A Performance Model to Execute Workflows on High-Bandwidth-Memory Architectures

Anne Benoit[*]
École Normale Supérieure de Lyon
Lyon, France
Anne.Benoit@ens-lyon.fr

Swann Perarnau
Argonne National Laboratory
Argonne, Illinois, USA
swann@anl.gov

Loïc Pottier
École Normale Supérieure de Lyon
Lyon, France
Loic.Pottier@ens-lyon.fr

Yves Robert[†]
École Normale Supérieure de Lyon
Lyon, France
Yves.Robert@inria.fr

ABSTRACT

This work presents a realistic performance model to execute scientific workflows on high-bandwidth-memory architectures such as the Intel Knights Landing. We provide a detailed analysis of the execution time on such platforms, taking into account transfers from both fast and slow memory and their overlap with computations. We discuss several scheduling and mapping strategies: not only tasks must be assigned to computing resources, but also one has to decide which fraction of input and output data will reside in fast memory and which will have to stay in slow memory. We use extensive simulations to assess the impact of the mapping strategies on performance. We also conduct experiments for a simple 1D Gauss-Seidel kernel, which assess the accuracy of the model and further demonstrate the importance of a tuned memory management. Our model and results lay the foundations for further studies and experiments on dual-memory systems.

ACM Reference Format:
Anne Benoit, Swann Perarnau, Loïc Pottier, and Yves Robert. 2018. A Performance Model to Execute Workflows on High-Bandwidth-Memory Architectures. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3225058.3225110

1 INTRODUCTION

Recently, many TOP500 supercomputers [10] use many-core architectures to increase their processing capabilities, such as the Intel Knights Landing (KNL) [13] or some custom many-core architectures [7, 9]. Among these many-core architectures, some systems add a new level in the memory hierarchy: a byte-addressable, high-bandwidth, on-package memory. One of the first widely available systems to exhibit this kind of new memory is the KNL [2, 13, 24]. Its on-package memory (called multi-channel dynamic random access memory, or MCDRAM) of 16 GB has a bandwidth five times larger than the classic double data rate (DDR) memory. At boot, a user can decide to use this on-package memory in three modes:

Cache mode: In cache mode, MCDRAM is used by the hardware as a large last-level direct-mapped cache. In this configuration, cache misses are expensive; indeed, all data will follow the path DDR $\rightarrow$ MCDRAM $\rightarrow$ L2 caches.

Flat mode: In flat mode, the MCDRAM is manually managed by programmers. It is a new fast addressable space exposed as a NUMA node to the operating system.

Hybrid mode: This mode mixes both previous modes. A configurable ratio of the memory is used in cache mode; the other part is configured in flat mode.

While Intel promotes the cache mode, the flat mode may be more interesting in some cases. The goal of this work is to demonstrate, theoretically and experimentally, that the flat mode can obtain better performance with particular workloads (for instance, bandwidth-bound applications). Unlike GPU and classic out-of-core models, with high-bandwidth-memory systems there is no need to transfer the whole data needed for computations into the on-package memory before execution and then to transfer back the data to the DDR after the computation. An application can start its computations using data residing in both memories at the same time.

We built a detailed performance model accounting for the new dual-memory system and the associated constraints. We focus our study on scientific workflows and provide a detailed analysis of the execution time on such platforms, taking into account transfers from both fast and slow memory and their overlap with computations. The problem can be stated as follows: given (i) an application represented as a directed acyclic graph (DAG), and (ii) a many-core platform with P identical processors sharing two memories, a large slow memory and a small fast memory, how should this DAG be scheduled (which processor should execute which task and in which order) and which memory mapping should be used (which data should reside in which memory) in order to minimize the total execution time, or *makespan*.

Our major contributions are the following:

- We build a detailed performance model to analyze the execution of workflows on high-bandwidth systems, and we design several scheduling and mapping strategies.

[*]Also with Georgia Institute of Technology, Atlanta, GA.
[†]Also with University of Tennessee, Knoxville, TN.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225110

- We conduct extensive simulations to assess the impact of these strategies on performance.
- We conduct experiments for a simple 1D Gauss-Seidel kernel, which establish the accuracy of the model and further demonstrate the importance of a tuned memory management.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 formally defines the performance model with all its parameters, as well as the target architecture. Section 4 discusses the complexity of a particular problem instance, namely, linear workflows. Mapping and scheduling heuristics are introduced in Section 5 and evaluated through simulations in Section 6. The experiments with the 1D Gauss-Seidel kernel are reported in Section 7. Section 8 summarizes our conclusions and provides ideas for future work.

2 RELATED WORK

Deep memory architectures have become widely available only in the last couple of years, and studies focusing on them are rare. Furthermore, since vendors recommend to make use of them as another level of hardware-managed cache, few works make the case for explicit management of these memories. Among existing works, two major trends can be identified: studies arguing for *data placement* or for *data migration.*

Data placement [23] addresses the issue of distributing data among all available memories only once, usually at allocation time. Several efforts in this direction aim at simplifying the APIs available for placement, similarly to work on general NUMA architectures: memkind [8], the Simplified Interface for Complex Memory [14] and Hexe [18]. These libraries provide applications with intent-based allocation policies, letting users specify *bandwidth-bound* data or *latency-sensitive* data, for example. Other works [20, 25] focus instead on tracing the application behavior to optimize data placement on later runs.

Data migration addresses the issue of moving data dynamically across memories during the execution of the application. Preliminary work [19] on this approach showcased that performance of a simple stencil benchmark can be improved by migration, using a scheme similar to out-of-core algorithms, when the compute-density of the application kernel is high enough to provide compute/migration overlapping. Closer to the focus of this paper, another study [6] discussed a runtime method to schedule tasks with data dependencies on a deep memory platform. Unfortunately, the scheduling algorithm is limited to scheduling a task only after all its input data has been moved to faster memory. Also, no theoretical analysis of this scheduling heuristic was performed.

We also mention the more general field of heterogeneous computing, usually focusing on CPU-GPU architectures. Until recently, these architectures were limited to separated memories: to schedule a task on a GPU, one had to move all of its data to GPU memory. Task scheduling for such architectures is a more popular research area [1, 3, 4, 12]. Unfortunately, the scheduling heuristics for this framework are poorly applicable to our case because we can schedule tasks without moving data first. More recent GPU architectures support accessing main memory (DDR) from GPU code, for example by using unified memory since CUDA 6 [15, 17]. To the best of our knowledge, however no comprehensive study has addressed memory movement and task scheduling for these new GPUs from a performance-model standpoint.

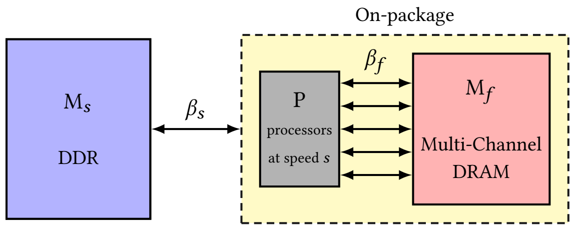

Figure 1: Target memory hierarchy.

3 MODEL

This section describes the performance model: architecture in Section 3.1, the target application in Section 3.2, scheduling constraints in Section 3.3, execution time in Section 3.4, and optimization objective in Section 3.5.

