Comparing the behavior of OpenMP Implementations with various Applications on two different Fujitsu A64FX platforms

Benjamin Michalowicz
benjamin.michalowicz@stonybrook.edu
Stony Brook University
New York, USA

Eric Raut
Stony Brook University
New York, USA
eric.raut@stonybrook.edu

Yan Kang
Stony Brook University
New York, USA
yan.kang@stonybrook.edu

Tony Curtis
Stony Brook University
New York, USA
anthony.curtis@stonybrook.edu

Barbara Chapman
Stony Brook University
New York, USA
barbara.chapman@stonybrook.edu

Dossay Oryspayev
Brookhaven National Laboratory
New York, USA
doryspaye@bnl.gov

ABSTRACT

The development of the A64FX processor by Fujitsu has been a massive innovation in vectorized processors and led to Fugaku: the current world's fastest supercomputer. We use a variety of tools to analyze the behavior and performance of several OpenMP applications with different compilers, and how these applications scale on the different A64FX processors on clusters at Stony Brook University and RIKEN.

CCS CONCEPTS

• **Computer systems organization → HPC Architecture**; Compiler Toolchains; • **High Performance Computing →** *OpenMP*; Parallel Programming.

KEYWORDS

OpenMP, High Performance Computing, Ookami, A64FX, Fujitsu, Fugaku

ACM Reference Format:
Benjamin Michalowicz, Eric Raut, Yan Kang, Tony Curtis, Barbara Chapman, and Dossay Oryspayev. 2021. Comparing the behavior of OpenMP Implementations with various Applications on two different Fujitsu A64FX platforms. In *Practice and Experience in Advanced Research Computing (PEARC '21), July 18–22, 2021, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3437359.3465592

1 INTRODUCTION

The introduction of the A64FX processor by Fujitsu has sparked an innovation in vectorized processors and the birth of Fugaku: the current world's-fastest supercomputer [1]. The A64FX chip also brings an unprecedented co-design approach, impressive performance, and energy-awareness that puts it at the top position on all 5 major HPC benchmarks. In this (short) paper, we analyze the OpenMP [7] shared-memory/parallel programming model, from how it scales on the A64FX – and its variants on Ookami and Fugaku, see Section 2 – to performance across different compiler toolchains.[2]

The A64FX processor [6, 10] is the processor used in the Fugaku supercomputer, enabled by the Japanese FLAGSHIP 2020 project as a co-design between RIKEN and Fujitsu. Currently, Fugaku is ranked number 1 on both the Top500 and HPCG lists. A64FX is a general-purpose processor based on the Armv8.2-A specification [10] and has 48 compute cores, divided into four core memory groups (CMGs) with 12 cores, and 2 – 4 cores dedicated to OS communications.

OpenMP is a directive-based standard for parallel programming on shared memory systems. Its ease of use makes OpenMP very attractive for obtaining efficient parallel versions of serial programs. The programmer can use compiler directives, library routines, and environment variables to write parallel programs for shared memory systems in Fortran and C/C++.

2 APPLICATIONS AND EXPERIMENTAL SETUP

2.1 List of Applications

- Minimod [4] - a seismic modeling mini-application that solves the acoustic wave equation. Minimod is used to study the performance of emerging compilers and runtimes for HPC. An OpenMP task-based version [8] is used in this paper.
- PENNANT [3] - a mesh physics mini-application for advanced architecture research, with the mesh size determined by input sets. PENNANT is dominated by pointer chasing. It can be run solely with MPI or in a hybrid MPI+OpenMP setup. It uses OpenMP's static loop-scheduling, and makes use of gather/scatter to send data to and from the root to other ranks, and reductions to consolidate partial results.
- SWIM [3] - a Fortran OpenMP weather forecasting program designed for testing current performance of supercomputers. Like PENNANT, SWIM also uses static scheduling in OpenMP loops.

[1] November, 2020, list of https://top500.org

[2] This work is in progress and here we present a partial list of results we so far have due to page limitations.

