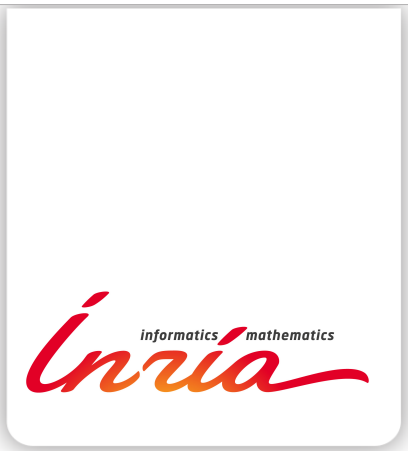
A performance model to execute workflows on high-bandwidth memory architectures

Anne Benoit, Swann Perarnau, Loïc Pottier, Yves Robert

RESEARCH REPORT
N° 9165
April 2018
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9165--FR+ENG



A performance model to execute workflows on high-bandwidth memory architectures

Anne Benoit*†, Swann Perarnau‡, Loïc Pottier*, Yves Robert*§

Project-Team ROMA

Research Report n° 9165 — April 2018 — 28 pages

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. Extensive simulations allow us to assess the impact of the mapping strategies on performance. We also conduct actual 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. Altogether, our model and results lay the foundations for further studies and experiments on dual-memory systems.

Key-words: performance model, workflow, task graph; scheduling, mapping, memory hierarchy, manycore, high-bandwidth memory.

* Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, LIP UMR5668, F-69342, Lyon, France
† Georgia Institute of Technology, Atlanta, GA, USA
‡ Argonne National Laboratory, USA
§ University Tennessee Knoxville, USA

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un modèle de perfomance pour exécuter des graphes de tâches sur des architectures à mémoire haute performance

Résumé : Ce travail présente un modèle de performance réaliste pour exécuter des workflows scientifiques sur des architectures ayant des mémoires à bande passante élevée, comme par exemple Intel Knights Landing. Nous fournissons une analyse détaillée du temps d'exécution sur ces plates-formes, en tenant compte des transferts depuis deux mémoires (rapide et lente), et leur recouvrement avec les calculs. Nous introduisons plusieurs stratégies d'ordonnancement et de placement mémoire: non seulement les tâches doivent être assignées aux ressources de calcul, mais il faut aussi décider quelle fraction des données d'entrée et de sortie va résider en mémoire rapide, alors que le reste sera en mémoire lente. Des simulations approfondies nous permettent d'évaluer l'impact des stratégies de placement sur la performance. Nous menons également des expériences réelles pour un noyau de Gauss-Seidel 1D simple, afin d'évaluer la précision du modèle. Nous démontrons ainsi l'importance d'une gestion fine de la mémoire sur les systèmes avec double mémoire.

Mots-clés : modèle de performance, graphe de tâches, ordonnancement, placement de données, hiérarchie mémoire, architecture massivement parallèle, mémoire haute performance.

1 Introduction

Recently, many TOP500 supercomputers [1] use many-core architectures to increase their processing capabilities, such as the Intel Knights Landing [2] or some custom many-core architectures [3, 4]. 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 Intel's Knights Landing [2, 5, 6]. Its on-package memory (called Multi-Channel Dynamic Random Access Memory, or MCDRAM) of 16GB has a bandwidth five times larger than the classic DDR (Double Data Rate) 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 very expensive, indeed all data will follow the path DDR → MCDRAM → 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 OS.

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 could be more interesting in some cases. The goal of this work is to demonstrate, theoretically and experimentally, that 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 build 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 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 to schedule this DAG (which processor executes which task and in which order) and to find a memory mapping (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 design several scheduling and mapping strategies.
- We conduct extensive simulations to assess the impact of these strategies on performance.
- We conduct actual 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 is devoted to formally defining 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. Finally, Section 8 provides final remarks and hints for future work.

2 Related work

Deep memory architectures only became widely available in the last couple of years, and studies focusing on them are rare. Furthermore, as 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 ones, two major trends can be identified: studies are either arguing for *data placement* or for *data migration*.