3.1 Architecture

We consider a deep-memory many-core architecture with two main memories: a large slow-bandwidth memory, M_s, and a small high-bandwidth memory, M_f. This two-unit memory system models that of the Xeon Phi (KNL) architecture [13, 24].

Let S_s denote the size and β_s the bandwidth of the memory M_s. We express memory size in terms of the number of data blocks that can be stored. A data block is any unit convenient for describing the application (e.g. bytes or words). Accordingly, bandwidths are expressed in data blocks per second. Similarly, let S_f denote the size and β_f the bandwidth of the memory M_f.

Both memories have access to the same P identical processing units, called *processors* in the following. Each processor computes at speed s. Figure 1 illustrates this architecture, where the fast MC-DRAM corresponds to M_f and the slow DDR memory corresponds to M_s.

3.2 Application

The target application is a scientific workflow represented by a directed acyclic graph $G = (V, E)$. Nodes in the graph are computation tasks, and edges are dependencies among these computation tasks. Let $V = \{v_1, \dots, v_n\}$ be the set of tasks. Let $E \subseteq V^2$ be the set of edges. If $(v_i, v_j) \in E$, task v_i must complete its execution before v_j can start. Each task $v_i \in V$ is weighted with the number of computing operations needed, w_i. Each edge $(v_i, v_j) \in E$ is weighted with the number of data blocks shared between tasks, v_i and v_j. Let $e_{i,j}$ be the number of shared (i.e., read or write) data blocks between v_i and v_j. We consider disjoint blocks; hence each $e_{i,j}$ is specific to the task pair (v_i, v_j). For each task, input edges represent data blocks that are read and output edges data blocks that are written. Hence, in the example of Figure 2, task v_2 reads $e_{1,2}$ blocks and writes $e_{2,3}$ blocks.

We define $\mathrm{succ}(v_i) = \{v_k \mid (v_i, v_k) \in E\}$ (resp. $\mathrm{pred}(v_i) = \{v_k \mid (v_k, v_i) \in E\}$) to be the successors (resp. predecessors) of task $v_i \in V$. Note that if G has multiple entry nodes (i.e., nodes without any predecessor), then we add a *dummy* node v_0 to G. We set $w_0 = 0$, and v_0 has no predecessor. Finally, v_0 is connected with edges representing the initial input to each entry node of G.

3.3 Scheduling constraints

Data blocks. At schedule time, we have to choose from which memory data blocks will be read and written. We define a variable for each edge, $e^f_{i,j}$, which represents the number of data blocks into the fast memory M_f. Symmetrically, let $e^s_{i,j}$ be for each edge the number of data blocks into the slow memory, M_s, defined as $e^s_{i,j} = e_{i,j} - e^f_{i,j}$.

We define $in^f_i = \sum_{v_j \in \mathrm{pred}(v_i)} e^f_{j,i}$ as the total number of blocks read from M_f by task v_i. Similarly, we define $out^f_i = \sum_{v_j \in \mathrm{succ}(v_i)} e^f_{i,j}$ as the total number of blocks written to M_f by task v_i. For the slow memory, M_s, we similarly define in^s_i and out^s_i.

Events. To compute the execution time and to express scheduling constraints, we define two events, $\{\sigma_1(i), \sigma_2(i)\}$, for each task v_i. These events are time steps that define the starting time and the ending time for each task. With n tasks, there are at most $2n$ such time steps (this is an upper bound since some events may coincide). A *chunk* is a period of time between two consecutive events. We denote by chunk k the period of time between events t_k and t_{k+1}, with $1 \le k \le 2n-1$. Let $t_{\sigma_1(i)}$ be the beginning and $t_{\sigma_2(i)}$ be the end of task v_i (see Figure 3). Let $\mathrm{S}^{(k)}_f$ be the number of blocks allocated to the fast memory, M_f, during chunk k. At the beginning, no blocks are allocated; hence we set $\mathrm{S}^{(0)}_f = 0$. At the start of a new chunk k, we first initialize $\mathrm{S}^{(k)}_f = \mathrm{S}^{(k-1)}_f$ and then update this value depending on the events of starting or ending a task. For task v_i, we consider two events (see Figure 3):

- At time step $t_{\sigma_1(i)}$: Before v_i begins its execution, the schedule decides which output blocks will be written in fast memory, hence what is the value of $e^f_{i,j}$, for each successor $v_j \in \mathrm{succ}(v_i)$. It must ensure that $\mathrm{S}^{(\sigma_1(i))}_f + out^f_i \le \mathrm{S}_f$. Thus at time step $t_{\sigma_1(i)}$, out^f_i blocks are reserved in M_f, hence $\mathrm{S}^{(\sigma_1(i))}_f \leftarrow \mathrm{S}^{(\sigma_1(i))}_f + out^f_i$.
- At time step $t_{\sigma_2(i)}$: After computation, we want to evict useless blocks. Since we have disjoint blocks, all read blocks in fast memory are useless after computation; hence $\mathrm{S}^{(\sigma_2(i))}_f \leftarrow \mathrm{S}^{(\sigma_2(i))}_f - in^f_i$. We do not need to transfer these blocks to M_s thanks to the disjoint blocks assumption.

To ensure that a task v_i starts only if all its predecessors have finished, we enforce the following constraint:

$$\forall (v_i, v_j) \in E,\ t_{\sigma_2(i)} \le t_{\sigma_1(j)}. \qquad (1)$$

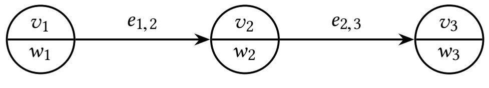

Figure 2: Simple DAG example.

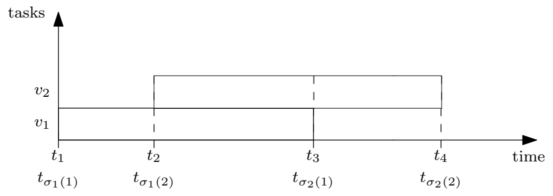

Figure 3: Events with two tasks.

Also, we have to ensure that, at any time, the number of blocks allocated in the fast memory, M_f, does not exceed S_f:

$$\forall 1 \le k \le 2n-1,\ \mathrm{S}^{(k)}_f \le \mathrm{S}_f. \qquad (2)$$

However, we must ensure that no more than P tasks are executing in parallel (no more than one task per processor at any time). Accordingly, we bound the number of executing tasks at each time step t:

$$\left| \left\{ v_i \mid t_{\sigma_1(i)} \le t < t_{\sigma_2(i)} \right\} \right| \le \mathrm{P}. \qquad (3)$$

We have at most $2n$ events in total, and we have to define a processing order on these events in order to allocate and free memory. We sort the events by nondecreasing date. If two different types of events, $\sigma_1(i)$ and $\sigma_2(j)$, happen simultaneously ($t_{\sigma_1(i)} = t_{\sigma_2(j)}$), then we process $\sigma_2(j)$ first.

3.4 Execution time

We aim at deriving a realistic model where communications overlap with computations, which is the case in most state-of-the-art multithreaded environments. We envision a scenario where communications from both memories are uniformly distributed across the whole execution time of each task, meaning that an amount of communication volume from either memory proportional to the execution progress will take place during each chunk, that is, in between two consecutive events, as explained below.