[3] https://www.spec.org/cpu2000/CFP2000/171.swim/docs/171.swim.html

PEARC '21, July 18–22, 2021, Boston, MA, USA

ACM ISBN 978-1-4503-8292-2/21/07…$15.00
https://doi.org/10.1145/3437359.3465592

2.2 Systems and Compilers

Fugaku. The world's fastest supercomputer, located at The RIKEN Center for Computational Science in Japan [9]. Its processor is the Fujitsu A64FX, and has a proprietary interconnect called Tofu, configured as a 6D torus. The cluster became operational in 2020 and enters production usage in 2021. Fugaku provides Fujitsu's native and cross-compilers alongside various versions of GNU compilers. Its underlying Tofu-D interconnect is not used in these experiments – only intranode performance is measured in our experiments. In addition, its stripped-down, lightweight, customized Linux kernel allows users to further enhance their applications' performance through the use of various environment variables not defined in standard Linux distributions.

Ookami. A new cluster installed at Stony Brook University (SBU) in late summer 2020. It has 174 compute nodes, with another 2 for debugging/experimentation. Ookami was funded through an NSF grant [5] as the first A64FX cluster outside of Japan. Ookami uses a non-blocking HDR 200 switching fabric via 9 40-port Mellanox Infiniband switches in a 2-level tree. Each node, similar to Fugaku's compute nodes, currently has 32GB of high-bandwidth memory. Our experiments do not use these switches in its intranode experiments. In addition, it runs a standard CentOS Linux distribution and its A64FX processors do not contain the extra cores that come in the "full" FX1000 variant of the chip. Table 1 shows the compilers used.

	Versions	
Compiler Family	**Fugaku**	**Ookami**
ARM	-	20.3
Cray	-	10.0.1
Fujitsu	4.3.0a	-
GCC	8.3.1, 10.2.1	8.3.1, 10.2.1, 11.0.0
LLVM	11.0.0	11.0.0, 12.0.0

Table 1: Compilers of Fugaku and Ookami.

2.3 Runtime Environment

Each benchmark was run on one compute node with one MPI rank to avoid shared memory operations that occur with two or more processes, and over-subscription of threads to cores, which might result in degraded performance. Threads are bound to cores using the `OMP_PLACES` environment variable with its semantics `start_core:num_cores`. This allows threads to be assigned to specific cores (e.g., Thread 0 is assigned to Core 0) as well as splitting threads among specific CMGs.

2.4 Compiler options

For each compiler mentioned in Section 2.2, we enabled specific flags, maximizing thread optimization and SVE instructions, and minimizing execution-time while maintaining correctness. We also enabled fine-tuning for the A64FX processor, where possible. The flags are listed for each compiler/group[4] as shown in Table 2.

[4]For any GNU and LLVM compiler: If compiling directly on an A64FX node, use `-mcpu=native` instead.

Compiler	Flags
Cray	`-homp -hvector3 -hthread3`
GCC	`-mcpu=a64fx` `-Ofast -fopenmp`
LLVM	`-mcpu=a64fx` `-Ofast -fopenmp`
Fujitsu-Traditional	`-Nnoclang -Nlibomp -O3` `-KSVE,fast,openmp,ARMV8_2_A`

Fujitsu-LLVM	`-Nclang -Nlibomp -Ofast`
	`-Kfast,openmp`
	`-mcpu=a64fx+sve`

Table 2: Flags used for each compiler.

3 EXPERIMENTAL RESULTS

We analyzed runtime and relative speed-up using OpenMP with different compiler classes – Cray, ARM, GNU, Fujitsu, and LLVM; and different versions among each class where applicable. Each set of results is drawn from running our programs on specified inputs five times per specified OpenMP thread count (one from 1, 2, 4, 8, 12, 16, 24, 32, 36, and 48) and obtaining the arithmetic mean for each set. Note that for each graph, the x-axis refers to the number of OpenMP threads, ranging from 1 to 48.

