An Overview on Mixing MPI and OpenMP Dependent Tasking on A64FX

Romain Pereira, Adrien Roussel, Miwako Tsuji, Patrick Carribault, Mitsuhisa Sato, Hitoshi Murai, Thierry Gautier

An Overview on Mixing MPI and OpenMP Dependent Tasking on A64FX

Romain PEREIRA
Exascale Computing Research Laboratory
Bruyères-Le-Chatel, France

Adrien ROUSSEL
adrien.roussel@cea.fr
CEA, LIHPC, F-91297
Arpajon, France

Miwako TSUJI
miwako.tsuji@riken.jp
RIKEN Center for Computational Science
Kobe, Hyogo, Japan

Patrick CARRIBAULT
patrick.carribault@cea.fr
CEA, LIHPC, F-91297
Arpajon, France

Mitsuhisa SATO
msato@riken.jp
RIKEN Center for Computational Science
Kobe, Hyogo, Japan

Hitoshi MURAI
h-murai@riken.jp
RIKEN Center for Computational Science
Kobe, Hyogo, Japan

Thierry GAUTIER
thierry.gautier@inrialpes.fr
Avalon, LIP, ENS, Inria
Lyon, France

ABSTRACT

The adoption of ARM processor architectures is on the rise in the HPC ecosystem. Fugaku supercomputer is a homogeneous ARM-based machine, and is one among the most powerful machine in the world. In the programming world, dependent task-based programming models are gaining tractions due to their many advantages: dynamic load balancing, implicit expression of communication/-computation overlap, early-bird communication posting, ...MPI and OpenMP are two widespreads programming standards that make possible task-based programming at a distributed memory level. Despite its many advantages, mixed-use of the standard programming models using dependent tasks is still under-evaluated on large-scale machines.

In this paper, we provide an overview on mixing OpenMP dependent tasking model with MPI with the state-of-the-art software stack (GCC-13, Clang17, MPC-OMP). We provide the level of performances to expect by porting applications to such mixed-use of the standard on the Fugaku supercomputers, using two benchmarks (Cholesky, HPCCG) and a proxy-application (LULESH). We show that software stack, resource binding and communication progression mechanisms are factors that have a significant impact on performance. On distributed applications, performances reaches up to 80% of effiency for task-based applications like HPCCG. We also point-out a few areas of improvements in OpenMP runtimes.

CCS CONCEPTS

• **Computing methodologies Distributed programming languages**; • **Computing methodologies Parallel programming languages**;

HPCAsiaWS 2024, January 25–27, 2024, Nagoya, Japan

KEYWORDS

OpenMP, MPI, Task, Dependency, Graph, HPC

ACM Reference Format:
Romain PEREIRA, Adrien ROUSSEL, Miwako TSUJI, Patrick CARRIBAULT, Mitsuhisa SATO, Hitoshi MURAI, and Thierry GAUTIER. 2024. An Overview on Mixing MPI and OpenMP Dependent Tasking on A64FX. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCAsiaWS 2024), January 25–27, 2024, Nagoya, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3636480.3637094

1 INTRODUCTION

On the road toward the Exascale, supercomputer designs have evolved differently. Most of the time, heterogeneous architectures with GPU accelerators have been adopted. Indeed, GPU accelerators provide massively parallel hardware with good parallel and energy efficiencies. However, such architectures have raised new programming challenges that still need to be overcome. Among alternative solutions to GPU, ARM processor designs have been growing in interest for the HPC community for a decade [24]. In 2020, RIKEN in Japan unveiled the supercomputer Fugaku [26]: a homogeneous system based on ARM processors. Its A64FX processor reaches high performance, while Fugaku is amongst the most performant and green supercomputers. Recently, NVIDIA has also chosen ARM designs for its first 72-core Grace processor [9].

While supercomputer architectures are evolving, the way of programming them is as well. Task-based programming models fix many cores and heterogeneous programming issues in an asynchronous fashion. Thanks to dependent task abstractions, developers can represent their applications as a dependency graph, providing a partial order of execution that a run-time scheduler can dynamically unroll. In recent years, task-based programming models have demonstrated benefits in several scenarios. At the node level, they show significant improvement on load balancing concerns [4] and help to improve the use of hierarchical memories [19]. On a

larger scale, they allow seamless expression of communication/-computation overlap [25] thanks to task synchronizations through dependences and reduce idle periods with early-bird requests posting [11, 17]. Dependent task-based programming has been adopted in OpenMP since its specifications 4.0 [8]. MPI is the *de-facto* standard for distributed memory programming. It has already been demonstrated that coupling MPI with OpenMP is challenging, especially in a task-based programming context [23, 25, 28]. This challenge may also vary depending on the supercomputer used.

Despite its many advantages, mixed-use of the standard programming models using dependent tasks is still under-evaluated on large-scale machines. As experienced in [19], task-based applications are sensitive to hardware and execution environments. In this previous work, authors only focused on very similar architectures and coupling Open MPI with MPC-OpenMP [5] runtime systems. With the arrival of ARM architectures in HPC systems like Fugaku, a study on the software stack's impact on performance needs to be performed. For this purpose, we propose an overview by experimenting with several MPI and OpenMP implementations on the Fugaku supercomputer. The contributions of this paper are:

(1) Evaluating task-based applications performances at node-level with different execution environments,
(2) Studying resources binding and performance scaling on distributed executions,
(3) Arguing on the impact of ARM architectures for task-based parallel applications.

We organize the paper as follows. Section 2 reviews related work already achieved in this area. Then, section 3 presents the different execution environments, understudy. Section 4 reports performances at node-level parallelism, while Section 5 focuses on distributed execution. Section 6 report technical issues encountered during our experiments that partly remain to be addressed. Eventually, we conclude this work in Section 7.

2 RELATED WORKS

Profiling Task-based Execution Performances. N. R. Tallent [29] proposed a breakdown of the execution time of parallel applications: the work, the idleness, and the overhead. It enables performance characterization of multithreaded executions to balance the parallelism (idleness) with management costs (overheads). In this work, we use this time breakdown adapted in the context of dependent task-based parallelism through MPC profiling tools [20]. It allows performance characterization using per-task hardware events counter mixing the Portable Application Profiling Interface (PAPI) [21] with the OpenMP Tools Interface (OMPT).

