How Many Threads will be too Many? On the Scalability of OpenMP Implementations

Christian Iwainsky[1](✉), Sergei Shudler[2], Alexandru Calotoiu[2], Alexandre Strube[3], Michael Knobloch[3], Christian Bischof[1], and Felix Wolf[1]

[1] Technische Universität Darmstadt, 64293 Darmstadt, Germany
{iwainsky,bischof}@sc.tu-darmstadt.de, wolf@cs.tu-darmstadt.de
[2] German Research School for Simulation Sciences, 52062 Aachen, Germany
{s.shudler,a.calotoiu}@grs-sim.de
[3] Forschungszentrum Jülich, 52425 Jülich, Germany
{a.strube,m.knobloch}@fz-juelich.de

Abstract. Exascale systems will exhibit much higher degrees of parallelism both in terms of the number of nodes and the number of cores per node. OpenMP is a widely used standard for exploiting parallelism on the level of individual nodes. Although successfully used on today's systems, it is unclear how well OpenMP implementations will scale to much higher numbers of threads. In this work, we apply automated performance modeling to examine the scalability of OpenMP constructs across different compilers and platforms. We ran tests on Intel Xeon multi-board, Intel Xeon Phi, and Blue Gene with compilers from GNU, IBM, Intel, and PGI. The resulting models reveal a number of scalability issues in implementations of OpenMP constructs and show unexpected differences between compilers.

Keywords: Performance modeling · OpenMP · Scalability

1 Introduction

In recent years, we saw a clear trend towards systems with more processing cores per node. All types of processors used in high-performance computing, including CPUs, GPUs, or accelerators such as Intel Xeon Phi, are nowadays either multicore or manycore processors. As a result of this trend, the degree of intra-node parallelism in supercomputers is on the rise. Before reaching exascale, it will still have to grow by one or two orders of magnitude [1]. However, this poses the question whether current implementations of multithreaded programming models can scale to the large number of threads this will entail.

In this paper, we try to answer this question for OpenMP, a mature and widely used API for multithreaded programming, and evaluate whether current implementations would scale to much larger numbers of threads. To this end, we adopt the automated performance-modeling method by Calotoiu et al. [2] and generate empirical scaling models of the most common OpenMP constructs.

© Springer-Verlag Berlin Heidelberg 2015
J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 451–463, 2015.
DOI: 10.1007/978-3-662-48096-0_35

The method takes measurements of execution time or other metrics at smaller scales as input and produces human-readable growth functions as output which describe the behavior for larger scales. To capture the cost of individual OpenMP constructs, we extended the EPCC OpenMP micro-benchmark suite [3,4] and combined it with the modeling toolchain. We evaluated OpenMP implementations from GNU, IBM, Intel, and PGI on Xeon, Xeon Phi, and Blue Gene. Our main discoveries are:

– Previously unknown and potentially serious scalability limitations in implementations from GNU, IBM, and PGI
– Different behavioral classes depending on whether the number of threads is a power of two or not

Among all the evaluated compilers, the GNU compiler is the most problematic in terms of scalability.

The next section introduces the model generator we used to create the scaling models and how it was customized for our study. In Sect. 3, we explain the EPCC OpenMP benchmark suite along with our own extensions. Experimental results for selected OpenMP constructs with particularly noteworthy behavior are presented in Sect. 4. Then, we discuss related work in Sect. 5 and draw our conclusion in Sect. 6.

2 Model Generation

The approach underlying our study rests on the identification of *scalability bugs* using automated performance modeling [2]. A scalability bug is a part of a program whose scaling behavior is unintentionally poor, that is, much worse than expected. As computing hardware moves towards exascale, developers need early feedback on the scalability of their software design so that they can adapt it to the requirements of larger problem and machine sizes.

The input of the model generator is a set of performance measurements where only one relevant parameter, in our case the number of threads, is varied while all others are kept constant. The idea is to create functions that describe how a metric, such as the execution time, the number of floating point operations, or the number of bytes injected into the network, changes as the chosen parameter is modified. Depending on the availability of measurements, such models can be created for each function in a program or just one particular code region of interest.

