Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 973–978
DOI: 10.15439/2023F7822
ISSN 2300-5963 ACSIS, Vol. 35

Performance assessment of OpenMP constructs and benchmarks using modern compilers and multi-core CPUs

Bartłomiej Gawrych and Paweł Czarnul
Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology
Narutowicza 11/12, 80-233 Poland, email: pczarnul@eti.pg.edu.pl

***Abstract*—Considering ongoing developments of both modern CPUs, especially in the context of increasing numbers of cores, cache memory and architectures as well as compilers there is a constant need for benchmarking representative and frequently run workloads. The key metric is speed-up as the computational power of modern CPUs stems mainly from using multiple cores. In this paper, we show and discuss results from running codes such as: batch normalization, convolution, linear function, matrix multiplication, prime number test and wave equation; using compilers such as: GNU gcc, LLVM clang, icx, icc; run on four different 1 or 2-socket systems: 1 x Intel Core i7-5960X, 1 x Intel Core i9-9940X, 2 x Intel Xeon Platinum 8280L, 2 x Intel Xeon Gold 6130. Results can be regarded as suggestions concerning scaling on particular CPUs including recommended thread number configurations.**

I. Introduction

PARALLEL computing has become increasingly popular due to the widespread availability of multi- and many-core CPUs and accelerators such as GPUs, not only in cluster nodes, servers and workstations but also desktops and even mobile devices. In line with the hardware developments, many APIs are used for general purpose programming in such environments, including: OpenMP and OpenCL for shared memory systems with offloading to accelerators, OpenACC for directive based accelerator programming, CUDA for NVIDIA GPUs, Message Passing Interface (MPI) for internode communication among processes of a parallel application. OpenMP is very important due to its relatively easy to learn directive + library based multithreaded model allowing easy parallelization of sequential codes and support for offloading computations to accelerators such as GPUs [1].

The contribution of this paper over the state-of-the-art described in Section II, is assessment of OpenMP's implementation performance for a combination of: a variety of specific constructs and benchmarks, each of which benchmarked on various 1 and 2 socket systems with modern multi-core Intel CPUs and each tested using 4 compilers: GNU gcc, LLVM clang, icx, icc, run for various data sizes. Benchmarks include: batch normalization used in deep learning, convolution frequently used in signal processing, linear addition function benchmark, matrix multiplication, prime number test as well as wave equation simulation.

II. Related work

In [2] a set of microbenchmarks derived from EPCC and based on SKaMPI was run and analyzed on IBM SP3 and SunFire systems. Those included OpenMP's lock/unlock, critical section, barrier, single, parallel and parallel for directives. Times were measured for the two systems between 1 and 8 processors showing generally much better values for the Sun system especially showing sharp increases of times across the ranges for IBM SP3 vs Sun for barrier, for reduction, parallel and for single for >2 processors, critical for >4 and lock/unlock for >5 processors. In [3] performance of a Loongson-3A SMP quad-core system was assessed for EPCC microbenchmarks and NPB, using: gcc, OMPi with pthreads or psthreads. Testing parallel, for, parallel for, barrier and single for 1-4 threads, OMPi+pthreads tested best; for critical, unlock/lock, ordered and atomic gcc resulted in much larger overhead for 2-4 threads than the other very comparable solutions. For loop scheduling: static OMPi+pthreads and gcc were best while for dynamic and guided OMPi+pthread shall be preferred. The analyzed platform was also compared to Intel i5 with normalized (versus CPU clock) ratios for NPB (4 threads) between 1.3 (EP) and 5.1 (CG).

Authors of paper [4] benchmarked a 72-way Sun Fire 15K multiprocessor system with several EPCC microbenchmarks including measurements of overheads of OpenMP's frequently used construct implementations. OpenMP directives benchmarked included parallel, for, parallel for, barrier, single, critical, lock/unlock, atomic, along with scheduling modes such as static, dynamic and guided (1-128 chunk size). C and Fortran implementations were tested using 6, 12, 24, 48, 64 and 70 threads. Generally, overheads increase expectedly with the number of threads, in selected cases considerably starting with a given number of threads e.g. 48+ threads for critical, lock/unlock and atomic for the C implementation. Additionally, overhead of approximately 20% was measured for separate parallel+for in C and equivalent parallel+do in Fortran compared to combined versions. For NAS parallel benchmarks various maximum speed-ups were obtained: approximately 50 for BT and SP, 70 for LU, over 95 (superlinear) for CG, over 50 for MG and over 20 for FT.