Data placement [7] 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 [9] and Hexe [10]. These libraries provide applications with intent-based allocation policies, letting users specify *bandwidth-bound* data or *latency-sensitive* data for example. Other works [11, 12] 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 [13] on this approach showcased that performance of a simple stencil benchmark could 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 [14] discussed a runtime method to schedule tasks with data dependencies on a deep memory platform. Unfortunately, the scheduling algorithm is limited to only scheduling a task after all its input data has been moved to faster memory. Also, no theoretical analysis of this scheduling

heuristic was performed.

We should 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 [15, 16, 17, 18]. 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 using unified memory since CUDA 6 [19, 20]. Unfortunately, to the best of our knowledge, there is not yet any comprehensive study that addresses memory movement and task scheduling for these new GPUs from a performance-model standpoint.

3 Model

This section details all the components of the performance model: architecture in Section 3.1, 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 Knight Landing (KNL) architecture [2, 6].

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 to describe the application, bytes or words etc. 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 Multi-Channel DRAM (MCDRAM) corresponds to M_f, and the slow Double Data Rate (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 $V =$

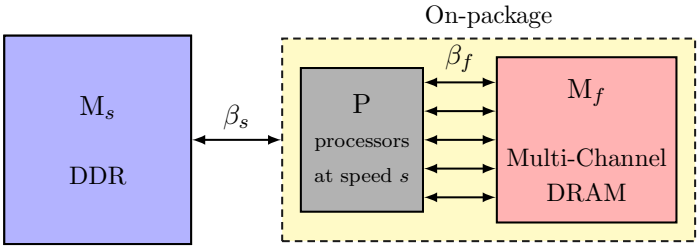


Figure 1: Target memory hierarchy.

$\{v_1, \ldots, v_n\}$ be the set of tasks. Let $E \subseteq V^2$ be the set of edges: if $(v_i, v_j) \in E$, it means that 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 $\text{succ}(v_i) = \{v_k \mid (v_i, v_k) \in E\}$ (resp. $\text{pred}(v_i) = \{v_k \mid (v_k, v_i) \in E\}$) 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 $in^f_i = \sum_{v_j \in \text{pred}(v_i)} e^f_{j,i}$, the total number of blocks read from M_f by task v_i. Similarly, we define $out^f_i = \sum_{v_j \in \text{succ}(v_i)} e^f_{i,j}$, the total number of blocks written to M_f by task v_i. For the slow memory M_s, we similarly

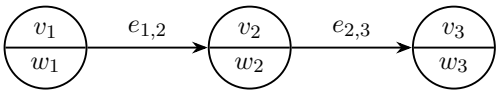


Figure 2: Simple DAG example.

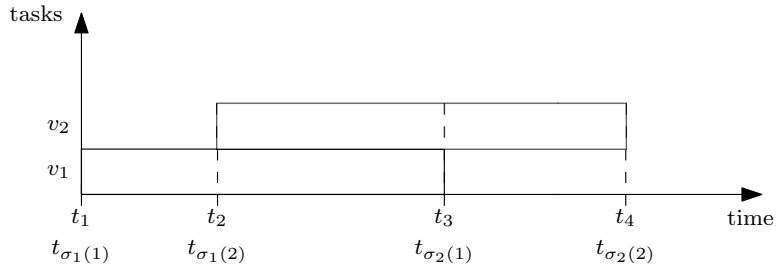


Figure 3: Events with two tasks.

define in_i^s and out_i^s.