When generating performance models, we exploit the observation that they are usually composed of a finite number n of terms, involving powers and logarithms of the parameter x of interest:

$$f(x) = \sum_{k=1}^{n} c_k \cdot x^{i_k} \cdot log_2^{j_k}(x)$$

This representation is, of course, not exhaustive, but works in most practical scenarios, since it is a consequence of how most computer algorithms are designed. We call it the *performance model normal form* (PMNF).

In this paper, we vary the number of threads t and model the time overhead of OpenMP constructs, i.e., the thread-management time lost in the OpenMP runtime system when executing certain constructs. While changes of the arithmetic intensity may restrict models of user code to specific segments of the domain of t [5], we believe that such effects do not have to be considered when judging the scalability of OpenMP runtime operations. Their critical resource is almost always the latency of memory accesses and, in particular, of cache coherence protocols. Moreover, our experience suggests that neither the sets I, J chosen from the set $\mathbb{Q}$ of rational numbers from which the exponents i_k and j_k are chosen nor the number of terms n have to be arbitrarily large or random to achieve a good fit. A possible assignment of all i_k and j_k in a PMNF expression is called a *model hypothesis*. Trying all hypotheses one by one, we find coefficients c_k with optimal fit. Then we apply cross-validation [6] to select the hypothesis with the best fit across all candidates.

For this study, we selected $n = 2$, $I = \left\{0, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, 1, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{7}{4}, 2\right\}$, and $J = \{0, 1, 2\}$. Our choices for I and J reflect a range of behaviors, from perfect to poor scalability, in 39 shades (13 options for i times 3 for j). In the case of OpenMP constructs, we are not aware of any literature that specifies precise scalability expectations. This is why we operate under the not uncommon assumption that anything significantly worse than logarithmic is unacceptable on the path towards exascale. Given the jitter present in measurements of OpenMP constructs with their minuscule execution times, we only allow one active term plus a constant. Trying to model behaviors past the leading term is likely to capture only noise. Note that we are not trying to create accurate models for OpenMP constructs but rather want to draw the attention to unscalable behavior. Making accurate predictions for the execution times of OpenMP constructs at larger scales is beyond the scope of this work.

3 Benchmark Design

Our goal is to investigate the costs of individual OpenMP constructs for different compilers with a focus on the OpenMP runtime system, disregarding actual workloads. For this purpose, we define time-based metrics that characterize the behavior of OpenMP constructs and that can be further used as an input to the model generator. Because initial experiments indicated a high noise-to-measurement ratio on some of the target platforms, we filter the raw data to reduce noise and remove extreme outliers.

3.1 EPCC OpenMP Micro-Benchmarks

The EPCC OpenMP micro-benchmark suite [3,4] is an established and comprehensive collection of benchmarks that covers almost all OpenMP constructs. The micro-benchmarks compare the cost of the constructs by measuring the difference between a parallelized workload and the workload itself, while the workload per thread is kept constant. Multiple executions (inner repetitions) of a given OpenMP construct scale the cost of the construct for easier measurement and comparison with the reference workload. This inner measurement is again repeated multiple times (outer repetitions) to calculate the average and the standard deviation of the target construct. We modified the EPCC measurement system to directly interface with our model generator.

3.2 Custom Benchmarks

While the EPCC benchmarks are well-designed to capture the overhead of copying data environments, they are less suited to precisely capture synchronization overheads. Since they do not measure the costs of individual OpenMP constructs directly, the resulting timings are much more prone to noise. To measure the costs of OpenMP constructs in isolation, we therefore had to develop additional benchmarks, which are designed as follows: (i) compute local clock offsets between master and all the other threads; (ii) synchronize threads using adjusted window-based mechanism (see next sub-section); (iii) take a per-thread time stamp and call the OpenMP construct; (iv) take another per-thread time stamp directly after the construct, or in the case of `parallel` or `for`, directly in the construct. From these measurements, we then derive our metrics, providing information on minimum construct cost (first out - last in), average cost (average of end times - last in), etc. For example, in this way we can deduce the minimum time a barrier was active across all threads.