We aim at providing a formula for $w^{(k)}_i$, the number of operations executed by task v_i during chunk k, that is, between time steps t_k and t_{k+1}. If the task v_i does not compute at chunk k, then $w^{(k)}_i = 0$. Otherwise, we have to express three quantities: (i) computations; (ii) communications from and to fast memory, M_f; and (iii) communications from and to slow memory, M_s. We assume that the available bandwidths β_f and β_s are equally partitioned among all tasks currently being executed by the system. Let $\beta^{(k)}_f$ (resp. $\beta^{(k)}_s$) be the available bandwidth during chunk k for memory M_f (resp. M_s) for each task executing during that chunk. Let $N^{(k)}_f$ (resp. $N^{(k)}_s$) be the set of tasks that perform operations using the fast (resp. slow) memory bandwidth. Hence, we have $\beta^{(k)}_f = \frac{\beta_f}{|N^{(k)}_f|}$ and $\beta^{(k)}_s = \frac{\beta_s}{|N^{(k)}_s|}$.

Computations are expressed as the number of operations divided by the speed of the resource used, hence $\frac{w^{(k)}_i}{s}$ for v_i. The task v_i needs to read or write $in^f_i + out^f_i$ blocks in total at speed $\beta^{(k)}_f$. We want to express the communication time between t_k and t_{k+1} also in terms of $w^{(k)}_i$. The number of data accesses in fast memory per computing operations for task v_i can be expressed as $\frac{in^f_i + out^f_i}{w_i}$.

The communication time is obtained by multiplying this ratio by the number of operations done during this chunk, $w_i^{(k)}$, and by dividing it by the available bandwidth.

Since each task can perform communications and compute in parallel, we are limited by one bottleneck out of three; computations, or communications from M_f or communications from M_s. Hence, for each chunk k with $1 \leq k \leq 2n-1$, we have

$$\frac{w_i^{(k)}}{s} \leq t_{k+1} - t_k, \tag{4}$$

$$\frac{w_i^{(k)}(in_i^f + out_i^f)}{w_i \beta_f^{(k)}} \leq t_{k+1} - t_k, \tag{5}$$

$$\frac{w_i^{(k)}(in_i^s + out_i^s)}{w_i \beta_s^{(k)}} \leq t_{k+1} - t_k. \tag{6}$$

Note that a more conservative (and less realistic model) would assume no overlap and replace Equations (4) to (6) by

$$\frac{w_i^{(k)}}{s} + \frac{w_i^{(k)}(in_i^f + out_i^f)}{w_i \beta_f^{(k)}} + \frac{w_i^{(k)}(in_i^s + out_i^s)}{w_i \beta_s^{(k)}} \leq t_{k+1} - t_k. \tag{7}$$

An important assumption is made here: we assume that the number of flops computed with one data block remains constant. In other words, the computation time $\frac{w_i^{(k)}}{s}$ does not depend on the data scheduling (into either fast or slow memory).

From the previous equation, we can derive the expression for $w_i^{(k)}$:

$$w_i^{(k)} = (t_{k+1} - t_k) \min\left(s, \frac{\beta_f^{(k)} w_i}{in_i^f + out_i^f}, \frac{\beta_s^{(k)} w_i}{in_i^s + out_i^s}\right). \tag{8}$$

Finally, we need to compute the time step t_{k+1} for the beginning of the next chunk. We express the time $E_i^{(k)}$ for a task i to finish its execution if there are no events after t_k. We call this time the *estimated execution time*, since we do not know whether there will be an event that could modify available bandwidths and change progress rate for the execution of the task:

$$E_i^{(k)} = t_k + \frac{w_i - \sum_{k'=\sigma_1(i)}^{k-1} w_i^{(k')}}{\min\left(s, \frac{\beta_f^{(k)} w_i}{in_i^f + out_i^f}, \frac{\beta_s^{(k)} w_i}{in_i^s + out_i^s}\right)}. \tag{9}$$

Hence, the time step of the next event t_{k+1} is

$$t_{k+1} = \min_{v_i \in V} E_i^{(k)}, \tag{10}$$

Note that the task that achieves the minimum is not impacted by any event and completes its execution at time step t_{k+1}. We point out that despite the simplifications we made, we still have a complicated model to compute execution time. The reason is that the partitioning of input and output data of each task into fast and slow memory has an impact on the execution of many other tasks, since it imposes constraints on available bandwidth for both memories and remaining space in the fast memory.

There remains to ensure that all tasks perform all their operations and communications. We have the following constraint:

$$\sum_{k=1}^{2n-1} w_i^{(k)} = w_i. \tag{11}$$

Indeed, Equation (8) guarantees that the communications corresponding to an amount of work $w_i^{(k)}$ can effectively be done during chunk k, since we assume that communications from both memories are uniformly distributed during execution time. Therefore, Equation (11) is enough to validate the correctness of computations. Let $in_i^{f(k)} = \frac{w_i^{(k)}}{w_i} in_i^f$ be the number of read operations performed at chunk k by v_i from M_f. We have the following constraint on communications:

$$\sum_{k=1}^{2n-1} in_i^{f(k)} = in_i^f. \tag{12}$$

Thanks to Equation (11), we ensure that the previous constraint is respected. We have the same type of constraints on in_i^s, out_i^f, and out_i^s. To compute the total execution time of a schedule, we have

$$\mathcal{T} = \max_{v_i \in V} t_{\sigma_2(i)}. \tag{13}$$

3.5 Objective

Given a directed acyclic graph $G = (V, E)$, our goal is to find a task memory mapping between the small high-bandwidth memory and the large slow-bandwidth memory, in order to minimize the time to execute the critical path of G. More formally, we have the following:

Definition 1 (MemDag). *Given an acyclic graph $G = (V, E)$ and a platform with P identical processors sharing a two-level memory hierarchy, a large slow-bandwidth memory M_s and a small high-bandwidth memory M_f, find a memory mapping $\mathcal{X} = \{e_{i,j}^f\}_{(v_i,v_j)\in E}$ and a schedule $\{t_{\sigma_1(i)}, t_{\sigma_2(i)}\}_{v_i \in V}$ satisfying all the above constraints and minimizing*

$$\max_{v_i \in V} t_{\sigma_2(i)}.$$

4 COMPLEXITY FOR LINEAR CHAINS

MemDag is NP-complete in the strong sense. To show this, we remove the memory size constraints and assume an unlimited fast memory with infinite bandwidth. We now have the classical scheduling problem with $n = 3P$ independent tasks to be mapped on P processors, which is equivalent to the 3-partition problem [11]. Since the problem is NP-hard for independent tasks, deriving complexity results for special classes of dependence graphs seems out of reach.