3.1 Ookami

For each application, we made graphs of the three compilers we deemed "best in class": of Ookami's compilers across the GCC, ARM, and Cray collections, binaries compiled from these gave the best runtime performance and speedup: GCC 10.2.0, ARM 20.1.3, and Cray 10.0.1, with the results for the other compilers explained in the following subsections.

3.1.1 PENNANT. For this application we focus on the **LeblancBig** input, which fits in the 32GB of on-chip memory while also being a non-trivial input size.

In Figure 1, we show the relative speedup observed between each of the three compilers, measured by the runtime T_{thread_value} compared to $T_{1\_OpenMP\_Thread}$. While the `armclang`-compiled code returns the slowest overall runtimes, it gives the most linear relative speed-up, with the Cray compilers having the smallest relative speed-up from being able to quickly saturate the on-chip memory bandwidth at the start of execution.

Profiling the **LeblancBig** input with CrayPat [2] and ARM Forge [1] on Ookami, we noticed that different values for `OMP_WAIT_POLICY` (active or passive) resulted in substantially different behaviors. An active wait policy resulted in PENNANT spending 66.3% of its runtime in OpenMP regions. Conversely, the passive wait policy results in only 17.8% of **LeblancBig**'s runtime inside OpenMP regions. The difference in time spent between computation and synchronization of threads is proportional to the requested thread count, with very little time (under 30%) used for thread synchronization.

3.1.2 SWIM. The default input for SWIM is `swim.ref.in`, which sets up a $7701x7701$ matrix running for 3000 iterations. In our experiments, we tested 7 different compiler versions, but to avoid

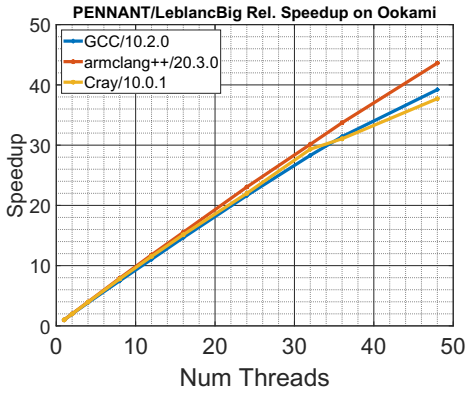

Figure 1: LeblancBig Speedup Comparison

clutter and data overlap, we have chosen 3 representatives from the various compiler families: GNU, ARM's LLVM-based compiler, and Cray. We present speed-up results as shown in Figure 2.

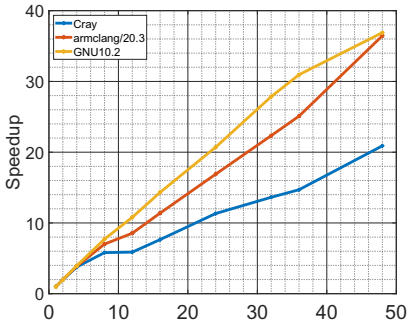

Figure 2: SWIM: speed-up results

As shown in Figure 2, we can see that among all of the compilers, the GNU compiler seems to have the greatest speed-up. 48 threads achieve a 37x speed-up over 1 thread. On the other hand, the Cray compiler has a much more moderate speed-up rate. The greatest speed-up is about 16 times between 1 and 48 OpenMP threads.

With profiling tools ARM MAP and CrayPat [2] on Ookami, SWIM spends 70.2% of its runtime on OpenMP regions, which is understandable since it is a purely OpenMP benchmark. OpenMP generates a small amount of overhead: 28.5% was seen with this particular run.

3.1.3 Minimod. Two different OpenMP configurations of Minimod (see [8] for details) were evaluated:

- Loop xy: Grid is blocked in x (largest-stride) and y dimensions. An OpenMP parallel for loop is applied to the 2-D loop nest over x-y blocks. (A `collapse(2)` is used to combine the two loops).
- Tasks xy: Grid is blocked in x and y dimensions. Each x-y block is a task using OpenMP's tasking directive. OpenMP's depend clause is used to manage dependencies between time steps.