In paper [5] the author investigated various OpenMP implementations of one of the most popular parallel programming

paradigms – master-slave. Six versions were implemented, based on: OpenMP locks, the tasking construct, for loop dynamically partitioned, the latter two without and with overlapping merging results and data generation. Two concrete applications were implemented: one with irregular adaptive quadrature numerical integration and the second implementing finding a region of interest within an irregular image. gcc version 9.3.0 was used on two systems with: the first one with Intel i7-7700 3.60 GHz Kaby Lake CPU and 8 logical processors, the second one with two Intel Xeon E5-2620 v4 2.10 GHz Broadwell CPUs and 32 logical processors. All in all, for integration the best results were obtained for tasking and dynamic for with or without overlapping (for systems 2 and 1) while for image recognition for system 1 dynamic for and using locks while for system 2 dynamic for (both versions) and tasking with overlapping.

Scalability and overheads during execution of parallel code is studied in more detail in [6] where authors distinguished 4 overhead categories such as: need for synchronization among threads, imbalance, limited parallelism i.e. not (fully) parallelized code and thread management. For benchmarking the authors used OpenMP's version of NAS Parallel Benchmarks (class C) characterizing presence of particular OpenMP constructs in particular benchmarks – present mostly LOOP, PARALLEL and PARALLEL_LOOP in all tested as well (except PARALLEL in FT) as master in BT, LU, MG and SP; ATOMIC in BT, EP, LU and SP; BARRIER in IS and LU; CRITICAL in SP; SINGLE in LU. Codes were benchmarked using between 2 and 32 threads on an 32 CPU Itanium-2 based SGI Altix machine. All in all, imbalance appeared to be the largest overhead generally, as much as 20% for SP; synchronization turned out to be significant for IS (largest) and visible for LU. Thread management was noticed in IS, MG and CG although not large.

In paper [7] authors implemented and benchmarked 3 versions of OpenMP codes for an iterative Jacobi solver for 2D structured grids, representative of geometric SPMD codes such as for e.g. CFD applications. The code versions included: standard shared memory OpenMP host code with `parallel` and `do` directives, standard code augmented with `target` and `target data` directives and code with `target`, `target data`, `teams`, `distribute` directives. Codes were run on 2 systems: one with 2 Intel Xeon E5-2670 CPUs + 4 Intel 5110P Phis, the other with AMD Interlagos CPU + NVIDIA K20X GPU. For the first system (Intel compiler), offloading has been shown to be effective, even if offloading to self case, almost as good as the standard OpenMP code. For the second system (Cray compiler), only standard code on CPU and offload to GPU performed well.

In work [8] authors benchmarked OpenMP as a programming API through its language constructs on the IBM Cyclops64 system with 160 processing cores within a single chip. Specifically, EPCC microbenchmarks were used with their 3 elements testing: synchronization, scheduling as well as array directives and clauses. Overheads in terms of cycles versus numbers of threads within the 1-128 range were tested. Specifically, the overhead of `FOR` turned out to be only minimally higher than that of `BARRIER` and of `PARALLEL FOR` minimally larger than that of `PARALLEL`. The overhead of `SINGLE` is comparatively large. `DYNAMIC(1)` resulted in very large overhead, especially compared to `DYNAMIC` for chunk sizes 64-128 and `STATIC` for equivalent chunk sizes. Additionally, overheads of `PRIVATE` and `FIRSTPRIVATE` used in conjunction with `PARALLEL` add very small, minimally larger for the latter.