Events In order 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 $1 \leq k \leq 2n-1$. Let $t_{\sigma_1(i)}$ be the beginning and $t_{\sigma_2(i)}$ be the end of task v_i (see Figure 3). Let $\mathrm{S}_f^{(k)}$ be the number of blocs allocated into the fast memory M_f during chunk k. At the beginning, no blocks are allocated, hence we set $\mathrm{S}_f^{(0)} = 0$. At the start of a new chunk k, we first initialize $\mathrm{S}_f^{(k)} = \mathrm{S}_f^{(k-1)}$ and then we update this value depending upon 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 $e_{i,j}^f$, for each successor $v_j \in \mathrm{succ}(v_i)$. It must ensure that $\mathrm{S}_f^{(\sigma_1(i))} + out_i^f \leq \mathrm{S}_f$. Thus at time-step $t_{\sigma_1(i)}$, out_i^f blocks are reserved in M_f, hence $\mathrm{S}_f^{(\sigma_1(i))} \leftarrow \mathrm{S}_f^{(\sigma_1(i))} + out_i^f$.
- 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 $\mathrm{S}_f^{(\sigma_2(i))} \leftarrow \mathrm{S}_f^{(\sigma_2(i))} - in_i^f$. 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 of its predecessors have finished, we enforce the following constraint:

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

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:

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

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

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

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 non-decreasing date. If two different types of events, namely $\sigma_1(i)$ 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 multi-threaded 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, i.e., in between two consecutive events, as explained below.

We aim at providing a formula for $w_i^{(k)}$, the number of operations executed by task v_i during chunk k, i.e., between time-steps t_k and t_{k+1}. If the task v_i does not compute at chunk k, then $w_i^{(k)} = 0$. 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 executing by the system. Let $\beta_f^{(k)}$ (resp. $\beta_s^{(k)}$) be the available bandwidth during chunk k for memory M_f (resp. M_s) for each task executing during that chunk. Let $N_f^{(k)}$ (resp. $N_s^{(k)}$) be the set of tasks that perform operations using the fast (resp. slow) memory bandwidth. Hence, we have $\beta_f^{(k)} = \frac{\beta_f}{|N_f^{(k)}|}$ and $\beta_s^{(k)} = \frac{\beta_s}{|N_s^{(k)}|}$.

Computations are expressed as the number of operations divided by the speed of the resource used, hence $\frac{w_i^{(k)}}{s}$ for v_i. The task v_i needs to read or write $in_i^f + out_i^f$ blocks in total at speed $\beta_f^{(k)}$. We want to express the communication time between t_k and t_{k+1} also in terms of $w_i^{(k)}$. The number of data accesses in fast memory per computing operations for task v_i can be expressed as $\frac{in_i^f + out_i^f}{w_i}$. 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, either 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 upon the data scheduling (either into 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. First, 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. This is because 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, at is 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. First, 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. This is because 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. Finally, 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 an acyclic directed 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:

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 above constraints and that minimizes*

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

4 Complexity for linear chains

Clearly, MemDag is NP-complete in the strong sense. To see this, remove memory size constraints and assume to have 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 [21]. 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

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

and let $e_{0,1}$ denote the input size, and $e_{n,n+1}$ the output size. Due to the dependences, each task executes in sequence. Partitioning $e_{i,i+1} = e^s_{i,i+1} + e^f_{i,i+1}$ into slow and fast memory, we aim at minimizing the makepsan as follows:

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

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

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 ILP (Integer Linear Program). 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 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 very least, this negative outcome for a very 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 $\mathcal{X} = \{e_{i,j}^f \,|\, (v_i, v_j) \in E\}$. Recall that, when a task finishes its execution, the memory used is released. Therefore, memory mapping is strongly impacted by the scheduling decisions. We aim at designing heuristics that consider both aspects and aim at minimizing the global makespan $\mathcal{T}$.

We introduce the general algorithm that computes the makespan according to scheduling and memory mapping policies in Section 5.1, 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 of 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 $\mathrm{S}_f - \mathrm{S}_f^{(k)}$. In other words, $\tau(v_i)$ returns all $e_{i,j}^f$ 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 [22]. 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 below 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, so 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. First, 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 we choose the memory allocation for each task scheduled at Line 16. At Line 20, we compute the set of tasks finishing at $k+1$; recall that $E_i^{(k)}$ computes the estimated finishing time of task v_i at chunk k (see Equation 10). Finally, at Line 23, we compute the list of tasks ready to execute at time-step $k+1$.