Evaluating OpenMP and MPI on A64FX. In 2020, T. Odajima et al. [15] evaluated several benchmarks (including LULESH) on A64FX processors, where each benchmark uses the OpenMP `pragma omp parallel for` construct (static workshare). In 2021, A. Poenaru et al. [22] benchmarked the A64FX processor. They compare it to more traditional processors of the HPC industry, such as Intel Skylake or AMD Rome processors, and conclude that A64FX's high-memory bandwidth allows it to outperform them on memory-bound benchmarks (BabelStream). In 2021, B. Michalowicz et al. [14] compared GCC, LLVM, and Fujitsu compilers on three OpenMP mini-applications (LeblancBig, Minimod, and SWIM) on A64FX, all using the static workshare construct as well. In [13], authors evaluated the SPEC CPU and SPEC OMP benchmark suites: their conclusions showed that A64FX performs lower than Xeon CPU with the same numbers of cores on the int and fp benchmarks. However, A64FX sometimes outperforms Xeon CPU due to its HBM. All this previous work relies on loop-level parallelism using the `pragma omp parallel for` construct. In our work, we studied data-flow parallelism using OpenMP tasks mixed with MPI requests for distributed execution.

3 EXECUTION ENVIRONMENTS

Our performance survey was executed on the supercomputer Fugaku. We start to present the architecture of this supercomputer, and we focus on the A64FX processor design. Supercomputer architecture has a major role in task-based performance study, but software stack as well: compilers, runtimes, and applications. We then present versions used on each software in the next section.

3.1 A64FX Compute Nodes

The A64FX [27] is a 64-bit ARM architecture many-core processor. Fig. 1 presents its specification (left-side table, retrieved from the processor manual [10]) and an overview of its architecture (right-side figure). It is a 52-core processor, where processors are packed in *Core Memory Groups* (i.e., CMGs) of 13 cores (12 cores usable by programmers, and 1 assistant core). Each core has 64KB of data and 64KB of instruction L1 cache. Each CMG has an L2 cache of 8MB connected to 8GB of High-Bandwidth Memory (HBM2); there is no L3 cache. CMGs of the same processor are connected with a network on the chip, and Fugaku interconnects multiple A64FX processors using TofuD interfaces. The default clock frequency that we used throughout this paper is 2.0GHz.

3.2 Runtimes and Compilers

Our work is based on the use of programming standards like OpenMP and MPI. Such programming standards have various implementations, depending on the compiler or execution environment used. Here, we present the software stack we used on Fugaku for our performance study.

Compilers. In this study, we compare the GNU Compiler Collection (GCC [2]) and the Low-Level Virtual Machine (LLVM [3]) C compilers. We always report the best performances obtained with any versions, using optimizations flags `-march=armv8.2-a+sve`, `-msve-vector-bits=512`, `-mtune=a64fx`, or `-O3`.

OpenMP. GCC [2] and LLVM [3] implement the OpenMP standard specifications. They both extend their own C compiler to support OpenMP-specific interfaces, coupled with a run-time library through an Application Binary Interface (ABI). We evaluate the most recent releases of GCC and LLVM that were retrieved and built on Fugaku: LLVM release/17.x and GCC releases/gcc-13. In addition, we also evaluate the Multi-Processor Computing (MPC [16]) OpenMP run-time library (MPC-OMP [5]), which is compatible with both LLVM and GCC OpenMP ABI: it allows us in later evaluations to compare only GCC and LLVM compiler performances under the same OpenMP run-time library (MPC-OMP).

A64FX Processor Specifications

	Specification
Number of processor cores	52 (13 cores / CMG) (12 visible core with 1 assistant core / CMG)
Number of CMGs	4 (with 8GB of HBM2 / CMG at 1,024GB/s)
L1I cache size	64 KiB / 4-way
L1D cache size	64 KiB / 4-way
L2 cache size	32 MiB / 16-way (8 MiB / CMG) (implemented per CMG)
Cache-line size	256 bytes
Memory controller	4 (1 MAC / CMG)
Interconnect	Tofu-D
I/O	PCI-Express Gen3 16 Lanes
Instruction set architecture	ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 (*1), SVE
SVE-implemented Vector Length	128 / 256 / 512 bits

(*1) ARMv8.3 supports only complex-number supported instructions.

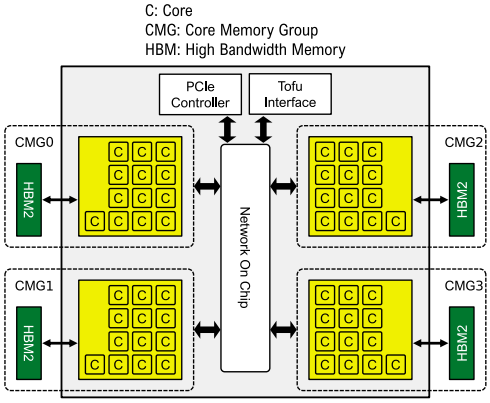

Figure 1: A64FX Node Specifications and Overview (from [10])

MPI. In all our experiments, the MPI run-time library is unchanged: we always use the MPICH-Tofu [1] implementation with support for `MPI_THREAD_MULTIPLE` (`mpich-tofu@1.0/epbdicy`) installed on the supercomputer Fugaku. In our mixed-use of the two standards, communication calls (`MPI_Send`, `MPI_Allreduce`, ...) are executed as part of OpenMP tasks scheduled by the OpenMP task scheduler. MPI requests progression and OpenMP task completion are compared using two state-of-the-art techniques:

- `MPI_Detach` (later referenced as `mpi-detach`) consists of a dedicated kernel thread oversubscribing physical cores implemented by J. Protze et al. [23] that periodically progresses pending requests with `MPI_Test`.
- `OpenMP Tools` (later referenced as `mpc-ompt`) consists in opportunistically progressing pending requests with `MPI_Test` in-between OpenMP scheduling points. It mimics the behavior of *ompt_callback_task_schedule* callback, but the progression mechanism is not based on the OMPT interface. This progression mechanism is only compatible with the MPC' OpenMP runtime [18].

Additionally, in [18], authors proposed a scheduling strategy to favor the early-bird posting of MPI requests by marking associated tasks with OpenMP `priority` clause. Their heuristic (`priority`) had only been implemented into the MPC's OpenMP runtime, which we compare with the default heuristic (`no-priority`).