III. Methodology and benchmarks

Within the paper, we aim at comparative analysis of several orthogonal aspects in terms of OpenMP applications, including: many various benchmarks that differ in compute and memory intensity, as well as OpenMP directives used; tests for various input data sizes; several popular compilers: GNU gcc, LLVM clang, icx, icc; several CPUs representing various architectures and generations.

To evaluate performance, several programs were written using various OpenMP directives and their combinations. Problems benchmarked are as follows:

- **Batch-Normalization** – popular function used in deep learning, especially in computer vision problems [9].
- **Convolution** – method commonly used in signal processing, but also very popular in computer vision problems. This benchmark tests the parallelization of five nested loops and how collapse directive and its parameters are impacting performance.
- **Linear function** – performing multiplication and addition to each element of the array ($y = a * x + b$), testing if OpenMP's SIMD directive affects the execution time of the program.
- **Matrix multiplication** – we used implementation with O(n^3) complexity and parallelization with OpenMP's `schedule(static)` and `collapse` directives.
- **Prime number test** – implementation, which divides number by all numbers from 2 to $\sqrt{n}$, has been chosen in order to compare schedule clauses that are static, guided, and dynamic.
- **Wave equation** – benchmark testing the performance impact of using the `parallel` directive both within and outside of the time step loop (making the `parallel` directive called only once).

The experiments were carried out to test the performance of specific OpenMP implementations with an increasing number of threads for varying issue sizes, as well as the effect of various work-sharing directives. Multiple iterations of benchmarks were performed using sizes that were selected based on subjective criteria in order to conduct operations on various sizes, from small to large. The number of rounds was adjusted so the fastest execution of full benchmark measurement took longer than 1 second. Apart from measuring average time of a single run, our testing framework also calculated standard deviation which can be found on GitHub [10], along with full compilation configuration and compiler flags used for each platform.

IV. Experiments

A. Testbed environments

Table I details tested systems with configurations imposed by the production environments.

	CPU	S/C/T	Operating System	RAM
a)	Xeon Gold 6130	2/16/32	Ubuntu 18.04.3 LTS	256 GB
b)	Xeon Platinum 8280L	2/28/56	CentOS Linux 7 (Core)	192 GB
c)	Core i7-5960X	1/8/16	Ubuntu 18.04.5 LTS	16 GB
d)	Core i9-9940X	1/14/28	Ubuntu 20.04.2 LTS	128 GB

TABLE I
CONFIGURATION USED TO BENCHMARK OPENMP IMPLEMENTATIONS (S - SOCKETS, C - CORES, T - THREADS)

Table II presents compilers and OpenMP versions which were used to evaluate the performance. Our intention was to compile with the latest stable OpenMP release available for each compiler at the time.

Compiler	Compiler Version	Name of OpenMP lib	OpenMP version in CMake
GNU GCC	10.2.0	libgomp.so	4.5
LLVM Clang	11.1.0	libomp.so	5.0
ICX	12.0.0	libiomp5.so	4.5
ICC	20.2.2.20210228	libiomp5.so	5.0

TABLE II
BENCHMARKED OPENMP IMPLEMENTATIONS

B. Tests

Within the following tests, we present speed-ups versus the number of threads executing a particular benchmark, in selected cases for several variants and settings.

1) Batch-Norm: With the small problem size for Batch-Norm, the best performance improvement was achieved on processor (b) - peak performance was observed using only 32 threads and it was 25 times faster than sequential run. Using only 32 threads also gives the best result on processor (a), for the rest, using all available threads constituted an optimal solution.

With increased size the best performance for all examined CPUs was achieved using all available threads (Figure 1). The rapid performance loss that occurs when employing one more thread than half of those available is an interesting phenomenon – at this point, hyper-threading begins to function. Despite this problem, performance increases linearly when using more and more threads. This does not apply to the result of the GCC compiler on machine (b) where performance started to decline progressively once more than 32 threads were used.