Algorithm 1: Compute the makespan of G

1 **procedure** MAKESPAN (G, φ, τ, S_f, P) **begin**
2 $k \leftarrow 1$;
3 $\mathrm{S}_f^{(0)} \leftarrow 0$;
4 $L^{(k)} \leftarrow \{v_i \text{ s.t } \mathrm{pred}(v_i) = 0\}$; // Roots of G
5 $p \leftarrow \mathrm{P}$; // Available processors
6 **foreach** $v_i \in V$ **do** $\sigma_1(i) \leftarrow +\infty$; $\sigma_2(i) \leftarrow +\infty$;
7 **while** $L^{(k)} \neq \emptyset$ **do**
8 $\mathrm{S}_f^{(k)} \leftarrow \mathrm{S}_f^{(k-1)}$;
9 **foreach** $v_i \in V$ **s.t.** $\sigma_2(i) = k$ **do**
10 $\mathrm{S}_f^{(k)} \leftarrow \mathrm{S}_f^{(k)} - in_i^f$; // Release input blocks
11 $p \leftarrow p + 1$;
12 $L^{(k)} = \varphi(L^{(k)})$; // Sort tasks according scheduling policy
13 **while** $p > 0$ **and** $L^{(k)} \neq \emptyset$ **do**
14 $v_i \leftarrow \mathrm{head}(L^{(k)})$;
15 $L^{(k)} \leftarrow \mathrm{tail}(L^{(k)})$;
16 $\{e_{i,j}^f \mid j \in \mathrm{succ}(v_i)\} \leftarrow \tau(v_i)$; // Allocate each $e_{i,j}^f$
17 $\mathrm{S}_f^{(k)} \leftarrow \mathrm{S}_f^{(k)} + out_i^f$; // Allocate output blocks
18 $p \leftarrow p - 1$;
19 $\sigma_1(i) \leftarrow k$;
20 $i \leftarrow \underset{\sigma_1(j) \leq k < \sigma_2(j)}{\mathrm{argmin}} E_j^{(k)}$; // Finishing task
21 $\sigma_2(i) \leftarrow k + 1$;
22 $t_{\sigma_2(i)} \leftarrow E_i^{(k)}$;
23 $L^{(k+1)} \leftarrow \{v_i \mid \forall v_j \in \mathrm{pred}(v_i) \text{ s.t. } \sigma_2(j) \leq k+1 < \sigma_1(i)\}$; // Ready tasks for next time-step
24 $k \leftarrow k + 1$;
25 **return** $\max_{v_i \in V} t_{\sigma_2(i)}$;

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 HEFT (Heterogeneous Earliest Finish Time) [23]. 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,

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

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

Gain Graph With this heuristic, we aim at avoiding short-term decisions that could lead to bad scheduling choices, by taking 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.

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 GG (Gain Graph) heuristic sorts the list of tasks in non-decreasing 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 $e^f_{i,j}$ for each successor of v_i. We propose below 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**
    $e_{i,j}^f \leftarrow \min\left(\mathrm{S}_f - \mathrm{S}_f^{(k)}, e_{i,j}\right)$;
    $\mathcal{X} \leftarrow \mathcal{X} \cup \{e_{i,j}^f\}$;
    $\mathrm{S}_f^{(k)} \leftarrow \mathrm{S}_f^{(k)} + e_{i,j}^f$;
  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 $\mathrm{S}_f - \mathrm{S}_f^{(k)}$ is the number of blocks available at chunk k. MemFair is spreading blocks from fast memory across the successors of the scheduled tasks: each successor has at most a number of blocks equal to $\mathrm{S}_f - \mathrm{S}_f^{(k)}$ 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**
    $e_{i,j}^f \leftarrow \min\left(\left\lfloor \frac{\mathrm{S}_f - \mathrm{S}_f^{(k)}}{\mid \mathrm{succ}(v_i) \mid} \right\rfloor, e_{i,j}\right)$;
    $\mathcal{X} \leftarrow \mathcal{X} \cup \{e_{i,j}^f\}$;
    $\mathrm{S}_f^{(k)} \leftarrow \mathrm{S}_f^{(k)} + e_{i,j}^f$;
  return $\mathcal{X}$;

Altogether, 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.