3.3 Applications

We conduct performance evaluations on a task-based version of three mainstream HPC benchmarks: the Cholesky tiled decomposition, the High-Performance Conjugate Gradient (HPCCG [7]), and the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH [12]). We introduce each application characteristics so HPC users can extrapolate our results to their needs.

Cholesky. The Cholesky benchmark computes the decomposition $A = L.L^T$ of a Hermitian positive-definite real matrix A. We retrieved the version implemented by J. Schuchart [28] that parallelized computation nesting BLAS/LAPACK kernels into OpenMP tasks (`netlib-lapack@3.10.1/selikac` on Fugaku) and in distributed memory by partitioning the matrix tiles in a cyclic-manner on a group of MPI processes. The matrix A is dense and made of `double` precision floating-point, built upon tiles of configurable size defining the computational tasks' granularity. Hence, this benchmark is representative of dense linear algebra codes where operations can be pipelined using tasks and dependencies.

HPCCG. The HPCCG benchmark resolves the linear system $A.x = b$ implementing a conjugate gradient (CG) algorithm, where A is a sparse matrix and b a dense vector, made of double precision floating-point. The CG algorithm consists of a series of three operations: matrix/vector multiplication (SPMV), dot products (dot), and scaled vector addition (axpby). In the task-based version used, each operation is programmed with `for` loops that can be decomposed into multiple tasks with a configurable parameter defining task granularity. This benchmark is representative of applications whose execution is typically bound by memory access time.

LULESH. The LULESH proxy-application models a hydrodynamic simulation of materials motion subject to forces over an unstructured mesh. It consists of a series of mesh-wide loops performing computation on mesh nodes or elements. We retrieved the task-based version of LULESH proposed by R. Pereira [1]. It defines a single grain parameter as the number of tasks sub-decomposing such a loop: we refer to it as the *tasks per loop* (TPL) parameter. Additionally, we added support for MPI using the `MPI_Detach` proposal [23]. Even though the proxy application remains simple, it is a fairly good representative of HPC production simulation codes with irregular computation schemes.

3.4 Summary

Table 1 summarizes the hardware, compilers, OpenMP/MPI runtimes, and applications considered in this paper.

4 SHARED MEMORY PERFORMANCES

Our performance overview crosses software configurations and starts with shared-memory evaluations on a single A64FX node. For each application, we first study the impact of their task granularity parameter on 12-threads bound 1:1 with physical cores (1 CMG);

[1] https://github.com/rpereira-dev/LULESH

Hardware	A64FX and TofuD Interconnect (Fugaku)
Compiler	LLVM/release/17.x (clang17), GCC/releases-13 (gcc)
OpenMP Runtime	LLVM/release/17.x (kmp), GCC/releases-13 (gomp), MPC (mpc)
MPI Runtime	MPICH-Tofu
MPI Request Progression Engine	MPI-Detach thread (mpi-detach), MPC-OMPT (mpc-ompt)
MPI Request Task Scheduling	Default (no-priority), Early-bird Posting (priority)
Applications	Cholesky, HPCCG, LULESH

Table 1: Hardware and Software Stack Understudy

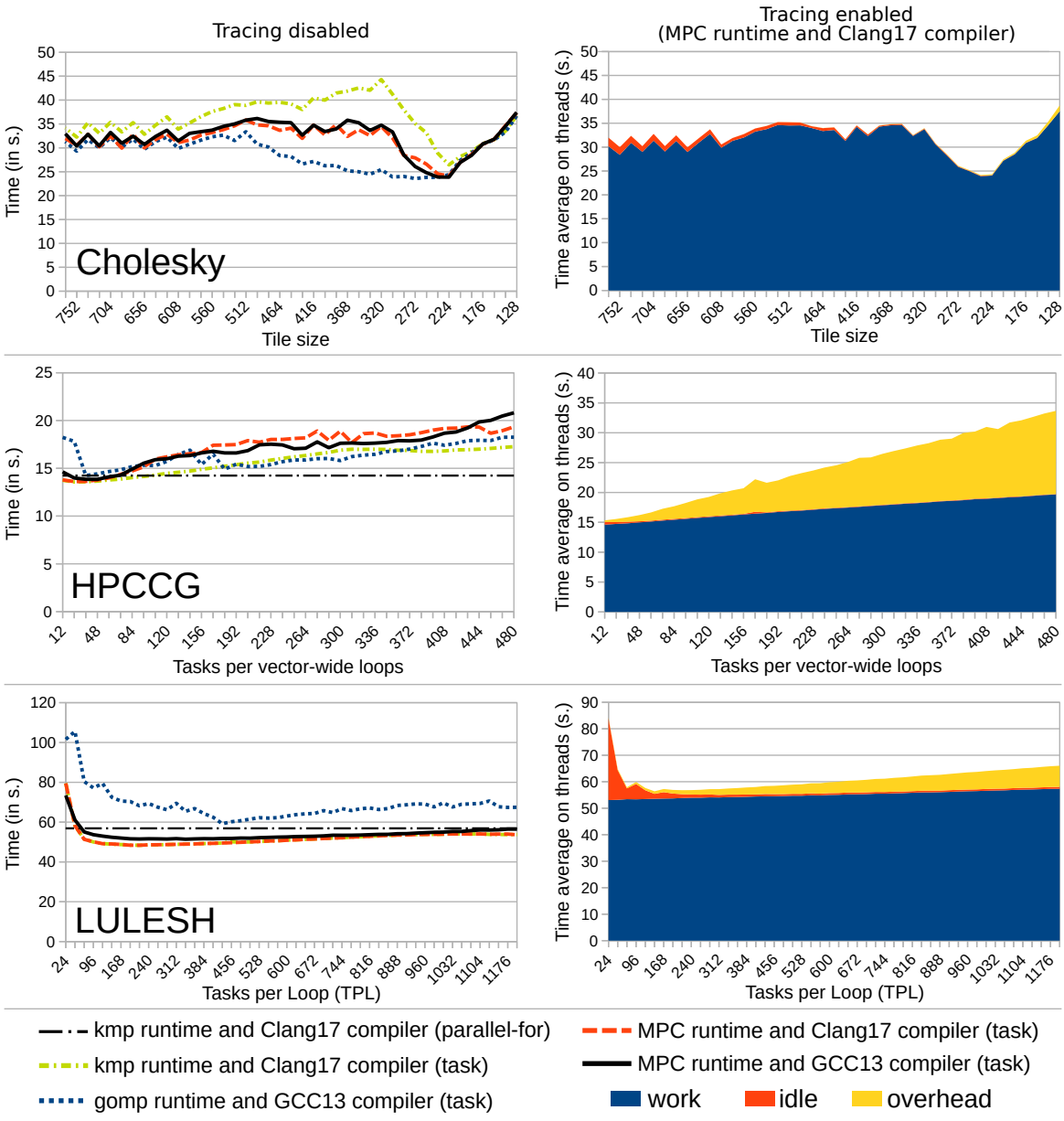

Figure 2: Grain study with 12 threads compactly bound on A64FX CMG0.

we then perform strong-scalability evaluations from 1 thread to 48 threads compactly bound to CMGs.

