codes. We showed modest gains could be achieved using this system for loop scheduling parameters, and discussed how it could be extended to derive mappings for the new `#pragma omp loop` construct. As OpenMP's capabilities continue to expand to support a diversity of architectures, we believe autotuning will play an increasingly important role in achieving performance portability of current and future OpenMP codes.

References

1. EPCC OpenACC benchmark suite. http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite
2. Balaprakash, P., et al.: Autotuning in high-performance computing applications. Proc. IEEE **106**(11), 2068–2083 (2018). https://doi.org/10.1109/JPROC.2018.2841200
3. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: ACM International Conference on Supercomputing 25th Anniversary Volume, pp. 253–260. ACM, New York (2014). http://doi.acm.org/10.1145/2591635.2667174
4. Chen, C., Chame, J., Hall, M.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: International Symposium on Code Generation and Optimization, pp. 111–122, March 2005. https://doi.org/10.1109/CGO.2005.10
5. Katarzynski, J., Cytowski, M.: Towards autotuning of OpenMP applications on multicore architectures. CoRR abs/1401.4063 (2014). http://arxiv.org/abs/1401.4063
6. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 308–322. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13374-9_21
7. Mustafa, D., Aurangzeb, A., Eigenmann, R.: Performance analysis and tuning of automatically parallelized OpenMP applications. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 151–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_12
8. Nelson, T., et al.: Generating efficient tensor contractions for GPUs. In: 2015 44th International Conference on Parallel Processing, pp. 969–978, September 2015. https://doi.org/10.1109/ICPP.2015.106
9. Pouchet, L.N., Yuki, T.: Polybench/c 4.2. http://sourceforge.net/projects/polybench/
10. Silvano, C., et al.: Autotuning and adaptivity in energy efficient HPC systems: the ANTAREX toolbox. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, pp. 270–275. ACM, New York (2018). https://doi.org/10.1145/3203217.3205338

11. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC 1998, pp. 1–27. IEEE Computer Society, Washington, DC (1998). http://dl.acm.org/citation.cfm?id=509058.509096
12. Williams, S.: Auto-tuning performance on multicore computers. Ph.D. thesis, University of California, Berkeley (2008)