2) Convolution: Results from Figure 2 demonstrate how crucial it is to employ the collapse clause when appropriate. If CPUs have enough threads to consume the first loop entirely, the rest of available threads will be idle. Using the `collapse` clause generates many more tasks which can be distributed among different threads, which results in better scalability than without this clause – characteristic speed-up when using divisible number of threads in relation to the iteration count

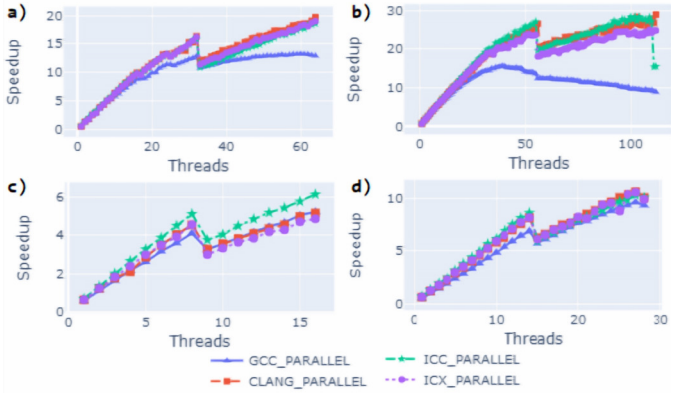

Fig. 1. Results of Batch-Norm with size N=32, C=2048, H=7, W=7

of first loop no longer exists. Overall, deciding whether it is always better to use 2-level or 3-level collapsing cannot be done, as it depends on the used compiler and the machine – e.g. for the Clang compiler on machine (a), best improvement is for collapse(2), but on machine (b) for collapse(3).

The results performed for N=256, H=112, W=112, kernel=3x3 indicate that when the first level loop has a large enough number of iterations to distribute tasks for each thread it is worthwhile to consider not using the `collapse` clause at all. It can be observed on machine (c) with GCC compiler and machine (d) ICC compiler.

3) Matrix Multiplication: For matrix-vector multiplication, in the case of desktop processors – (c) and (d), the results are satisfying and linear improvement can be observed for all compilers. Using the `collapse` clause has neither beneficial nor negative effect there. However, in server type CPUs differences show up. On machine (a) using the `collapse` clause causes performance degradation for every compiler. On machine (b) linear speed-up was disrupted by occurred anomalies after using more than 28 threads, which indicates the use of the second NUMA node.

For small square matrix-matrix multiplication desktop CPUs scale well and only a characteristic performance drop becomes apparent when hyper-threading comes into play. For configuration (a) and (b) compilers ICC and ICX allow good scaling and positive effect of using the `collapse` clause can be observed. For other configurations, scaling is much more irregular - especially for Clang on machine (b).

Increasing the size by an order of magnitude in each dimension causes all configurations but (b) to suffer from using hyper-threading as performance drops dramatically. Additionally, differences between using or not-using collapse construction are not visible for this size. The best scaling can be observed for configuration (b), but again, with anomalies visible in Figure 3 - when using HT threads.

4) Linear function: Tests performed for size=10000 showed very limited speed-ups. When the size is two orders of magnitude larger, the results are significantly better. In this example, letting the OpenMP implementation to determine chunk size automatically produces far better results than using