3.3 Window-Based Adjusted Synchronization Mechanism

The quality of our models depends on how accurately we can measure the timings of OpenMP constructs. All the threads should enter the construct at the same time, such that we have a uniform start time that does not depend on the particular construct being measured. A simple barrier synchronization is not enough, since the only guarantee it provides is that all threads will have arrived at the barrier before any thread leaves it. The solution, therefore, is to use a synchronization mechanism that forces all the threads to exit the synchronization construct at the same time. In this study, we use a variation of the window-based synchronization mechanism for MPI collective operations [7]. This mechanism forces the threads to wait until the agreed time-point is reached and only then allows them to enter the target construct.

The window-based synchronization mechanism assumes that all threads use the same clock. However, we discovered that this assumption does not apply to all test platforms equally. On some platforms, such as the BCS systems of RWTH Aachen University, which is described in Sect. 4.1, which consist of multiple motherboards, the high-precision timer used for our measurements was not well synchronized across all boards. Since we observed considerable clock skew, we had to calibrate clock offsets relative to the master thread using the cache coherency mechanisms as communication medium. This type of synchronization is similar to the NTP protocol [8].

4 Results

For the sake of brevity, we focus on a few very important OpenMP constructs: `parallel`, `barrier`, `single`, and `for` with all three schedule types (`static`, `dynamic` and `guided`) and the `firstprivate` modifier. Since our benchmarks consume a negligible amount of memory bandwidth, we can safely ignore bandwidth saturation effects. This also applies to the `firstprivate` measurement, which, in our case, uses a single eight-byte variable, which is sure to fit in the cache. We specified a chunk size of 16 for all loop schedules.

All models shown in the following sections depend on the number of threads as their sole parameter. Table 1 shows the performance models generated for the above-mentioned constructs together with their adjusted coefficient of determination as an indicator of model quality. Figures 2, 3 and 4 provide fit-comparisons between measurement and model. In general, we consider models with $\hat{R}^2 \geqslant 0.95$ valid descriptions of the observed behavior and define constructs with valid models of significantly faster than logarithmic growth to exhibit problematic scaling behavior.

4.1 Setup

We conducted our study on three different systems: (i) a node of the BCS cluster at RWTH Aachen University, (ii) an Intel Xeon Phi 7120 coprocessor, and (iii) a node of an IBM Blue Gene/Q system. The BCS cluster [9] is an Intel Xeon X7550-based hierarchical NUMA machine, where four boards with four sockets each are connected via the Bull Coherence Switch (BCS) to create a shared-memory domain of 128 physical cores. The Xeon Phi and the Blue Gene/Q node have 61 and 16 physical cores, respectively, with 4-way simultaneous multi-threading (SMT), i.e., four hardware threads per core. We used the GNU 4.9, IBM XL 12.1, Intel 15, and PGI 14 compilers. To reduce the effects of noise, we configured all benchmarks to generate at least 100 individual data points for each metric, i.e., we set the outer-repetitions of EPCC to 100 and compiled our own benchmarks with 100 internal repetitions after the warmup phase. We ran our benchmarks using numbers of threads that are either a power of two, multiples of eight, or a sequence between two and the number of physical cores of a single CPU. Each benchmark was executed in both `spread` or `close` configuration using `OMP_PROC_BIND`, with an additional binding to cores via `OMP_PLACES="threads"`. Afterwards, we eliminated outliers by removing the 25 % best and 25 % worst values of a series. Since the `close` measurements were noisier than the `spread` measurements on Intel platforms and largely identical to `spread` measurements on Blue Gene, we exclusively focus on `spread` in this paper.

4.2 GNU 4.9, Intel 15, and PGI 14 Compilers on BCS

Parallel. We obtain a timestamp on the master thread just before entering the `parallel` construct and on each thread when it is ready for work in the parallel region. Then, we calculate the difference between the master timestamp before entering the construct and the average of all timestamps after entering the construct. We expect either close to constant behavior, e.g., if a thread pool is used, or logarithmic behavior otherwise, as one could ideally implement a tree-based thread-creation scheme.

Unfortunately, indiscriminately feeding data points for all thread counts into the model generator did not yield any meaningful models. A subsequent manual analysis of the available data showed separate trend functions for different subsets of the data: for powers of two and for multiples of 16 with and without an eight-thread offset. We call these classes *PO2* ($t = 2^x$), *EVEN* ($t = 16x$ but $t \neq 2^x$), and *ODD* ($t = 16x + 8$ but $t \neq 2^x$) with $x \in \{0, 1, .., 7\}$. The effects observed for *EVEN* and *ODD* are most likely the result of the multi-board hardware configuration of the BCS system. However, regardless of internal hardware boundaries, *PO2* measurements consistently follow their characteristic pattern even if these thread counts are multiples of 16 with and without an eight-thread offset.