Still, we have partial results for workflows whose graph is a linear chain, as detailed hereafter. Consider a linear chain of tasks

$$v_1 \xrightarrow{e_{1,2}} v_2 \rightarrow \cdots \rightarrow v_i \xrightarrow{e_{i,i+1}} v_{i+1} \rightarrow \cdots \rightarrow v_n,$$

and let $e_{0,1}$ denote the input size and $e_{n,n+1}$ the output size. Because of the dependences, each task executes in sequence. Partitioning $e_{i,i+1} = e_{i,i+1}^s + e_{i,i+1}^f$ into slow and fast memory, we aim at minimizing the makespan as follows:

$$\textsc{Minimize}\ \sum_{i=1}^{n} m_i$$

$$\textsc{Subject to}\ \begin{cases} e_{i,i+1} = e^s_{i,i+1} + e^f_{i,i+1} & \text{for } 0 \le i \le n \\ \frac{w_i}{s} \le m_i & \text{for } 1 \le i \le n \\ \frac{e^s_{i-1,i}+e^s_{i,i+1}}{\beta_s} \le m_i & \text{for } 1 \le i \le n \\ \frac{e^f_{i-1,i}+e^f_{i,i+1}}{\beta_f} \le m_i & \text{for } 1 \le i \le n \\ e^f_{i-1,i} + e^f_{i,i+1} \le \mathrm{S}_f & \text{for } 1 \le i \le n \end{cases} \tag{14}$$

Equation (14) captures all the constraints for the problem. There are $3n + 2$ unknowns, the n values m_i and the $2n + 2$ values $e^s_{i,i+1}$ and $e^f_{i,i+1}$. Of course, we can replace one of the latter values, say $e^s_{i,i+1}$, by $e_{i,i+1} - e^f_{i,i+1}$, so there are only $2n + 1$ unknowns, but the linear program reads better in the above form.

To solve Equation (14), we look for integer values, so we have an integer linear program (ILP). We attempted to design several greedy algorithms to solve Equation (14) but failed to come up with a polynomial-time algorithm for an exact solution. We also point out that it is not difficult to derive a pseudo-polynomial dynamic programming algorithm to solve Equation (14), using the size S_f of the fast memory as a parameter of the algorithm. Furthermore, on the practical side, we can solve Equation (14) as a linear program with rational unknowns and round up the solution to derive a feasible schedule.

Still, the complexity of the problem for linear workflows remains open. At the least, this negative outcome for a simple problem instance, fully evidences the complexity of MemDag.

5 HEURISTICS

Since MemDag is NP-complete, we derive polynomial-time heuristics to tackle this challenging problem. We have two types of heuristics: (i) processor allocation heuristics that compute a schedule $\mathcal{S}$, defined as a mapping and ordering on the tasks onto the processors and (ii) memory mapping heuristics that compute a memory mapping $\mathcal{X} = \{e^f_{i,j} \mid (v_i, v_j) \in E\}$. Recall that when a task finishes its execution, the memory used is released. Therefore, memory mapping is strongly affected by the scheduling decisions. We aim to design heuristics that consider both aspects and minimize the global makespan $\mathcal{T}$.

In Section 5.1, we introduce the general algorithm that computes the makespan according to scheduling and memory-mapping policies. Then we present scheduling policies in Section 5.2 and memory-mapping policies in Section 5.3.

5.1 Makespan heuristics

We outline the algorithm to compute the makespan of a task graph according to (i) a processor-scheduling policy called φ and (ii) a memory mapping policy called τ. Let $L^{(k)}$ be the list of ready tasks at time step k. A task is called *ready* when all its predecessors have completed their execution. The scheduling policy, φ, sorts the list of tasks $L^{(k)}$ according to its priority criterion, so that the task in first position in $L^{(k)}$ will be scheduled first. The memory-mapping policy, τ, returns the number of blocks in fast memory for each successor of a task, according to the size of the fast memory available for this chunk, namely, $\mathrm{S}_f - \mathrm{S}^{(k)}_f$. In other words, $\tau(v_i)$ returns all $e^f_{i,j}$ with $v_j \in \mathrm{succ}(v_i)$. Algorithm 1 computes the makespan of a task graph G, given a number of processors P, a fast memory of size S_f, and two policies: φ for processors and τ for the memory. The scheduling algorithm is based on a modified version of the *list scheduling* algorithm [16]. The idea of *list scheduling* is to build, at each time step k, an ordered list $L^{(k)}$ of tasks that are ready to be executed. Then, the algorithm greedily chooses the first task in the list if one resource is available at this time step, and so on. The key of list scheduling algorithms lies in the sorting function used to keep the ordered list $L^{(k)}$. We detail several variants in Section 5.2. Since we have homogeneous computing resources, we do not need to define a function that sorts computing resources, in order to use the most appropriate one. We simply choose any computing resource available at time step k.

We now detail the core of the algorithm. At Line 7, we iterate until the list of tasks to execute is empty, in other words until the workflow G has been completely executed. At Line 12, we sort the list of ready tasks at time-step k according to the scheduling policy. At Line 9, we release processors for each task ending at chunk k. At Line 13, we try to schedule all available tasks at time step k, and at Line 16 we choose the memory allocation for each task scheduled. At Line 20, we compute the set of tasks finishing at $k + 1$; recall that $E^{(k)}_i$ computes the estimated finishing time of task v_i at chunk k (see Equation 10). At Line 23, we compute the list of tasks ready to execute at time step $k + 1$.

5.2 Scheduling policies φ

The function $\varphi(L^{(k)})$ aims at sorting the list $L^{(k)}$ that contains the ready tasks at step k, in order to decide which tasks should be scheduled first. We define several scheduling policies to schedule tasks onto processors.

Critical path. The first heuristic, called critical path (CP), is derived from the well-known algorithm heterogeneous earliest finish time (HEFT) [22]. The HEFT algorithm chooses the task with the highest critical path at each step and schedules this task to a processor that minimizes its earliest finish time. In our model, we consider homogeneous processors; hence we select the first available processor. We define the critical path CP_i of task v_i as the maximum time to execute, without fast memory, any chain of tasks between v_i and an exit task. Formally,

$$CP_i = \max\left(\frac{w_i}{s}, \frac{in_i + out_i}{\beta_s}\right) + \max_{j \in \mathrm{succ}(v_i)} CP_j. \tag{15}$$

CP sorts the list of ready tasks according to their critical paths (in nonincreasing order of CP_i).

Gain graph. With this heuristic, we avoid short-term decisions that could lead to bad scheduling choices, we take into consideration the potential gain of using fast memory. To estimate the potential gain of a node v_i, we estimate the potential gain of the subgraph rooted at v_i, called G_i.