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 [6]. 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 only access 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, into fast memory. If the slice into fast memory is to small to contain the output blocks of each successor, we consider the successors in non-decreasing index order (v_{j-1} is handled before v_j). CcMode aims at providing a more realistic comparison baseline than 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 Knights Landing (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 [6]. 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 [2]). Finally, 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 [25]. The STG set provides 180 randomly generated DAGs with different sizes ranging from 50 to 5000 nodes. We select two different 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.

Finally, 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:

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

We let the CCR vary in our experiments and 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, it means that 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 $\left[\frac{w_i^{\min} \times \beta_s}{s \times \text{CCR}}, \frac{w_i^{\max} \times \beta_s}{s \times \text{CCR}}\right]$.

6.2 Results

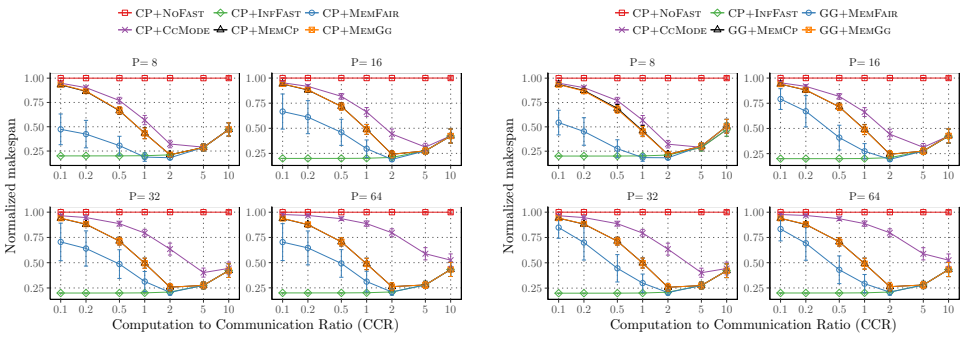
To evaluate the heuristics, we execute each heuristic 50 times with different random weights on the 20 graphs from each STG subset, hence each point is the average of 1000 executions. Then, we compute the average makespan over all the runs. All makespans are normalized with the baseline without fast 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

Sparse case. Figure 4a presents the normalized makespan of graphs of 50 nodes, and with 1GB 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 4b 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, we observe that 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 decreases, mainly because when P increases, the size of each fast memory slice decreases.

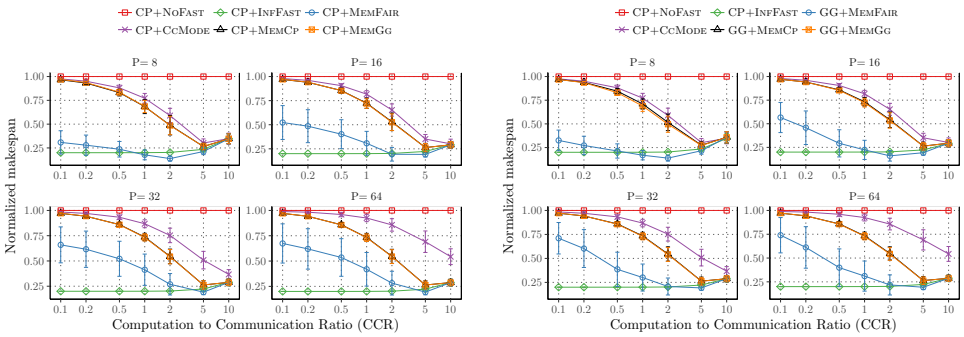


(a) CP (b) GG

Figure 4: Impact of the number of processors with 50 nodes and $S_f = 1$GB fast memory for CP and GG scheduling heuristics, and for the sparse case.