For example, as we can see in Figs. 2c and d, the behavior in the *ODD* case (half-circles) precludes the existence of a unifying simple model for the GNU compiler. Models for *EVEN* have very low $\hat{R}^2$ and will not be considered. Note that the number of thread counts in EVEN is very small because many multiples of 16 are at the same time powers of two and, thus, belong to a different behavioral class. In the remainder of the paper, we therefore concentrate exclusively on *PO2* and *ODD*. In contrast to GNU and PGI, the Intel compiler shows no observable differences between *PO2* and *ODD* configurations. We therefore omit *ODD* models for Intel on BCS in Table 1 and Fig. 2. Obviously, not all compliers are sensitive to the machine architecture.

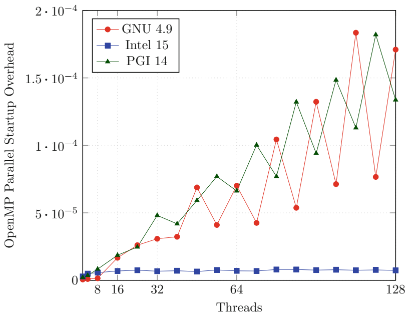
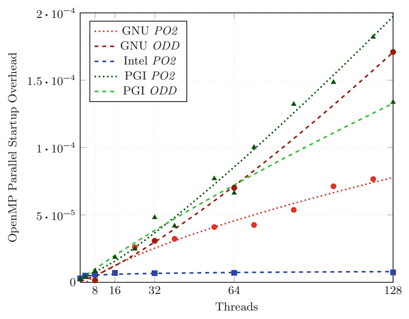


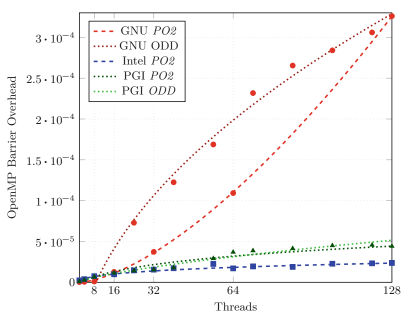
Fig. 1. Measurements of `parallel` on the BCS node in spread configuration. To make trends or their absence more visible, we the connected the measurement points with solid lines.

In Fig. 1, we see notable differences between GNU, Intel, and PGI compilers. Using the *PO2* and *ODD* configurations, we obtain two separate models each for both the GNU and the PGI compiler (Fig. 2a). These four models show super-logarithmic scaling behavior. In contrast, the Intel compiler exhibits a uniform trend, but with low $\hat{R}^2$. The almost constant time visible in Fig. 1 for Intel suggests the use of some form of stand-by threads that can be cost-efficiently activated.

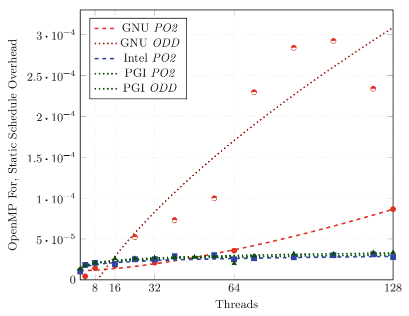
Barrier. We observe two different behavioral classes for the GNU compiler, while we observe similarly uniform behaviors for Intel and PGI (Fig. 2b). The *PO2* implementation of GNU shows super-linear growth in contrast to its somewhat