In both cases a grid size of 512^3 was used. Minimod speedups are shown for the tasks-xy configuration in Figure 3. In general,

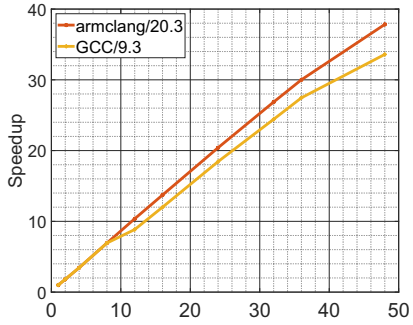

Figure 3: Minimod: speedup plot (tasks xy) for each compiler

for the task-based configuration, LLVM tends to outperform GCC (particularly at higher thread counts), although the total runtimes for this grid size are quite similar. In the loop-xy configuration, GCC performs slightly better than LLVM in terms of total runtime, and the speedups are similar between the two compilers.

Profiling using the Arm Forge Performance Report tool, we find that both configurations are entirely compute-bound, and both have a high number of stalled cycles (76.5% and 80.7% of cycles for loop-xy and tasks-xy configurations, respectively), indicating that the application is memory-bound. This makes the HBM2 memory of the A64FX processor potentially advantageous for these applications.

3.2 Fugaku

The Fujitsu compiler has two backends (traditional; LLVM), so we can compare performance and thread-scaling between them. In this section, we will break down and explain our results on Fugaku comparing results between the GNU and Fujitsu compilers. Per the experiments in Section 2, we ran each application 5 times and took the average of the runtimes.

3.2.1 PENNANT. The Fujitsu compiler gave the longest recorded runtimes for the **LeblancBig** input. In particular, the single-threaded runtimes for both inputs had surprisingly large standard deviations (107 seconds as opposed to a fraction of 1 second). Both versions of the GNU compilers on Fugaku were still competitive with Ookami. We noticed that the traditional back-end options for the Fujitsu compiler, compared to the LLVM-backend (see section 2.4), took substantially longer in smaller-threaded runs. Profiling **LeblancBig** shows that the traditional Fujitsu back-end generally results in a less efficient execution compared to the LLVM back-end. In particular, both back-ends are faster at 24 OpenMP threads (181 seconds with LLVM, 186 with traditional) than at 48 threads (233 and 236 seconds, respectively).

Similarly, this results in observed reduced speedup for the Fujitsu compiler, especially after reaching 12 threads placed in 1 CMG (see Figure 4)[5].

[5]While not the focus of our study, this observation would be resolved by utilizing PENNANT's MPI features in addition to OpenMP.

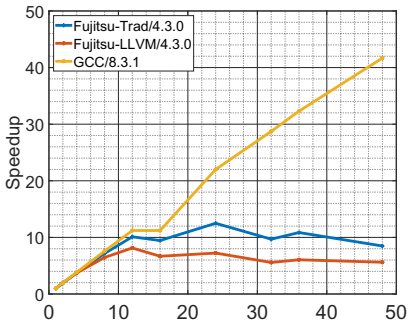

Figure 4: LeblancBig/Fugaku Speedup

3.2.2 SWIM. On Fugaku, the Fujitsu and GCC (10.2.1) compilers were used to test SWIM's capabilities. Compared to results reported in section 3.2.1, SWIM ran significantly faster when compiled with Fujitsu than with Cray on Ookami. As shown in Figure 5, the GNU compiler has a greater speed-up than Fujitsu. 36 threads achieve a 32x speed-up over 1 thread. The Fujitsu compiler only obtained a 25x speed-up with 36 threads over 1 thread. The Fugaku-based runs show a drop in relative speed-up starting at 8 OpenMP threads before leveling out at 12 threads, which could be explained by insufficient OpenMP optimization of the Fujitsu compiler.