4.1 The Impact of Tasks Granularity

Fig. 2 is a study on the impact of task granularity on performances for the three applications. For each sub-figure, the X-axis varies grain parameters: there are few/coarse tasks on the left-most points and many/fine tasks on the right-most points. Y-axes represent the execution time. Left and right-side figures are respectively non-traced and traced execution (using MPC runtime with Clang17 compiler) on a single execution per point. Traced performances show a work/overhead/idle time breakdown cumulated and averaged on threads: the work section is the time spent executing explicit task instructions; the overhead section is the time spent executing non-explicit task instructions while there is an explicit task that would be ready for schedule; and the idle section is the time spent executing non-explicit tasks instructions and there is no ready-task for schedule anyway. There is 1 MPI process of 12 threads compactly bound to the CMG0, and applications problem sizes from 60% to 70% of the CMG0 8GB High-Bandwidth (HBM). It corresponds to a matrix of size $n = 16,384$ for Cholesky, a triplet $(s_x, s_y, s_z) = (216, 216, 216)$ for HPCCG, and a mesh size $s = 160$ for LULESH. We present and analyze results per application.

Cholesky. There are only a few coarse tasks on the left-most points, and performances are poor (about 30 to 35 s.). We observe that coarse-grain performances are worst for MPC and KMP OpenMP runtimes over GOMP, most likely due to task scheduling policies. Refining tasks, tile size in the range $[176; 320]$ provides the best performances (about 24s.). Refining furthermore only degrades performances for any compiler/runtimes. As seen on the right-side traced execution, performance variations are mostly due to work time inflations, most likely related to optimal LAPACK kernels tile size used by tasks. Finally, results using the MPC-OMP runtime show that the compiler (GCC13 or Clang17) does not impact performances.

HPCCG. On the left-most points, there are only a few coarse tasks for which the benchmark reaches the best performances. In this granularity, performances are similar to the original `parallel-for` version of the benchmark that uses no tasks. Refining the grain size furthermore, tasking overheads bound the execution time, and we also observe a slight work time inflation.

LULESH. Refining the grain size improves performances with diminishing idleness, and the task-based version under MPC and KMP OpenMP runtimes even outperforms the original `parallel-for` version for $TPL = 168$. We explain these gains thanks to better load balancing due to task dependencies synchronizations over the original implementation that implies synchronization barriers after each loop. Hence, work-stealing scheduling strategies implemented in MPC and KMP reduce idleness on threads and can execute workload earlier than the `parallel for` version could. However, GOMP runtime performances remain lower than the `parallel for` version for every task grain. While the exact reasons remain to be further investigated, we suspect the issue to be throttling mechanisms implemented in GCC that limit the parallelism visible to the runtime scheduler.

4.2 Strong Scalability on a Single Node

Fig. 3 is a strong-scalability study over A64FX 48-cores for each application. For each sub-figure, the X-axis varies the number of cores (on which threads are compactly bound 1:1). Left-side tables show wall-clock execution time (in s.) on 1, 12, 24, 32 and 48 cores. In the middle figures, the Y-axis represents efficiency built upon the execution time using single-core as a reference. On the right-side figures, the Y-axis represents the time breakdown cumulated on cores. There is only 1 MPI process for each application, and the problem sizes used are the same as of the previous granularity study: it occupies 60%-70% of the CMG0 memory, allocated and first-touched by the first core of the CMG0. We tuned each application task grain with respect to previous experiments so it provides the best performances for the 12-core (CMG0) configuration. We present and analyze results per application.

Cholesky. For any OpenMP runtime, efficiency remains high (above 60%) but the MPC runtime shows the best scaling with above 90% all the time. One reason could be the default scheduling heuristic of MPC that favors the execution of successors (in terms of task dependencies) on the same cores as per their predecessors, which could improve data temporal locality over KMP/GOMP that have no such heuristics.

In addition, we also observe that work time inflates from 1 to 12 cores, as there is more business on the CMG0. However, when the 13th thread is allocated and bound on the CMG1, even though the matrix is fully allocated on the CMG0, we observe a significant work time deflation that leads to improved efficiency. We collected hardware counters for each explicit task using MPC tracing capabilities coupled with the Portable Application Profiling Interface (PAPI) [21]. We observed a correlation between the number of `PAPI_RES_STL` events ("*Cycles processor is stalled on resource*") and the work time inflation. While we do not come with the exact resource that is causing stalls, it does not seem related to the memory controller as L1/L2 caches miss, Translation Lookaside Buffer (TLB) misses, and `PAPI_MEM_SCY` ("*Cycles Stalled Waiting for Memory Access*") events occurrences had no such inflations.

HPCCG. For any OpenMP version but GOMP with tasks, performance efficiency degrades linearly until around 38 cores (full CMG[0,1,2], and 2 cores on CMG3). Tracing with MPC, we observe linearly growing idleness, as the number of workers keeps increasing while the available parallelism remains constant going left to right on the figure. We also observe a work time and memory-related hardware events inflations (`PAPI_RES_STL` and `PAPI_MEM_SCY`): HPCCG is well-known for being memory-bound, which is observed on this work time inflation when increasing the number of threads. Above 38 cores, these phenomena are getting exacerbated with significant performance degradation. The increasing overheads also suggest that many threads create contention on the data structures shared by the runtime threads (tasks, queues...), slowing down the overall scheduling decisions.