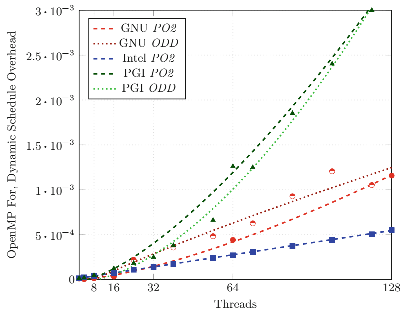
(a) BCS `parallel`



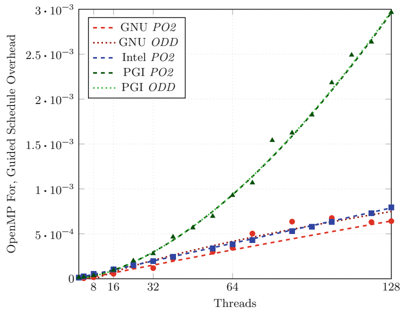
(b) BCS `barrier`



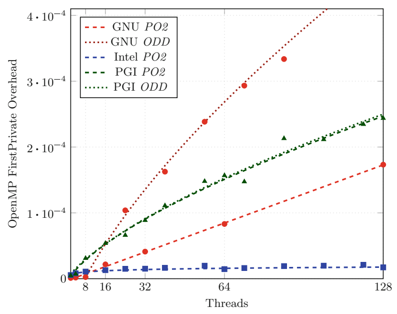
(c) BCS `static`



(d) BCS `dynamic`



(e) BCS `guided`



(f) BCS `firstprivate`

Fig. 2. Measurements (points) and models (lines) on the BCS node.

better-scaling *ODD* implementation. All but PGI *ODD* show worse-than-logarithmic growth, indicating that logarithmic implementations are possible.

Loop Schedules. For the `static` schedule (Fig. 2c), we expect constant overhead as no synchronization between threads is necessary and for `dynamic` (Fig. 2d) and `guided` (Fig. 2e) some thread-dependent growth for synchronizing the assignment of the remaining iterations. We obtained no acceptable *PO2* models for

Intel and PGI with `static`, as the model generator did not detect a clear trend; visual analysis of the data suggests close to constant overheads (Fig. 2c).

Firstprivate. This modifier requires the compiler to broadcast the values of one or more variables (in this particular case an 8-byte double) from the master thread to all participant threads. We expect this operation to be very sensitive to the hardware, as the latency between cores, sockets, and motherboards plays a crucial role. The Intel compiler exhibits logarithmic overheads for copying the data to each thread, whereas the overheads of both PGI and GNU grow faster. Again, the GNU compiler shows two clearly separable behaviors. Models for Intel and PGI show no sensitivity to the BCS hardware layout (Fig. 2f).

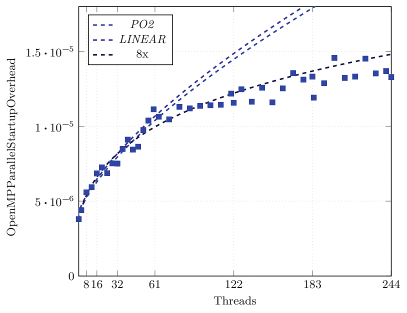
4.3 Intel 15 Compiler on Xeon Phi

On Xeon Phi, we expect less noise and more scalable OpenMP constructs. In Fig. 3a, we observe distinct behaviors for the first 2 to 61 threads, between 62 and 122 threads, and between 123 to 244 threads. This coincides with the physical

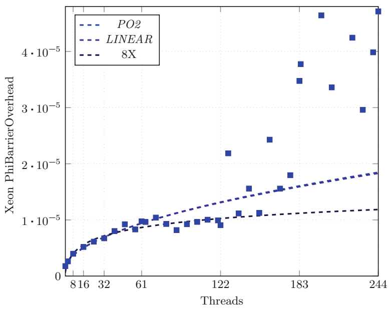
Table 1. Scaling models for the BCS node, XeonPhi, and Blue Gene/Q. Measurements with a † were generated using EPCC, measurements with ⋆ were generated using our supplemental benchmarks. Each row showing models is followed by a row with the corresponding adjusted coefficient of determination ($\hat{R}^2$). Since we are only interested in the scaling behavior and do not strive to predict the overhead in absolute terms, all models are shown in big O notation.