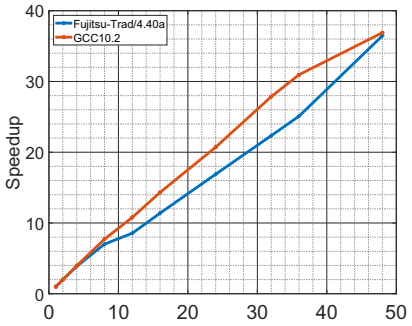

Figure 5: SWIM: Speedup Plot

On Fugaku, SWIM has shown a much better performance than all other applications tested previously on Ookami. It has achieved a performance of over 31 GFLOPS, which correlates well with the high SVE operation rate (99.9983%).

3.2.3 Minimod. Figure 6 shows the speedup result for the "loopxy" configuration of Minimod. The Fujitsu-Traditional compiler suffers a significant slowdown at higher thread counts, although the speedups for both compilers are significantly worse than the compilers evaluated on Ookami.

The Fujitsu-Trad compiler failed to run the tasks-xy configuration, presumably because of a lack of support for the OpenMP depend clause of the task directive. We are still working on evaluating with other compilers on Fugaku.

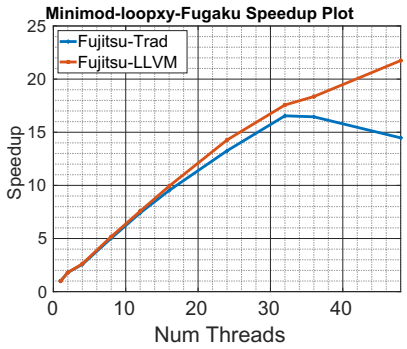

Figure 6: Minimod: speedup plot (loop xy) for each compiler on Fugaku

ACKNOWLEDGMENTS

We would like to thank the NSF for supporting the Ookami cluster, and the ability to research the Fujitsu A64FX processor, through grant OAC 1927880. We would like to thank the Riken Center for Computational Science for providing us with access to Fugaku and to conduct research on it. We would also like to thank Stony Brook University and the Institute for Advanced Computational Science for providing the resources to allow us to conduct our studies on Ookami.

REFERENCES

[1] ARM. [n.d.]. *ARM Forge Documentation.* https://developer.arm.com/documentation/101136/2021/Performance-Reports
[2] Hewlett Packard Enterprise. [n.d.]. *CrayPat Documentation.* https://pubs.cray.com/bundle/HPE_Performance_Analysis_Tools_User_Guide_S-8014_2012/page/CrayPat_Runtime_Environment.html
[3] Charles R. Ferenbaugh. [n.d.]. *PENNANT: An Unstructured Mesh Mini-App for Advanced Architecture Research.* https://www.osti.gov/biblio/1079561-pennant-unstructured-mesh-mini-app-advanced-architecture-research
[4] Jie Meng, Andreas Atle, Henri Calandra, and Mauricio Araya-Polo. 2020. Minimod: A Finite Difference solver for Seismic Modeling. arXiv:2007.06048 [cs.DC]
[5] NSF. [n.d.]. Ookami: A high-productivity path to frontiers of scientific discovery enabled by exascale system technologies. https://www.nsf.gov/awardsearch/showAward?AWD_ID=1927880.
[6] Ryohi Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. 2020. Supercomputer Fugaku CPU A64FX Realizing High Performance, High-Density Packaging, and Low Power Consumption. https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article03.html. *Fujitsu Rechnical Review* (Nov. 2020). https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article03.html
[7] OpenMP-ARB. [n.d.]. *OpenMP Website.* https://www.openmp.org
[8] Eric Raut, Jie Meng, Mauricio Araya-Polo, and Barbara Chapman. 2020. Evaluating Performance of OpenMP Tasks in a Seismic Stencil Application. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 67–81. https://doi.org/10.1007/978-3-030-58144-2_5
[9] RIKEN. [n.d.]. *Fugaku Project.* https://www.r-ccs.riken.jp/en/fugaku/project
[10] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and "Fugaku". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 47, 15 pages.