Algorithm 1: Compute the makespan of G

```
procedure MAKESPAN ($G$, $\varphi$, $\tau$, $S_f$, $P$) begin
    $k \leftarrow 1$ ;
    $S_f^{(0)} \leftarrow 0$ ;
    $L^{(k)} \leftarrow \{v_i$ s.t pred$(v_i) = 0\}$ ;                // Roots of $G$
    $p \leftarrow P$ ;                                                 // Available processors
    foreach $v_i \in V$ do $\sigma_1(i) \leftarrow +\infty$ ; $\sigma_2(i) \leftarrow +\infty$ ;
    while $L^{(k)} \neq \emptyset$ do
        $S_f^{(k)} \leftarrow S_f^{(k-1)}$ ;
        foreach $v_i \in V$ s.t. $\sigma_2(i) = k$ do
            $S_f^{(k)} \leftarrow S_f^{(k)} - in_i^f$ ;                // Release input blocks
            $p \leftarrow p + 1$ ;
        $L^{(k)} = \varphi(L^{(k)})$ ;        // Sort tasks according scheduling policy
        while $p > 0$ and $L^{(k)} \neq \emptyset$ do
            $v_i \leftarrow$ head$(L^{(k)})$ ;
            $L^{(k)} \leftarrow$ tail$(L^{(k)})$ ;
            $\{e_{i,j}^f \mid j \in$ succ$(v_i)\} \leftarrow \tau(v_i)$ ; // Allocate each $e_{i,j}^f$
            $S_f^{(k)} \leftarrow S_f^{(k)} + out_i^f$ ;               // Allocate output blocks
            $p \leftarrow p - 1$ ;
            $\sigma_1(i) \leftarrow k$ ;
        $i \leftarrow \operatorname{argmin}_{\sigma_1(j) \leq k < \sigma_2(j)} E_j^{(k)}$ ;   // Finishing task
        $\sigma_2(i) \leftarrow k + 1$;
        $t_{\sigma_2(i)} \leftarrow E_i^{(k)}$;
        $L^{(k+1)} \leftarrow \{v_i \mid \forall v_j \in$ pred$(v_i)$ s.t. $\sigma_2(j) \leq k+1 < \sigma_1(i)\}$ ;   // Ready tasks for next time-step
        $k \leftarrow k + 1$ ;
    return $\max_{v_i \in V} t_{\sigma_2(i)}$ ;
```

Definition 2 (Rooted subgraph). *Let $G_x = (V_x, E_x)$ be the subgraph rooted at v_x, with $v_x \in V$. The set of vertices $V_x \subseteq V$ contains all nodes in V reachable from v_x. An edge is in $E_x \subseteq E$ if and only if both of its endpoints are in V_x. Formally,*

$$(v_i, v_j) \in E_x \Leftrightarrow v_i \in V_x \text{ and } v_j \in V_x.$$

The gain of using fast memory for a graph is defined as

$$gain(G_i) = \frac{Bl_f(G_i)}{Bl_s(G_i)}, \tag{16}$$

where $Bl_f(G_i)$ is the makespan of G_i with an infinite number of processors and with an infinite fast memory and $Bl_s(G_i)$ is the makespan using only slow memory. If $gain(G_i) = 1$, then G_i is compute bound, and using fast memory might not improve efficiently its execution time. The gain graph (GG) heuristic sorts the list of tasks in nondecreasing order of potential gains using fast memory $gain(G_i)$.

5.3 Memory mapping policies τ

In addition to scheduling policies with function φ, we need to compute a memory mapping for tasks ready to be scheduled. Recall that the function $\tau(v_i)$ aims at computing the amount of data in fast memory, $e_{i,j}^f$, for each successor of v_i. We propose three heuristics returning a memory mapping.

MemCP and MemGG. The idea behind these two heuristics is to greedily give the maximum amount of memory to each successor of the task v_i that is going to be scheduled. The difference lies in the criterion used to order the successors. The MemCP heuristic uses the critical path to choose which successors to handle first (see Algorithm 2), while MemGG sorts the list of successors in increasing order of their potential gains using fast memory.

Algorithm 2: Heuristic MemCP

```
procedure MEMCP ($v_i$) begin
    Let $U$ be the set of $v_i$'s successors ordered by $CP_i$ ;
    $\mathcal{X} \leftarrow \emptyset$ ;
    foreach $j \in U$ do
        $e_{i,j}^f \leftarrow \min\left(S_f - S_f^{(k)}, e_{i,j}\right)$;
        $\mathcal{X} \leftarrow \mathcal{X} \cup \{e_{i,j}^f\}$ ;
        $S_f^{(k)} \leftarrow S_f^{(k)} + e_{i,j}^f$ ;
    return $\mathcal{X}$ ;
```

MemFair. The previous greedy heuristics MemCP and MemGG give as much as possible to the first tasks according to their criterion. The idea of MemFair is to greedily give data blocks in fast memory to the tasks, according to their amount of computations, but accounting for other successors. Recall that $S_f - S_f^{(k)}$ is the number of blocks available at chunk k. MemFair spreads blocks from fast memory across the successors of the scheduled tasks: each successor has at most a number of blocks equal to $S_f - S_f^{(k)}$ divided by the number of successors. Algorithm 3 details this heuristic.

Algorithm 3: Heuristic MemFair

```
procedure MEMFAIR ($v_i$) begin
    Let $U$ be the set of $v_i$'s successors ordered by $w_i$ ;
    $\mathcal{X} \leftarrow \emptyset$ ;
    foreach $j \in U$ do
        $e_{i,j}^f \leftarrow \min\left(\left\lfloor \frac{S_f - S_f^{(k)}}{|\operatorname{succ}(v_i)|} \right\rfloor, e_{i,j}\right)$;
        $\mathcal{X} \leftarrow \mathcal{X} \cup \{e_{i,j}^f\}$ ;
        $S_f^{(k)} \leftarrow S_f^{(k)} + e_{i,j}^f$ ;
    return $\mathcal{X}$ ;
```

By combining two heuristics for processor scheduling and three heuristics for memory mapping, we obtain a total of six heuristics.

5.4 Baseline heuristics

For comparison and evaluation purposes, we define three different baseline heuristics for memory mapping. Because of lack of space, we combine them only with CP as a processor-scheduling heuristic. Results when combining with GG are similar and available in the extended version [5].

CP +NoFast and CP +InfFast. NoFast considers that no fast memory is available, while InfFast uses a fast memory of infinite size (but still with a finite bandwidth, β_f).

CP +CcMode. This baseline heuristic is more complicated. Recall that our target architecture is the Xeon Phi KNL, which proposes two principal modes to manage the fast memory: the cache mode and the flat mode [24]. In the cache mode, the fast memory is managed by the system as a large cache. Our memory-mapping heuristic CcMode aims at imitating the KNL cache mode behavior. In CcMode, we divide the fast memory into P slices, where P is the total number of processors and each processor has access only to its own slice into the fast memory. When a node v_i is scheduled onto a processor, all its output blocks are allocated, if possible, to fast memory. If the slice into fast memory is too small to contain the output blocks of each successor, we consider the successors in nondecreasing index order (v_{j-1} is handled before v_j). CcMode aims at providing a more realistic comparison baseline than does NoFast.

6 SIMULATIONS

To assess the efficiency of the heuristics defined in Section 5, we have conducted extensive simulations. Simulation settings are discussed in Section 6.1, and results are presented in Section 6.2. The simulator is publicly available at https://perso.ens-lyon.fr/loic.pottier/archives/simu-deepmemory.zip so that interested readers can instantiate their preferred scenarios and repeat the same simulations for reproducibility purpose.

6.1 Simulation settings

To evaluate the efficiency of the proposed heuristics, we conduct simulations using parameters corresponding to those of the Xeon Phi KNL architecture. Unless stated otherwise, the bandwidth of the slow memory, β_s, is set to 90 GB/s, while the fast memory is considered to be five times faster, at 450 GB/s [24]. The processor speed, s, is set to 1.4 GHz (indeed the processor speed of KNL cores ranges from 1.3 to 1.5 with the Turbo mode activated [13]). The size of the fast memory is set to 16 GB unless stated otherwise, and the slow memory is considered infinitely large.