	Parallel Open⋆	Barrier⋆	Dynamic 16†	Static 16†	Guided 16†	Firstprivate†
BCS - GNU						
PO2	$\mathcal{O}(t^{1.25})$	$\mathcal{O}(t^{1.33}\log t)$	$\mathcal{O}(t^{1.25}\log t)$	$\mathcal{O}(t^{1.33}\log t)$	$\mathcal{O}(t^{0.75}\log t)$	$\mathcal{O}(t)$
$\hat{R}^2$	0.99	0.99	0.99	0.98	0.99	0.99
ODD	$\mathcal{O}(t^{0.67})$	$\mathcal{O}(t^{0.5})$	$\mathcal{O}(t^{0.67}\log t)$	$\mathcal{O}(t^{1.25}\log t)$	$\mathcal{O}(t^{0.5}\log t)$	$\mathcal{O}(t^{0.5}\log t)$
$\hat{R}^2$	0.95	0.93	0.96	0.94	0.93	0.98
BCS - Intel						
PO2	$\mathcal{O}(\log t)$	$\mathcal{O}(t^{0.25})$	$\mathcal{O}(t)$	$\mathcal{O}(\log t)$	$\mathcal{O}(t)$	$\mathcal{O}(\log t)$
$\hat{R}^2$	0.78	0.98	0.99	0.84	0.99	0.94
BCS - PGI						
PO2	$\mathcal{O}(t^{0.67}\log t)$	$\mathcal{O}(\log^2 t)$	$\mathcal{O}(t^{1.25}\log t)$	$\mathcal{O}(\log t)$	$\mathcal{O}(t^{1.67})$	$\mathcal{O}(t^{0.67})$
$\hat{R}^2$	0.99	0.95	0.99	0.62	0.99	0.99
ODD	$\mathcal{O}(t\log t)$	$\mathcal{O}(t^{0.5}\log t)$	$\mathcal{O}(t^{1.5}\log t)$	$\mathcal{O}(t^{2.5})$	$\mathcal{O}(t^{1.67})$	$\mathcal{O}(t^{0.5}\log t)$
$\hat{R}^2$	0.97	0.90	0.99	0.50	0.99	0.89
XeonPhi - Intel						
PO2	$\mathcal{O}(t^{0.67})$	$\mathcal{O}(t^{0.5})$	$\mathcal{O}(t^{0.25})$	$\mathcal{O}(t^{1.5})$	$\mathcal{O}(t)$	$\mathcal{O}(t^{0.67})$
$\hat{R}^2$	0.97	0.99	0.65	0.75	0.99	0.98
LINEAR	$\mathcal{O}(t^{0.67})$	$\mathcal{O}(t^{0.5})$	$\mathcal{O}(\log t)$	$\mathcal{O}(t^{2.33})$	$\mathcal{O}(t)$	$\mathcal{O}(t^{0.67})$
$\hat{R}^2$	0.95	0.94	0.55	0.30	0.99	0.96
8X	$\mathcal{O}(\log^2 t)$	$\mathcal{O}(\log t)$	$\mathcal{O}(t^{1.25}\log t)$	$\mathcal{O}(t^{0.75}\log t)$	$\mathcal{O}(t)$	$\mathcal{O}(\log^2 t)$
$\hat{R}^2$	0.95	0.86	0.92	0.70	0.99	0.94
Blue Gene/Q - IBM XL						
PO2	$\mathcal{O}(t^{1.25})$	$\mathcal{O}(t^{1.33}\log t)$	$\mathcal{O}(t^{2.33})$	$\mathcal{O}(t^{2.33})$	$\mathcal{O}(t^{2})$	$\mathcal{O}(t^{1.25})$
$\hat{R}^2$	0.99	0.99	0.99	0.99	0.99	0.99

structure of the Xeon Phi, which has 61 cores supporting four hardware threads each. The first 122 threads show less spread in comparison with thread counts above 122. Because the erratic runtimes above 122 threads prevent the use of our model generator, we model the first two clusters only. We consider the first 2–61 threads in linear fashion, called *LINEAR*, and multiples of eight up to and including 120 threads, called *8X*. In addition, we also analyze powers of two up to and including 64 threads, again called *PO2*. All results are available in Table 1.



(a) XeonPhi `parallel`



(b) XeonPhi `barrier`

Fig. 3. Measurements (points) and models (lines) on XeonPhi.

Parallel. The *LINEAR* and *PO2* thread distributions have similar scalability models and closely model the first 61 threads. For thread counts beyond 61, the deviation becomes larger. The model derived from *8X* configurations captures the overall behavior of Xeon Phi thread creation quite well, including thread counts above 122. The erratic runtimes for thread counts above 122 (Fig. 3a) cannot be explained with our model normal form. When comparing the different configurations, models generated from *8X* base points seem to scale better.