Figure 5 presents the normalized makespan of graphs with 100 nodes, and with 1GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64.The results are similar to the case with 50 nodes (see the Figure 4), the impact of the size of graphs is not strong,

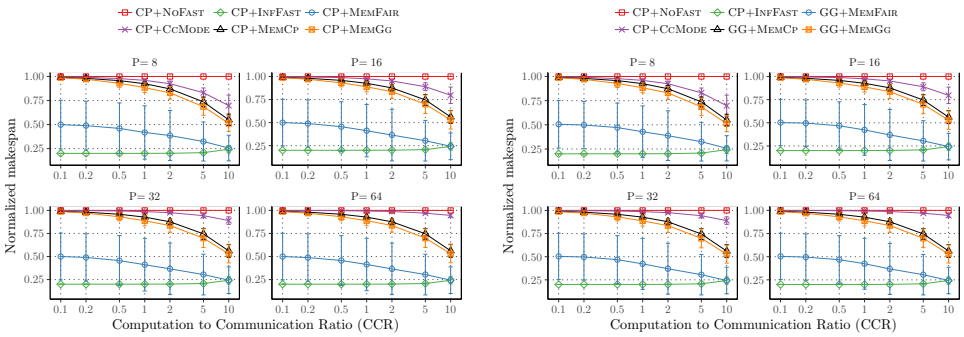
mainly because the performance are strongly linked to the CCR .



(a) CP (b) GG

Figure 5: Impact of the number of processors with 100 nodes and $S_f = 1$GB fast memory for CP and GG scheduling heuristics, and for the sparse case.

Dense case. Figure 6a presents the normalized makespan of dense graphs of 50 nodes, and with 1GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64.Compared to the sparse case (see Figure 4) all heuristics shows degraded performance, mainly due to the fact that dense graphs are larger than sparse graphs, in terms of memory usage. But, global performance are very good, with an average gain around 50% for the best combination.

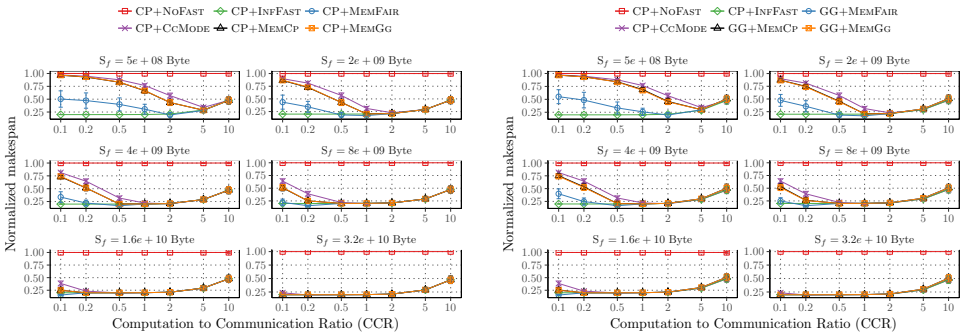


(a) CP (b) GG

Figure 6: Impact of the number of processors with 50 nodes and $S_f = 1$GB fast memory for CP and GG scheduling heuristics, and for dense case.

6.2.2 Impact of fast memory size

Sparse case. Figure 7 presents the results for graphs with 50 nodes, with 8 processors when we vary the fast memory size and the CCR. As always, we vary the CCR from 0.1 to 10 and the size of fast memory from 200MB to 16GB. Recall that, the fast memory bandwidth is set to 450 GB/s (five times faster). Both scheduling heuristics CP and GG show similar performance. Clearly, when the size of the memory is increasing, 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 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.



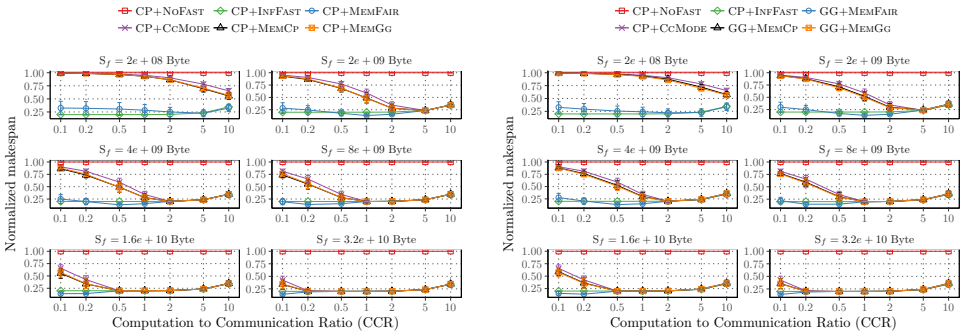
(a) CP (b) GG

Figure 7: Impact of fast memory size with 50 nodes and 8 processors for CP and GG scheduling heuristics, and for the sparse case.