LULESH. For any OpenMP version but GOMP with tasks, performance efficiency degrades linearly from 1 to 48 cores due to (1) slight work time inflation related to memory access time (measured with `PAPI_MEM_SCY`), (2) more overheads and idleness due to an

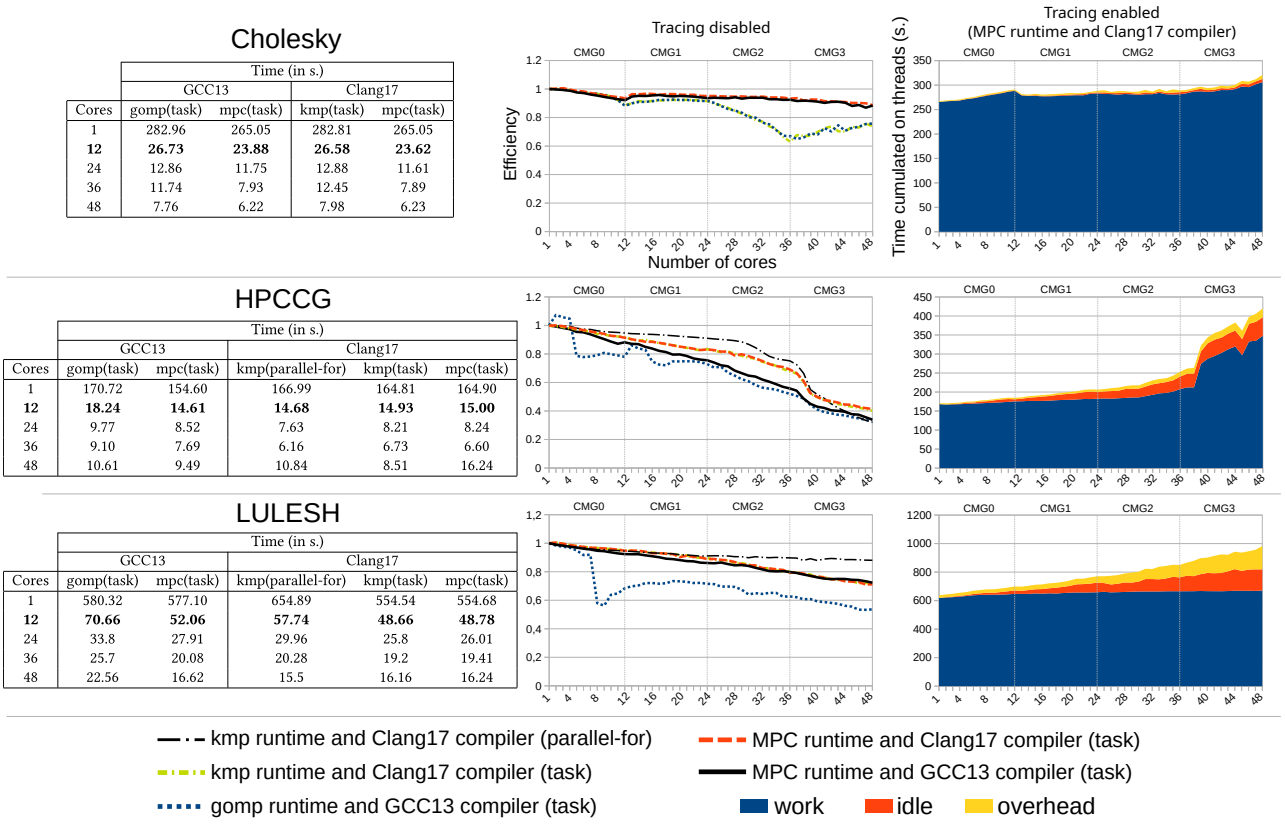

Figure 3: Strong scaling compactly bounding 1 to 48 threads on A64FX. Tables are execution time with tracing disabled.

increase of workers. For GOMP, we observe an unexplained performance efficiency irregularity at 8 cores. The high efficiency of the reference `parallel for` is due mainly to its poor performance on 1-core against task-based versions (654.59s. against 554.68s.).

5 DISTRIBUTED MEMORY PERFORMANCES

In this section, we focus on multi-processes execution on A64FX using MPI. Our first experiment evaluates the repartition of cores between MPI processes and OpenMP threads to find optimal bindings. Then, we perform a weak and strong scaling on the supercomputer Fugaku up to 32 interconnected A64FX. In terms of methodology, we allocate a small set of Fugaku's A64FX compute nodes and report worst/best/median performances over 5 executions.

5.1 A Study on Threads Repartition

Fig. 4 is a study of core allocation between MPI processes and OpenMP threads for Cholesky and HPCCG. Cores are allocated compactly. For instance, `1-48` means there is only one MPI process per A64FX (of 48 OpenMP threads), and `4-12` means there is one MPI process per CMG (of 12 OpenMP threads). We evaluate 6 configurations using software presented in Section 3.4, always compiling the source code with Clang 17.x:

- `(kmp, parallel-for)` runs `parallel for` versions using KMP 17.x runtime.
- `(kmp, task, mpi-detach, no-priority)` runs task-based versions using KMP 17.x runtime, the `MPI_Detach` progression thread, and KMP' scheduler.
- `(mpc, task, mpi-detach, no-priority)` runs task-based versions using the MPC-OMP runtime, the `MPI_Detach` progression thread, and MPC' default scheduler.
- `(mpc, task, mpi-detach, priority)` runs task-based versions using the MPC-OMP runtime, the `MPI_Detach` progression thread, and favoring early-bird requests posting.
- `(mpc, task, mpc-ompt, no-priority)` runs task-based versions using the MPC-OMP runtime, OpenMP Tool for requests progression, and MPC's default scheduler.
- `(mpc, task, mpc-ompt, priority)` runs task-based versions using the MPC-OMP runtime, OpenMP Tool for requests progression, and favoring early-bird requests posting.

First, we observe a performance gap between the `1-48` and other repartitions for both applications. While we expected the `4-12` configuration (one MPI process per CMG) to outperform other repartition, the `2-24` one also provides a high level of performances that could be interesting to dampen the number of MPI processes when scaling on Fugaku.