Barrier. The behavior we observe is similar to the parallel construct. The *PO2*, *LINEAR* and *8X* configurations provide a good fit for their respective domains. The erratic behavior above 122 threads is even more dominant here, which is why again no models could be generated for this part of the domain (Fig. 3b). However, the measurements above 122 threads still suggest some undesirable performance effect, potentially resulting from Xeon Phi's internal network, something that is traditionally hard to model

4.4 IBM XL 12.1 Compiler on Blue Gene/Q

Blue Gene/Q nodes are single-socket systems without any explicit cache hierarchy. Analysis of our measurements showed very reliable data with very little noise and no indication of multiple algorithms or thread-count depended hardware scalability limitations. We therefore used only power-of-two configurations as input for our model generator.

Parallel and Barrier. Contrary to our expectations, either the IBM implementation of OpenMP or the Blue Gene/Q architecture exhibits problematic scaling behavior. We observe that metrics exhibit superlinear growth (see lower lines in Fig. 4). The model for the barrier exhibits similar behavior with just an order of magnitude lower overheads.

Loop Schedules. For `static` scheduling, which should have constant overhead, we detected non scalable growth. The `static` schedule showed runtimes and behavior almost identical to the dynamic schedule, suggesting that both use the same algorithm; the `guided` scheduling clause behaves similarly. While these results are less of a concern for today's Blue Gene/Q systems with only 64 threads per node, the scaling model indicates problematic overheads of the OpenMP constructs for larger thread counts on future systems with similar architecture and software. In comparison with the often logarithmic implementations of the Intel Compiler, the IBM XL compiler shows considerable room for improvement.

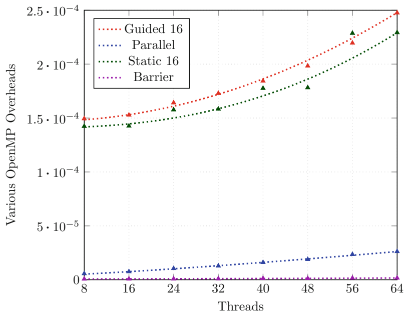


Fig. 4. Measurements (points) and models (lines) on BlueGene/Q.

5 Related Work

Performance models can provide important insights into application and systems. Manually-produced models were very effective in describing many qualities and characteristics of applications, systems, and even entire tool chains [10–12]. Recent work suggests to use source-code annotations [13] or specialized languages [14] to support developers in the creation of analytical performance models.

There are other automated modeling methods besides the one underlying our study. Many of these tools focus on learning the performance characteristics automatically using various machine-learning approaches [15]. Zhai et al. extrapolate single-node performance to complex parallel machines using a trace-driven network simulator [16], whereas Wu and Müller extrapolate traces to predict communications at larger scale [17]. Similar to our method, Carrington et al. extrapolate trace-based performance measurements using a set of canonical functions [18].

Several studies investigated the overheads of OpenMP constructs on various platforms [19–22]. Similar to our work, many of them used the EPCC OpenMP benchmark suite [4]. While they mainly concentrated on the implications the overhead of OpenMP may have on the scalability of scientific applications, our goal is to identify scalability issues in OpenMP implementations. One of the first performance evaluation of OpenMP on XeonPhi was performed by Cramer et al. [23]. Eichenberger and O'Brien evaluated the overhead of the OpenMP runtime on Blue Gene/Q [24].

6 Conclusion

In this work, we analyzed the scalability of OpenMP constructs using automatically generated empirical performance models. We conducted extensive evaluations of OpenMP implementations from Intel, GNU, PGI and IBM on Intel-based nodes as well as on IBM Blue Gene/Q nodes. In many cases, the behavior of OpenMP constructs deviated from our expectations and numerous scalability issues became apparent. We expected either logarithmic or constant growth of OpenMP overheads, but discovered mostly linear and super-linear growth. Neither of the evaluated compilers proved to be the best implementation in all situations. The Intel compiler showed the best absolute performance and scaling behavior for most of the metrics in our tests, but it was still surpassed by the PGI compiler on two occasions. Considering the increasing degree of intra-node parallelism, OpenMP compilers will have to tackle theses scalability issues in the future. Our benchmarking method is designed to support this process, as it can be used to continuously evaluate implementations as their scalability is improved.