Figure 8 presents the normalized makespan of graphs with 100 nodes, and with 1GB fast memory, when we vary the CCR from 0.1 to 10 and the number of processors from 8 to 64. The results with 100 nodes are similar to the results with 50 nodes, the memory mapping MemFair performs better with 100 nodes.

6.2.3 Impact of fast memory bandwidth

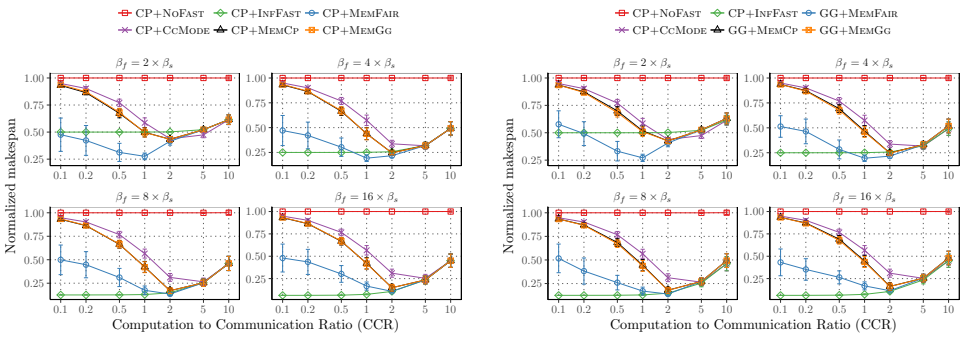
Sparse case. Figure 9 presents the results for graphs with 50 nodes, with 8 processors and 1GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow memory bandwidth. Both scheduling heuristics CP and GG exhibit similar performance when we vary the fast memory 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



(a) CP (b) GG

Figure 8: Impact of fast memory size with 100 nodes and 8 processors for CP and GG scheduling heuristics, and for the sparse case.

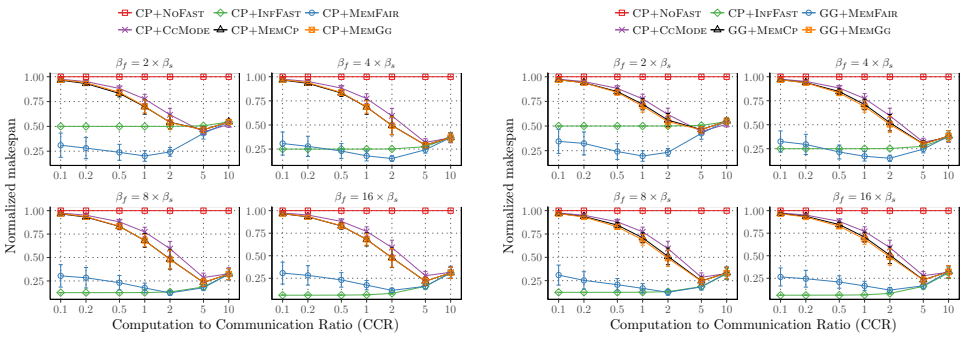
InfFast has a infinite fast memory with a finite bandwidth. When the bandwidth is too small compared to the slow memory bandwidth, saturating the fast memory leads to decreased performance because the fast memory bandwidth is shared by the number of tasks concurrently trying to gain access to it.



(a) CP (b) GG

Figure 9: Impact of fast memory bandwidth with 50 nodes, 8 processors, and S_f = 1GB for CP and GG scheduling heuristics, and for the sparse case.