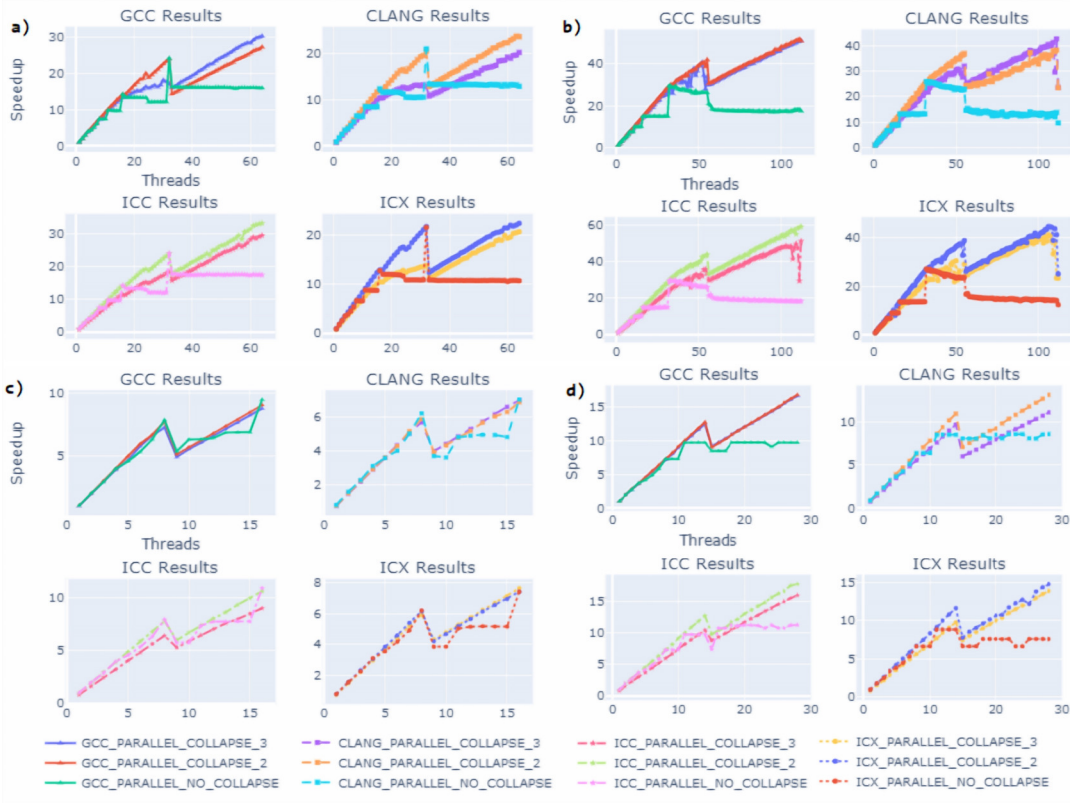

Fig. 2. Results of Convolution with size N=32, H=224, W=224, kernel=7x7

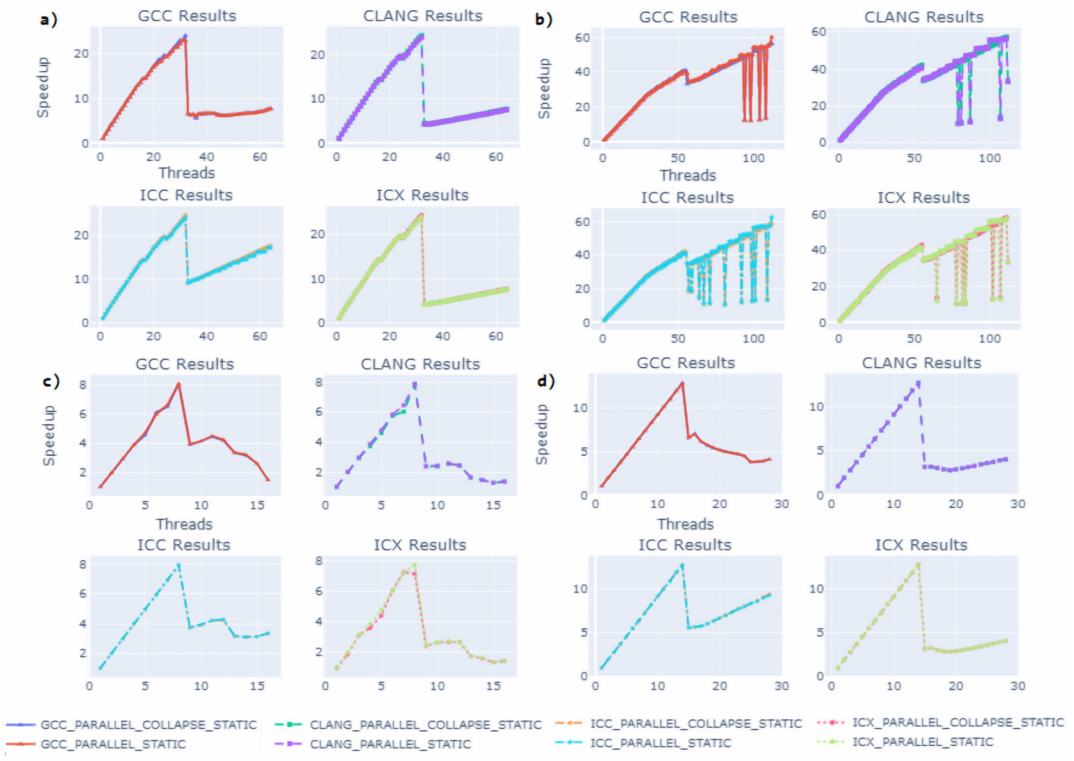

Fig. 3. Results of Matrix Multiplication with size N=1000, M=1000, K=1000

the `static` clause with a manually set chunk size. When it comes to the SIMD construction, in almost all cases it does not matter, but using this clause is beneficial when using ICC, but only on machine (d).

5) PrimeTest: Testing whether a given number is prime or not with a plain algorithm produces unbalanced amounts of work for particular threads. Results for size=10000 show that using the `guided` scheduling clause is the most stable one for every compiler. Poor performance for dynamic scheduling is visible for GCC and ICC for configurations (a), (b) and (d), but not for configuration (c), where it gives the best improvement. Another interesting observation is that a characteristic performance drop when CPU starts using hyper-threading vanished with the usage of guided scheduling.

For larger vectors of numbers to test, charts in Figure 4 are reasonably smooth and regular. Differences between using the `dynamic` and `guided` clauses are not visible and in the end, almost every configuration achieves the same level of parallelization when using all threads (except for dynamic scheduling using GCC and ICC compilers). Performance drop for static scheduling and hyper-threading still appears.

6) Wave Equation: The final benchmark determines whether it is better to place the `parallel` directive within or outside of a time-step loop, as the second iteration depends on first iteration's results. Results shown in Figure 5 are ambiguous. In most cases placing the `parallel` directive together with work-sharing for-loop gives better results. One exception to this appears on machine (b) when compiling with Clang – the chart is very irregular, but it is clear that placing parallel outside of the time-step loop produces better results.