Acknowledgment. This work was performed under the auspices of the DFG Priority Programme 1648 "Software for Exascale Computing" (SPPEXA). The authors thank Christian Terboven for the fruitful discussions on scalability expectations for OpenMP and for providing access to the BCS machine at RWTH Aachen University.

References

1. Stevens, R., et al.: Architectures and Technology for Extreme Scale Computing. Technical report, ASCR Scientific Grand Challenges Workshop Series, December 2009
2. Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2013), p. 45 (2013)
3. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
4. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. ACM SIGARCH Comput. Archit. News **29**(5), 41–48 (2001)
5. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
6. Picard, R.R., Cook, R.D.: Cross-validation of regression models. J. Am. Stat. Assoc. **79**(387), 575–583 (1984)
7. Hoefler, T., Schneider, T., Lumsdaine, A.: Accurately measuring collective operations at massive scale. In: Proceedings of the IEEE International Parallel & Distributed Processing Symposium, IPDPS 2008, pp. 1–8 (2008)

8. Mills, D.L.: Internet time synchronization: the Network Time Protocol. IEEE Trans. Commun. **39**(10), 1482–1493 (1991)
9. Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T.W., Müller, M.S., Hentschel, B.: Visualization of memory access behavior on hierarchical NUMA architectures. In: Proceedings of the First Workshop on Visual Performance Analysis, VPA 2014, Piscataway, NJ, USA, pp. 42–49. IEEE Press (2014)
10. Mathis, M.M., Amato, N.M., Adams, M.L.: A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. Technical report, College Station, TX, USA (2000)
11. Pllana, S., Brandic, I., Benkner, S.: Performance modeling and prediction of parallel and distributed computing systems: a survey of the state of the art. In: Proceedings of the 1st International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 279–284 (2007)
12. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC 2003), p. 55 (2003)
13. Tallent, N.R., Hoisie, A.: Palm: easing the burden of analytical performance modeling. In: Proceedings of the International Conference on Supercomputing (ICS), pp. 221–230 (2014)
14. Spafford, K.L., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC 2012, Los Alamitos, CA, USA, pp. 84:1–84:11. IEEE Computer Society Press (2012)
15. Lee, B.C., Brooks, D.M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: Methods of inference and learning for performance modeling of parallel applications. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2007), pp. 249–258 (2007)
16. Zhai, J., Chen, W., Zheng, W.: PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. SIGPLAN Not. **45**(5), 305–314 (2010)
17. Wu, X., Mueller, F.: ScalaExtrap: trace-based communication extrapolation for SPMD programs. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP 2011), pp. 113–122 (2011)
18. Carrington, L., Laurenzano, M., Tiwari, A.: Characterizing large-scale HPC applications through trace extrapolation. Parallel Process. Lett. **23**(4), 1340008 (2013). doi:10.1142/S0129626413400082
19. Fredrickson, N.R., Afsahi, A., Qian, Y.: Performance characteristics of OpenMP constructs, and application benchmarks on a large symmetric multiprocessor. In: Proceedings of the 17th Annual International Conference on Supercomputing, pp. 140–149. ACM (2003)
20. Fürlinger, K., Gerndt, M.: Analyzing overheads and scalability characteristics of OpenMP applications. In: Daydé, M., Palma, J.M.L.M., Coutinho, A.L.G.A., Pacitti, E., Lopes, J.C. (eds.) VECPAR 2006. LNCS, vol. 4395, pp. 39–51. Springer, Heidelberg (2007)
21. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on chip multithreading platforms. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/2006. LNCS, vol. 4315, pp. 178–190. Springer, Heidelberg (2008)

22. Bronevetsky, G., Gyllenhaal, J., de Supinski, B.R.: CLOMP: accurately characterizing OpenMP application overheads. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 13–25. Springer, Heidelberg (2008)
23. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP programming on Intel Xeon Phi coprocessors: an early performance comparison. In: Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, pp. 38–44, November 2012
24. Eichenberger, A.E., O'Brien, K.: Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q. IBM J. Res. Dev. **57**(1/2), 8–1 (2013)