To instantiate the simulations, we use random directed acyclic graphs from the Standard Tasks Graphs (STG) set [21]. The STG set provides 180 randomly generated DAGs with different sizes ranging from 50 to 5,000 nodes. We select two sizes: 50 and 100 nodes. This leads us to two sets of 180 same-size graphs. For these two sets, we further distinguish between sparse and dense graphs. Recall that the density of a graph $G = (V, E)$ is defined as $\frac{|E|}{|V|(|V|-1)}$; hence the density is 0 for a graph without edges and 1 for a complete graph. We consider two different subsets of each set: (i) the 20 graphs, over the 180 available for each set, that exhibit the lower densities and (ii) the 20 graphs with the higher densities in the set. Because of lack of space, we report results only for sparse subsets and graphs with 50 nodes, but detailed results for the dense subsets and for larger graphs with 100 nodes are available in the companion research report [5].

We need to set the number of computing operations, w_i, for each node, v_i, in the DAG and the number of data blocks, $e_{i,j}$ (i.e., number of bytes) on each edge. One of the key metrics in task graph scheduling with multiple memory levels is the computation-to-communication ratio (CCR). In our framework, for a node v_i and an edge $e_{i,j}$, the CCR is the ratio of the time required to compute w_i operations over the time required to transfer $e_{i,j}$ blocks to slow memory:

$$\mathrm{CCR} = \frac{w_i}{s} \Big/ \frac{e_{i,j}}{\beta_s}.$$

We let the CCR vary in our experiments and we instantiate the graphs as follows. For the computing part, we choose w_i uniformly between $w_i^{\min} = 10^4$ and $w_i^{\max} = 10^6$ flops: since the processor speed s is set to 1.4 GHz, the computing part of each node is comprised between 10^{-3} and 10^{-5} seconds. For data transfers, we randomly and uniformly pick $e_{i,j}$ in the interval $\left[\frac{w_i^{\min}\times\beta_s}{s\times\mathrm{CCR}}, \frac{w_i^{\max}\times\beta_s}{s\times\mathrm{CCR}}\right]$.

6.2 Results

To evaluate the heuristics, we execute each heuristic 50 times with different random weights on the 20 graphs from each STG subset; hence each point is the average of 1,000 executions. Then, we compute the average makespan over all the runs. All makespans are normalized with the baseline without fast memory, CP +NoFast. The standard deviation is represented as error bars. We study the impact of the number of processors, the size of fast memory, and the fast memory bandwidth, by varying these parameters in the simulations.

6.2.1 Impact of the number of processors. Figure 4 presents the normalized makespan of graphs of 50 nodes, and with 1 GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64 with the scheduling policy CP combined with each memory mapping. Figure 5 presents the same results but for the scheduling policy GG. All heuristics exhibit good performance in comparison to the two baselines CP +NoFast and CP +CcMode, but only GG +MemFair and CP +MemFair clearly outperform other heuristics, with an average gain around 50% over the baseline CP +NoFast. CP and GG present similar trends; the difference between heuristics performance lies in the memory mapping. With the approaches MemCP and MemGG, we give the maximum number of blocks possible to the successors (according to the heuristic rules). Several nodes might be strongly accelerated but likely at the expense of other nodes in the graph. On the contrary, MemFair aims at giving a fair amount of fast memory to each successor of the scheduled task. As a result, the usage of fast memory is more balanced across tasks in the graph than for mappings produced by MemCP and MemGG.

When the CCR decreases, the number of data blocks on the edges increases, and the graph no longer fits into fast memory. On the contrary, when the CCR increases, the number of data blocks on the edges decreases, so that the graph fits, at some point, into the fast memory; but then computations become the bottleneck, and the benefits of the high-bandwidth memory are less important. For small values of P, MemCP and MemGG show almost the same behavior with noticeable improvements over the case without fast memory NoFast, but are close to the cache mode CcMode. When the number of processors increases, the performance of CcMode

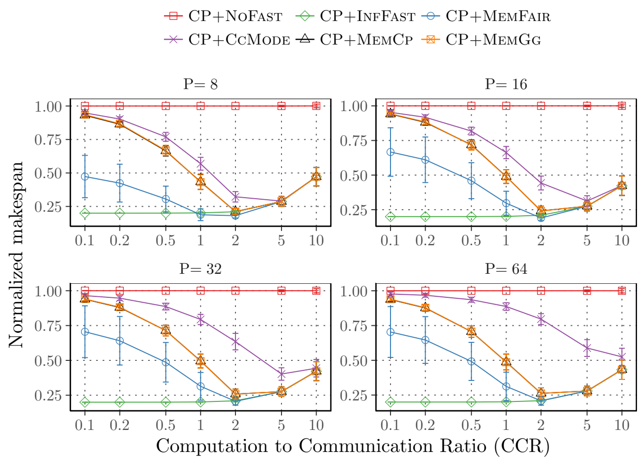

Figure 4: Impact of the number of processors with 50 nodes and $S_f = 1$ GB fast memory for CP scheduling heuristic.

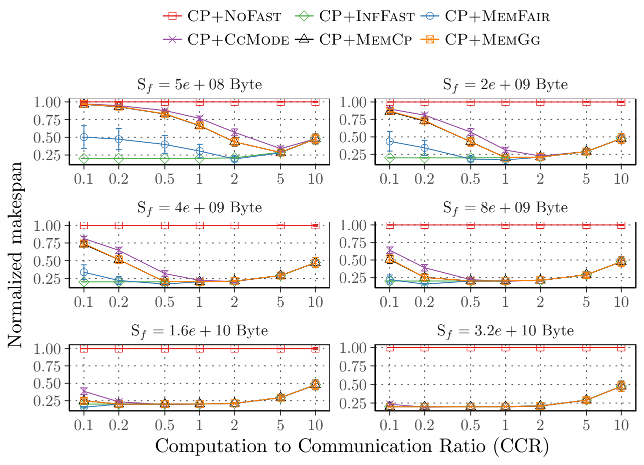

Figure 6: Impact of fast memory size with 50 nodes and 8 processors.

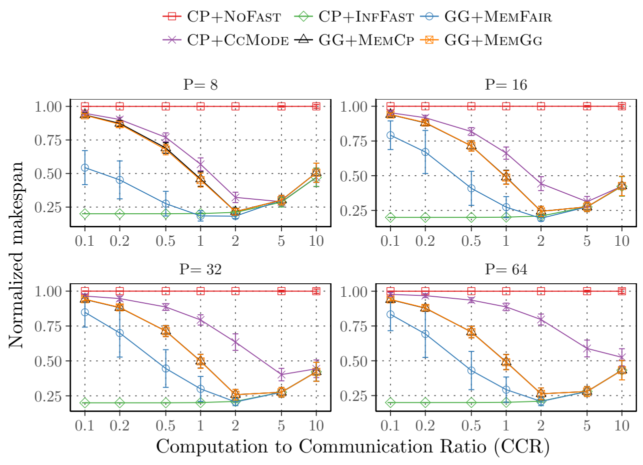

Figure 5: Impact of the number of processors with 50 nodes and $S_f = 1$ GB fast memory for GG scheduling heuristic.