For the larger sizes of the problem (N=5000, M=5000, T=100 tested) the differences are smaller and for desktop CPUs are almost not visible. Similar conclusions can be drawn for the server CPU from configurations where charts are overlapping each other. Only for configuration (a) some differences occurs – slightly better performance is observed when parallel clause is inside time-step loop, but scaling is then more irregular.

V. Summary and future work

Results show that no single best compiler nor OpenMP implementation can be chosen. In many cases compilers showed similar speed-up patterns on charts, however they differed in speed-up values. Different results and rankings were collected for various problems and various sizes of problems – consequently best configurations need to be considered on a case by base basis. In line with expectations, better scaling was achieved for bigger data sizes of problem – that suggests that data size must be large enough to get satisfying speed-ups. Often for small data sizes better performance can be achieved by using relatively few cores. It is especially visible when comparing desktop CPUs (a small number of cores) with server CPUs (a large number of cores) – in some cases exceeding a certain number of used cores caused degradation of performance. Consequently, it is recommended to benchmark own program with different OpenMP implementations in a production environment to get the best results in terms of performance before final deployment.

For future work, it would be valuable to benchmark the workloads also under power caps and determine performance-energy trade-offs [11] including optimization goals EDP, EDS as well as percentage wise performance loss for energy gains. Additionally, other benchmarks would also be of interest, such as: parallel similarity measure computations for large vectors [12] or image processing [13].

References

[1] P. Czarnul, *Parallel Programming for Modern High Performance Computing Systems*. CRC Press, Taylor & Francis, 2018, iSBN 9781138305953.

[2] A. Prabhakar, V. Getov, and B. Chapman, "Performance comparisons of basic openmp constructs," in *High Performance Computing*, H. P. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. ISBN 978-3-540-47847-8 pp. 413–424.