Secondly, we observe a significant slowdown whenever mixing the `MPI_Detach` progression thread with the MPC-OMP runtime. The MPC-OMP runtime relies on MPC low-level threading library, and we expect this performance issue to come from interference

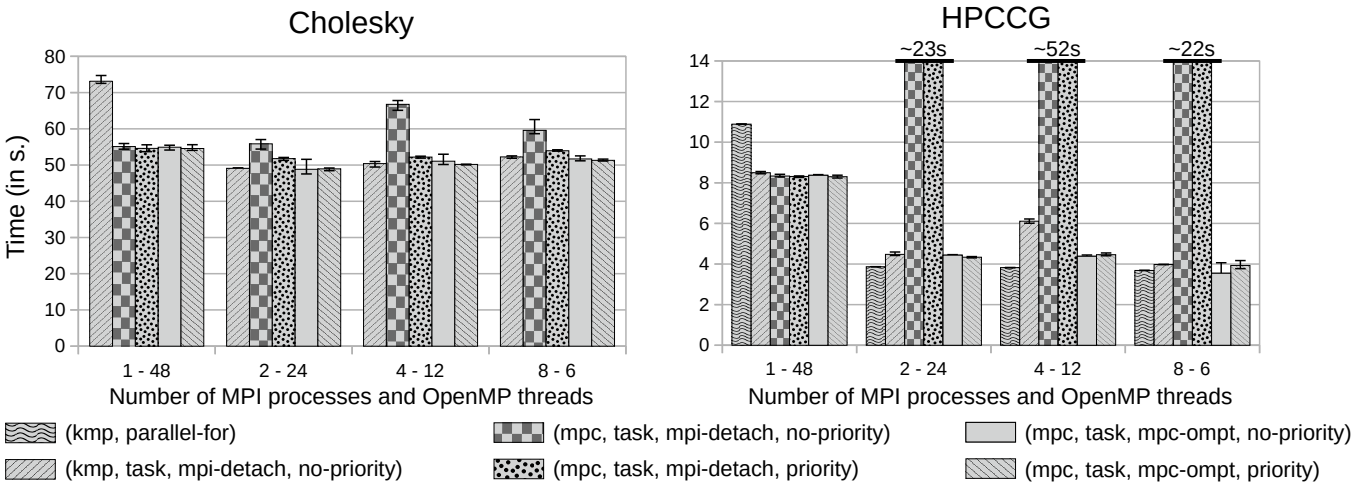

Figure 4: Study of core allocation (compact) between MPI processes and OpenMP threads for Cholesky and HPCCG on A64FX.

related to the co-existence of multiple threading libraries (pthread and MPC threads). Using the MPC's OpenMP tool (mpc-ompt) over the dedicated (p)thread (mpi-detach), MPI requests are opportunistically progressed in-between scheduling points without spawning a new kernel (p)thread which removes previously observed interferences.

Finally, regarding scheduling heuristics, early-bird posting of MPI communications does not significantly impact performances on Cholesky and HPCCG, most likely because we are running on a single A64FX node where inter-process synchronization times are already short. However, it seems to dampen performance degradation observed previously when using the `MPI_Detach` thread with MPC-OMP runtime.

In the next section, we conduct evaluations executing on distributed A64FX processors interconnected via TofuD. From now on, we always use the `4-12` repartition that is compactly binding processes of 12 threads on CMGs.

5.2 Scalability on a Cluster of A64FX

Fig. 5 reports a weak and strong scaling for Cholesky and HPCCG from 1 to 128 MPI Processes of 12 OpenMP threads compactly bound to A64FX' CMGs (that is 1 to 32 A64FX nodes). Both applications had been executed with the same 6 configurations as before. The left-most points are performance references obtained on a single process, and correspond to the 12 threads execution of Fig. 3. We scale weakly and strongly up to 128 MPI processes of 12 threads.

On both the strong and weak scaling, coupling the MPC-OMP runtime with the `MPI_Detach` progression thread only leads to poor performances due to threading libraries co-existence. Hence, in the following paragraphs, we only focus on other configurations.

Strong Scalability. Task-based versions of both applications have good strong scalability (>80% efficiency) up to 8 MPI processes on 2x A64FX, but performance degrades above. HPCCG performance degradation most likely comes from tasks becoming too fine, leading to important management overhead, as observed previously in the grain study. On Cholesky, task grain remains constant, but each MPI process ends up with less work while the amount of communications on-the-fly increases, leading to efficiency degradation.

Weak Scalability. HPCCG's weak scalability is a lot better (>80% We suspect it comes from the application communication pattern and MPI runtime management of many concurrent requests on the fly. HPCCG only executes a few point-to-point requests with its topological neighbors: for 32 MPI Processes, each process communicates on 1 or 2 neighbors. On the other hand, Cholesky tends to post many more concurrent point-to-point requests and to remote nodes: for 32 MPI Processes, each process communicates with 10 others.

6 TECHNICAL DIFFICULTIES REPORT

On our journey to evaluating the mixed use of MPI and OpenMP using dependent tasks, we encountered several technical difficulties with LULESH that still need to be addressed.

Vectorizing Irregular Applications. We profiled an execution of LULESH compiled with Clang17 using the Modular Assembler Quality Analyzer and Optimizer (MAQAO) [6]. The few loops with regular access patterns successfully took advantage of vectorized instructions. However, most of the workload has irregular memory accesses that had not been vectorized. Hence, the application cannot take full advantage of the SVE512 instructions of the A64FX processor.

GOMP Performances. The GOMP runtime seems to have a performance issue related to the task dependency graph construction. Fig. 6 shows the median execution time on 10 instances (y-axis) of LULESH, for a size `-s 50` and 12 tasks per loop, on a single MPI process of 12 threads bound to the CMG0, varying the number of simulation iterations (x-axis) In this specific configuration, we observe that using the KMP and MPC-OMP runtime, the execution time grows linearly with the number of iterations, that is, the expected behavior, as the amount of work grows linearly with the number of iterations. However, with the GOMP runtime,