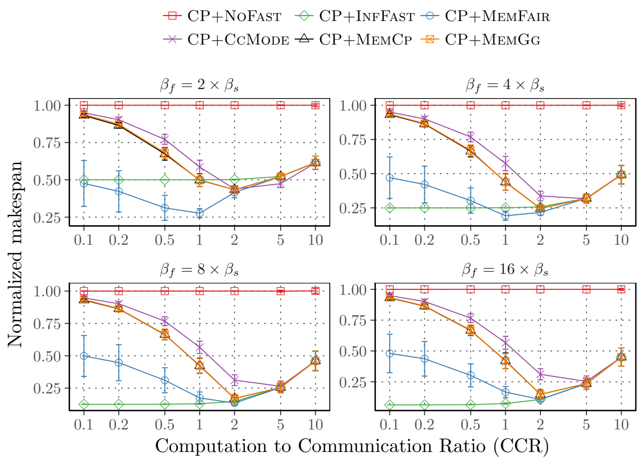

Figure 7: Impact of fast memory bandwidth with 50 nodes, 8 processors, and $S_f = 1$ GB.

decreases, mainly because when P increases, the size of each fast memory slice decreases.

6.2.2 Impact of fast memory size. Figure 6 presents the results for graphs with 50 nodes, with 8 processors when we vary the fast memory size and the CCR (see detailed results with more processors in the research report [5]). As always, we vary the CCR from 0.1 to 10 and the size of fast memory from 200 MB to 16 GB. Recall that the fast memory bandwidth is set to 450 GB/s (five times faster). Clearly, when the size of the memory increases, the global performance of heuristics converges to the baseline CP +InfFast. All proposed heuristics perform better than the cache mode CcMode, and MemFair outperforms other memory mappings with an average gain of around 25%, when the size of fast memory is small enough so that all data do not fit in fast memory. We observe that the CCR for which all heuristics reach the lower baseline InfFast decreases when the fast memory size increases.

6.2.3 Impact of fast memory bandwidth. Figure 7 presents the results for graphs with 50 nodes, with 8 processors and 1 GB fast memory. The bandwidth of the fast memory ranges from 2 times up to 16 times the slow memory bandwidth. We observe that for small bandwidths, the memory mapping MemFair outperforms the baseline InfFast. Recall that the fast memory bandwidth is the same for every memory heuristic, so InfFast has an infinite fast memory with a finite bandwidth. When the bandwidth is too small compared with the slow memory bandwidth, saturating the fast memory leads to decreased performance because the fast memory bandwidth is shared by the number of tasks concurrently trying to gain access to it.

6.2.4 Summary. All heuristics are efficient compared with the baseline without fast memory. But only two combinations, CP +MemFair and GG +MemFair, clearly outperform the baseline CP +CcMode. Recall that CcMode aims at imitating KNL's behavior when the system manages the fast memory as a cache. Therefore, obtaining better performance than this mode demonstrates the importance of a fine-tuned memory management when dealing with deep-memory architectures.

7 EXPERIMENTS

In this section, we assess the accuracy of the model by running both simulations and actual experiments for a 1D Gauss-Seidel computational kernel, using data movement between the slow and fast memories. We detail experimental settings in Section 7.1 and present results in Section 7.2. The code is available at https://gitlab.com/perarnau/knl/.

7.1 Experimental settings

Application data is partitioned into rectangular tiles and iteratively updated as shown in Algorithm 4, where Tile_i^t denotes tile i at iteration t.

Algorithm 4: 1D Gauss-Seidel algorithm

```
1 procedure 1D-GS(array) begin
2     for $t = 1$ to ... do
3         for $i = 1$ to ... do
4             $\text{Tile}_i^t \leftarrow$ Gauss-Seidel $(\text{Tile}_{i-1}^t, \text{Tile}_i^{t-1}, \text{Tile}_{i+1}^{t-1})$;
```

At each step of the procedure 1D-GS, Tile_i^t is computed as a combination of three tiles: (i) Tile_{i-1}^t, its left neighbor that has just been updated at iteration t; (ii) Tile_i^{t-1}, its current value from iteration $t-1$; and (iii) Tile_{i+1}^{t-1}, its right neighbor from iteration $t-1$. Each tile is extended with phantom borders whose size depends on the updating mask of the Gauss-Seidel kernel (usually we need one or two columns on each vertical border), so that each tile works on a single file of size m.

Our model currently does not allow for data movements between the slow and fast memories, so we decompose the update of each tile Tile_i^t into three sequential tasks: (i) task R_i^t transfers the tile from slow memory to fast memory; (ii) task C_i^t computes the tile in fast memory; and (iii) task W_i^t writes the updated tile back into slow memory. This leads to the task graph shown in Figure 8. We use this graph as input for the simulations and run the scheduling and mapping heuristics presented in Section 5.

For the experiments, we extend the previous study developed for parallel stencil applications in [19] and provide a deep-memory implementation of the 1D Gauss-Seidel kernel for the KNL architecture. First, we copy tiles to migrate input and output data between slow and fast memory. Then, migration tasks and work tasks are pipelined, so that for a given iteration, three batches of tasks are executing concurrently: prefetching of future tiles in fast memory, computing on tiles already prefetched, and flushing of computed tiles back into slow memory. This scheme corresponds to executing tasks R_{i+1}^t, C_i^t and W_{i-1}^t in parallel, as in the classical wavefront execution of the dependence graph in Figure 8.

For the experiments, the parameters of the benchmark were the following: (i) input array of 64 GB; (ii) tiles of size 32 MB: (iii) 64 cores at 1.4 GHz; and (iv) 64 threads used. We vary the CCR by increasing the number of operations done per tile.

7.2 Results

For the benchmark runs, the platform runs CentOS 7.2, and experiments were repeated 10 times for accuracy. Figure 9 gives the performance of the benchmark against a baseline running entirely in slow memory with 64 threads. Figure 10 reports the results of the simulations for the same task graph, using the best heuristic, CP +MemFair, on 64 threads.

We observe a good concordance between the experiments and the simulations. In both cases, the performance of the application is greatly increased when using the overlapping scheme and fast memory access. For small values of the CCR, the execution time is divided by half. Then the gain starts to decrease when the CCR reaches the value 2, until reaching a threshold where there is no gain left. This is expected: the threshold is reached when the cost of computations becomes higher than the transfer time of a whole tile

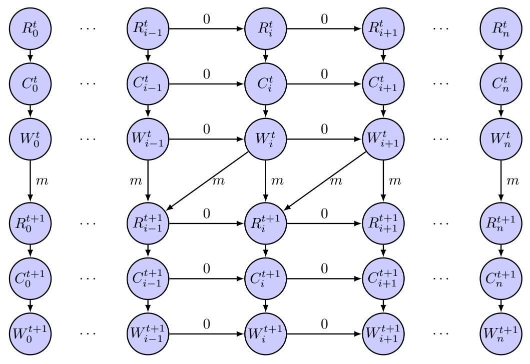

Figure 8: 1D stencil task graph, where t is the iteration index, i is the tile index, and m is the size of one tile.