[3] Q. Luo, C. Kong, Y. Cai, and G. Liu, "Performance evaluation of openmp constructs and kernel benchmarks on a loongson-3a quad-core smp system," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2011. doi: 10.1109/PDCAT.2011.66 pp. 191–196.

[4] N. R. Fredrickson, A. Afsahi, and Y. Qian, "Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: Association for Computing Machinery, 2003. doi: 10.1145/782814.782835. ISBN 1581137338 p. 140–149. [Online]. Available: https://doi.org/10.1145/782814.782835

[5] P. Czarnul, "Assessment of openmp master–slave implementations for selected irregular parallel applications," *Electronics*, vol. 10, no. 10, 2021. doi: 10.3390/electronics10101188. [Online]. Available: https://www.mdpi.com/2079-9292/10/10/1188

[6] K. Fürlinger and M. Gerndt, "Analyzing overheads and scalability characteristics of openmp applications," in *High Performance Computing for Computational Science - VECPAR 2006*, M. Daydé, J. M. L. M. Palma, Á. L. G. A. Coutinho, E. Pacitti, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. ISBN 978-3-540-71351-7 pp. 39–51.

[7] V. G. M. Vergara, W. D. Joubert, M. G. Lopez, and O. R. Hernandez, "Early experiences writing performance portable openmp 4 codes," in *Cray User Group Conference*, London, United Kingdom, May 2016.

[8] W. Zhu, J. del Cuvillo, and G. R. Gao, "Performance characteristics of openmp language constructs on a many-core-on-a-chip architecture," in *OpenMP Shared Memory Parallel Programming*, M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68555-5 pp. 230–241.

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[10] B. Gawrych and P. Czarnul, "Performance investigation of openmp constructs and benchmarks using modern compilers and multi-core cpus," September 2021, https://github.com/bgawrych/openmp_benchmark.

[11] A. Krzywaniak, J. Proficz, and P. Czarnul, "Analyzing energy/performance trade-offs with power capping for parallel applications on modern multi and many core processors," in *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2018, pp. 339–346.

[12] P. Czarnul, P. Rościszewski, M. Matuszek, and J. Szymański, "Simulation of parallel similarity measure computations for large data sets," in *2015 IEEE 2nd International Conference on Cybernetics (CYBCONF)*, 2015. doi: 10.1109/CYBConf.2015.7175980 pp. 472–477.

[13] P. Czarnul, A. Ciereszko, and M. Frązak, "Towards efficient parallel image processing on cluster grids using gimp," in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and