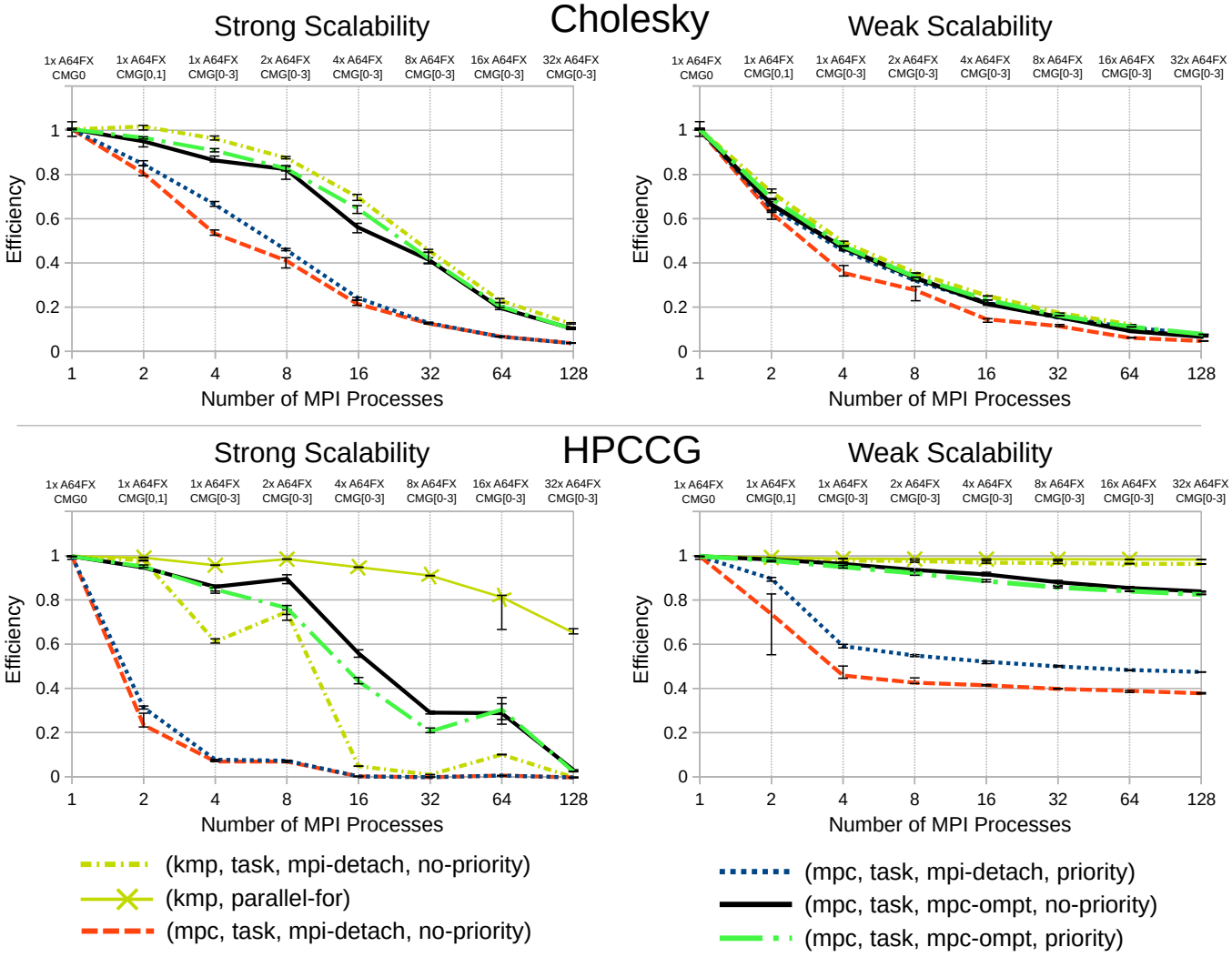

Figure 5: Scalability Study on multiple A64FX Nodes

the execution time grows with a quadratic shape. Preliminary investigations with GDB suggest that the GOMP runtime could be creating many unnecessary dependencies between tasks whose correct order of execution is already granted by other edges. Hence, the single-producer thread is significantly slowed down when constructing the task dependency graph and ends up bounding the total execution time.

Distributing Execution. Distributing the proxy-application LULESH mostly ends with deadlocks or crashes. Hence, we could not record performances for LULESH for distributed executions. We performed preliminary investigations and reported the following issues in the OpenMP runtimes.

- With the GOMP runtime, an assertion related to the `detach` clause fails, making us believe in a race condition in the clause implementation.
- With the KMP runtime, the application sometimes ends up deadlocking.

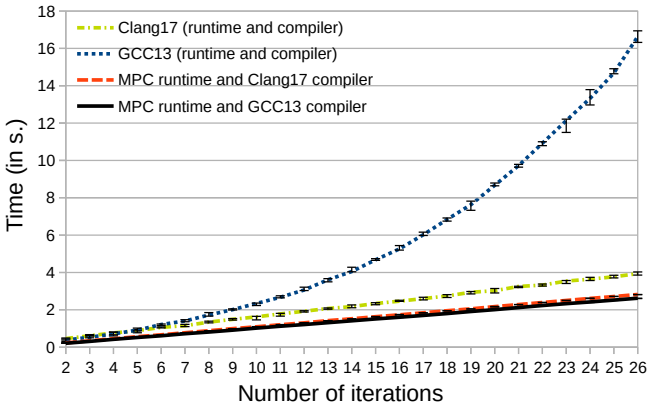

Figure 6: LULESH performances over iterations

- With the MPC runtime, the OpenMP standard version of the application does not compile due to missing ABI, even though a non-standard (MPC-OMP specific) version does compile and execute correctly.

7 CONCLUSION AND FUTURE WORK

ARM supercomputers are growing in interest. While processor architectures should have almost no impact on how codes are programmed, it impacts their performances at run-time. In this paper, we evaluated the performances of mixing MPI and OpenMP using the dependent tasking model on Fugaku, an ARM-based supercomputer at RIKEN. Our evaluations include a study crossing a relevant set of software used in HPC: compilers (GCC13, Clang17), runtime systems (GOMP, KMP, MPC-OMP), and applications (Cholesky, HPCCG, LULESH).