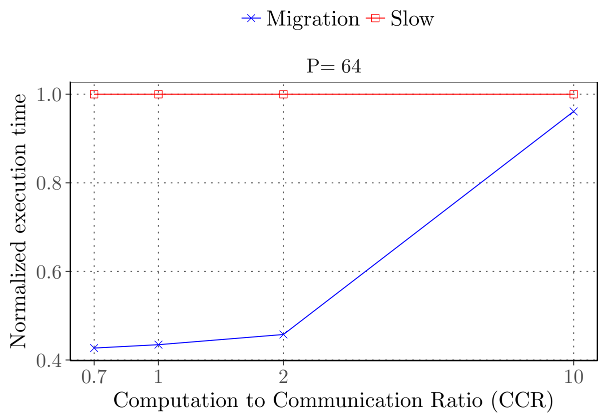

Figure 9: Performance of a 1D stencil running on a KNL with 64 threads.

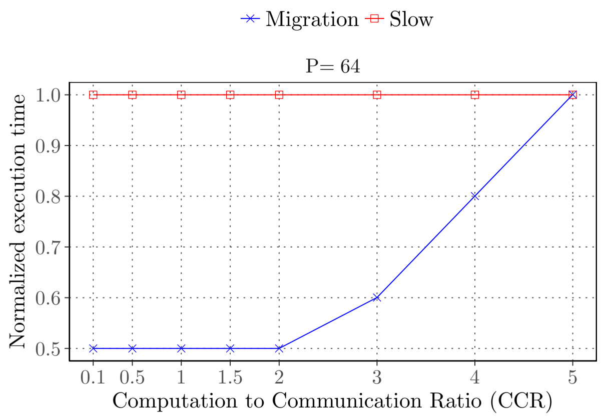

Figure 10: Performance of a 1D stencil according to the model, with 64 threads.

from slow memory. We have a discrepancy here since the threshold value is 10 for the experiments and 5 for the simulations. Still, both plots nicely demonstrate the impact of the CCR and the possibility of gaining performance when the CCR is low, hence when access to slow memory is the bottleneck.

8 CONCLUSION

In this paper, we address the problem of scheduling task graphs onto deep-memory architectures such as the Intel KNL. In addition to the traditional problems of ordering the tasks and mapping them onto processors, a key decision in the scheduling process is what proportion of fast memory should be assigned to each task. We provide a complete and realistic performance model for the execution of workflows on dual-memory systems, as well as several polynomial-time heuristics for both scheduling and memory mapping. These heuristics have been tested through extensive simulations and were shown to outperform the baseline strategies, thereby demonstrating the importance of a good memory-mapping policy. These results also demonstrate that the KNL cache mode can be outperformed by a customized memory mapping. We also conducted experiments on a KNL platform with a 1D Gauss-Seidel computational kernel and compared the performance of a tuned memory mapping with that of the heuristics in simulation, thereby demonstrating the accuracy of the model and bringing another practical proof of the importance of a fine-tuned memory management of the fast memory.

Future work will be devoted to extending simulations on other kinds of workflow graphs, such as fork-join graphs, and extending the model in order to allow for moving data across both memory types. This is a challenging endeavor, because it requires deciding which data blocks to move, and when to move them, while other tasks are executing. Also, we would like to conduct additional experiments with more complicated workflows, such as those arising from dense or sparse linear factorizations in numerical linear algebra. All this future work will rely on the model and results of this paper, which represent a first, yet crucial, step toward a full understanding of scheduling problems on deep-memory architectures.

ACKNOWLEDGMENTS

Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] Massinissa Ait Aba, Lilia Zaourar, and Alix Munier. 2017. Approximation Algorithm for Scheduling a Chain of Tasks on Heterogeneous Systems. In *European Conference on Parallel Processing*. Springer, 353–365.
[2] Ryo Asai. 2016. *Clustering Modes in Knights Landing Processors: Developer's Guide.* Technical Report. Colfax International.
[3] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. 2010. Data-Aware Task Scheduling on Multi-Accelerator Based Platforms. In *IEEE Int. Conf. on Parallel and Distributed Systems.* 291–298.
[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.
[5] Anne Benoit, Swann Perarnau, Loïc Pottier, and Yves Robert. 2018. *A performance model to execute workflows on high-bandwidth memory architectures.* Research report RR-9165. INRIA. Available at hal.inria.fr.
[6] Kavitha Chandrasekar, Xiang Ni, and Laxmikant V. Kalé. 2017. A Memory Heterogeneity-Aware Runtime System for Bandwidth-Sensitive HPC Applications. In *IEEE Int. Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA.* 1293–1300.
[7] PEZY Computing. 2017. ZettaScaler-2.0 Configurable Liquid Immersion Cooling System. (2017). http://www.exascaler.co.jp/wp-content/uploads/2017/11/zettascaler2.0_en_page.pdf
[8] Intel Corporation. 2018. Memkind: A User Extensible Heap Manager. https://memkind.github.io. (2018).
[9] Jack Dongarra. 2016. *Report on the Sunway TaihuLight system.* Research report UT-EECS-16-742. Univ. Tennessee. Available at www.netlib.org.
[10] Erich Strohmaier et al. 2017. The TOP500 benchmark. (2017). https://www.top500.org/.
[11] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Company.
[12] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* 1299–1308.
[13] Intel. 2017. *Intel Xeon Phi Processor: Performance Monitoring Reference Manual – Volume 1: Registers.* Technical Report. Intel.
[14] Los Alamos National Laboratory. 2017. Simplified Interface to Complex Memory. https://github.com/lanl/SICM. (2017).
[15] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. 2014. An Investigation of Unified Memory Access Performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC).* 1–6.
[16] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits* (1st ed.). McGraw-Hill Higher Education.
[17] NVIDIA. 2018. CUDA: Unified Memory Programming. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd. (2018).
[18] Lena Oden and Pavan Balaji. 2017. Hexe: A Toolkit for Heterogeneous Memory Management. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS).*
[19] Swann Perarnau, Judicael A. Zounmevo, Balazs Gerofi, Kamil Iskra, and Pete Beckman. 2016. Exploring Data Migration for Future Deep-Memory Many-Core Systems. In *IEEE Cluster.*
[20] Harald Servat, Antonio J. Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017.* 126–136.
[21] Takao Tobita and Hironori Kasahara. 2002. A Standard Task Graph Set for Fair Evaluation of Multiprocessor Scheduling Algorithms. *Journal of Scheduling* 5, 5 (2002), 379–394.
[22] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (Mar 2002), 260–274.
[23] Didem Unat, John Shalf, Torsten Hoefler, Thomas Schulthess, Anshu Dubey, et al. 2014. *Programming Abstractions for Data Locality.* Technical Report.

[24] Andrey Vladimirov and Ryo Asai. 2016. *MCDRAM as High-Bandwith Memory (HBM) in Knights Landing Processors: Developer's Guide.* Technical Report. Colfax International.

[25] Gwendolyn Voskuilen, Arun F. Rodrigues, and Simon D. Hammond. 2016. Analyzing Allocation Behavior for Multi-Level Memory. In *Proceedings of the Second International Symposium on Memory Systems, MEMSYS 2016, Alexandria, VA, USA, October 3-6, 2016*. 204–207.