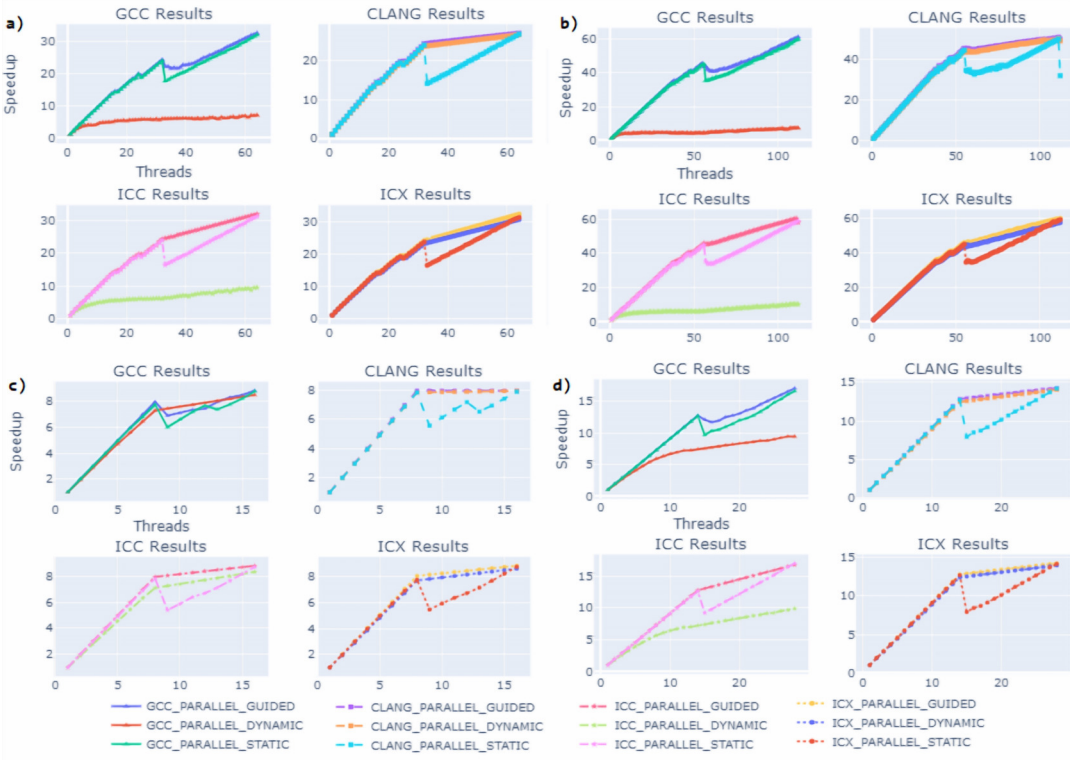

Fig. 4. Results of Prime Test with size=10000000

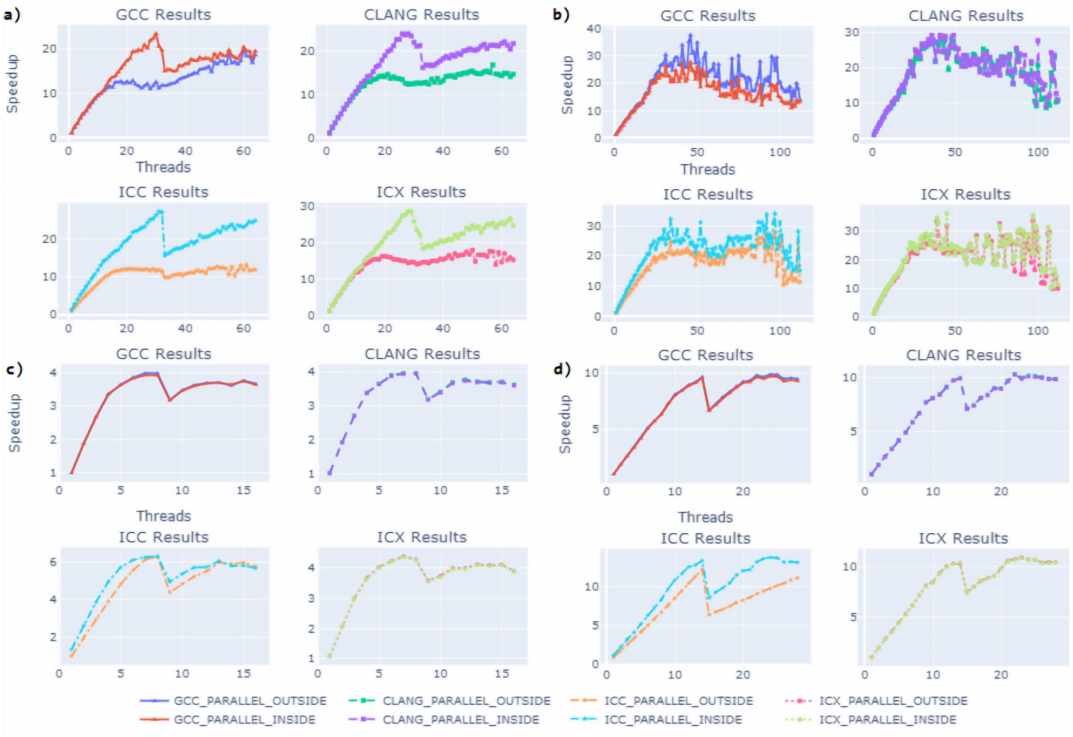

Fig. 5. Results of Wave Equation with N=1000, M=1000, T=10

J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN 978-3-540-24687-9 pp. 451–458.