At the node level, the study shows that any combination of compiler/runtime provides similar peak performances on Cholesky and HPCCG. Though on LULESH specifically, using the GCC/GOMP runtime over Clang17/KMP or MPC-OMP leads to a 20% performance loss. We also conducted a study on up to 32 A64FX nodes enabling MPI. We showed that binding a single MPI process on two CMGs (instead of every CMGs usually) does not deteriorate performances, and could be a way of dampening the number of processes at extreme scale. We also showed that communication progression mechanisms can have a considerable impact: in particular, when extending the MPC-OMP runtime system with a dedicated progression thread (using `MPI_Detach` implementation [23]). Clang17/KMP and MPC-OMP runtimes provide the same level of scalability for task-based versions of Cholesky and HPCCG. However, the non-task-based (`parallel for`) version of HPCCG has better strong scalability than its task-based version: as shown in the grain study, this most likely comes from tasking overheads as tasks are getting too fine. Weak scalability remains high, with above 80% of efficiency for any task-based version.

We identified a few issues in Section 6 that we would like to investigate further. We reported performance problems with the GCC/GOMP executions. We also experienced deadlocks and crashes when executing the standard task-based LULESH on distributed environments with any OpenMP runtime. As a perspective, we will investigate these issues to enhance application performance.

ACKNOWLEDGMENTS

We would like to thank people at the RIKEN Center for Computational Science (R-CCS) that granting access to, and support on the supercomputer Fugaku for the purpose of this research.

REFERENCES

[1] 2023. An Overview of RIKEN MPI (MPICH-Tofu). https://www.r-ccs.riken.jp/wp/wp-content/uploads/2021/01/MPICH-Tofu.pdf. [Online; accessed 13-November-2023].

[2] 2023. GCC Wiki - OpenMP. https://gcc.gnu.org/wiki/openmp. [Online; accessed 30-March-2023].

[3] 2023. LLVM/OpenMP Documentation. https://openmp.llvm.org/. [Online; accessed 30-March-2023].

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://doi.org/10.1002/cpe.1631

[5] Patrick Carribault, Marc Pérache, and Hervé Jourdren. 2010. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*, Mitsuhisa Sato, Toshihiro Hanawa, MatthiasS. Müller, BarbaraM. Chapman, and BronisR. de Supinski (Eds.). Lecture Notes in Computer Science, Vol. 6132. Springer Berlin Heidelberg, 1–14. https://doi.org/10.1007/978-3-642-13217-9_1

[6] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. 2005. MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. (03 2005).

[7] Jack Dongarra, Michael Heroux, and Piotr Luszczek. 2015. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications* 30 (08 2015). https://doi.org/10.1177/1094342015593158

[8] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. 2008. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. de Supinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–122.

[9] Anne C. Elster and Tor A. Haugdahl. 2022. Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering* 24, 2 (2022), 95–100. https://doi.org/10.1109/MCSE.2022.3163817

[10] Fujitsu. 2022. *A64FX® Microarchitecture Manual (English).* https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.8.pdf

[11] Ryan Grant, Matthew Dosanjh, Michael Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. 330–350. https://doi.org/10.1007/978-3-030-20656-7_17

[12] Ian Karlin, Jeff Keasler, and Neely Rob. 2013. LULESH 2.0 Updates and Changes (Technical Report). https://www.osti.gov/biblio/1090032

[13] Yuetsu Kodama, Masaaki Kondo, and Mitsuhisa Sato. 2021. Evaluation of SPEC CPU and SPEC OMP on the A64FX. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 553–561. https://doi.org/10.1109/Cluster48925.2021.00088

[14] 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* (Boston, MA, USA) *(PEARC '21)*. Association for Computing Machinery, New York, NY, USA, Article 28, 4 pages. https://doi.org/10.1145/3437359.3465592

[15] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhisa Sato. 2020. Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 523–530. https://doi.org/10.1109/CLUSTER49012.2020.00075

[16] Marc Perache, Herve Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-par Conference on Parallel Processing* (Las Palmas de Gran Canaria, Spain) *(Euro-Par '08)*. Spinger-Verlag, Berlin, Heidelberg, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9

[17] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. 2021. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In *OpenMP: Enabling Massive Node-Level Parallelism*, Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 197–210.

[18] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. 2021. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In *IWOMP 2021 - 17th International Workshop on OpenMP (OpenMP : Enabling Massive Node-Level Parallelism (IWOMP 2021))*. Bristol, United Kingdom, 1–15. https://doi.org/10.1007/978-3-030-85262-7_14

[19] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. 2023. Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances. In *52nd International Conference on Parallel Processing (ICPP 2023)*, Vol. 52nd International Conference on Parallel Processing (ICPP 2023). Salt Lake City, United States. https://doi.org/10.1145/3605573.3605602

[20] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. 2023. Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances. In *52nd International Conference on Parallel Processing (ICPP 2023)*, Vol. 52nd International Conference on Parallel Processing (ICPP 2023). Salt Lake City, United States. https://doi.org/10.1145/3605573.3605602

[21] Te Phhh, Shirley Moore, Jack Dongarra, N. Garner, K. London, and Phil Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications* 14 (07 2000).

[22] Andrei Poenaru, Tom Deakin, Simon N McIntosh-Smith, Simon D Hammond, and Andrew J Younge. 2021. An Evaluation of the Fujitsu A64FX for HPC Applications. In *Cray User Group 2021*. https://cug.org/cug-2021/ Cray User Group 2021 ; Conference date: 03-05-2021 Through 05-05-2021.

[23] Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S. Müller, and Torsten Kuhlen. 2020. MPI Detach - Asynchronous Local Completion. In *Proceedings of the 27th European MPI Users' Group Meeting* (Austin, TX, USA) *(EuroMPI/USA '20)*. Association for Computing Machinery, New York, NY, USA,

71–80. https://doi.org/10.1145/3416315.3416323

[24] Nikola Rajovic, Paul M. Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. 2013. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. https://doi.org/10.1145/2503210.2503281

[25] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. 2019. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Comput.* 85 (2019), 153–166. https://doi.org/10.1016/j.parco.2018.12.008

[26] Mitsuhisa Sato. 2020. The Supercomputer "Fugaku" and Arm-SVE enabled A64FX processor for energy-efficiency and sustained application performance. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. 1–5. https://doi.org/10.1109/ISPDC51135.2020.00009

[27] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00051

[28] Joseph Schuchart, Keisuke Tsugane, José Gracia, and Mitsuhisa Sato. 2018. The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 3–17.

[29] Nathan R. Tallent and John M. Mellor-Crummey. 2009. Effective Performance Measurement and Analysis of Multithreaded Applications. *SIGPLAN Not.* 44, 4 (feb 2009), 229–240. https://doi.org/10.1145/1594835.1504210