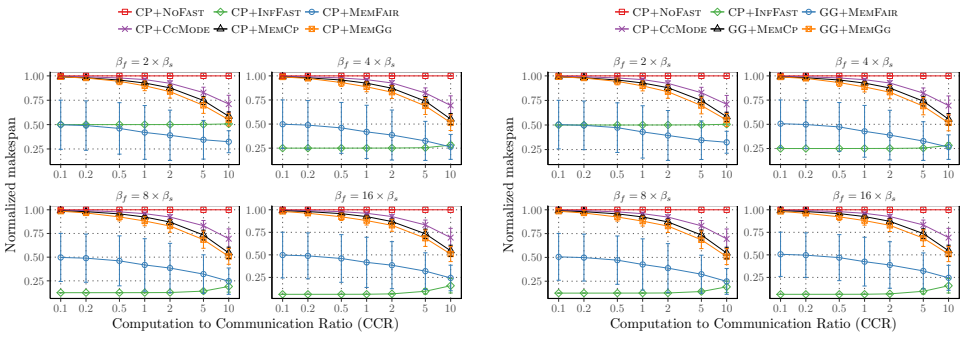
Figure 10 presents the results for graphs with 100 nodes, with 8 processors and 1GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow memory bandwidth. Results with 100 nodes and with 50 nodes present similar trends, the key point is when the CCR increases the graph no longer fits into the fast memory memory



(a) CP (b) GG

Figure 10: Impact of fast memory bandwidth with 100 nodes, 8 processors, and $S_f = 1$GB for CP and GG scheduling heuristics, and for the sparse case.

Dense case. Figure 11 presents the results for dense graphs with 50 nodes, with 8 processors and 1GB fast memory. The bandwidth of the fast memory is ranging from 2 times up to 16 times the slow memory bandwidth. We observe similar trends between dense and sparse case, when the CCR increases the performance of all heuristics increase as well. The combinations with MemFair perform the best, with an average gain around 50%.



(a) CP (b) GG

Figure 11: Impact of fast memory bandwidth with 50 nodes, 8 processors, and $S_f = 1$GB for CP and GG scheduling heuristics, and for the dense case.

6.2.4 Summary

All heuristics are very efficient compared 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 unidimensional (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             Tile_i^t ← Gauss-Seidel(Tile_{i-1}^t, Tile_i^{t-1}, Tile_{i+1}^{t-1});
```

At each step of the procedure 1D-GS, Tile_i^t is computed as a some 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 upon 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 12. We use this graph as input for the simulations and run the scheduling and mapping heuristics presented in Section 5.

As for the experiments, we extend the previous study developed for parallel stencil applications in [13] 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,

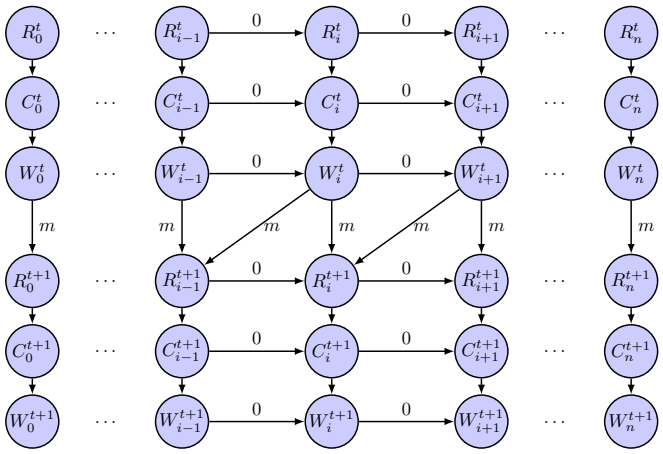


Figure 12: 1D stencil task graph, where t is the iteration index, i is the tile index, and m is the size of one tile.

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 12.

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

7.2 Results

For the benchmark runs, the platform is running CentOS 7.2, and experiments were repeated 10 times for accuracy. Figure 13 gives the performance of the benchmark against a baseline running entirely in slow memory with 64 threads. Figure 14 reports the results of the simulations for the same task graph, using the best heuristic CP +MemFair, on 64 threads.

We observe quite 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 from slow memory. We have a discrepancy here, since

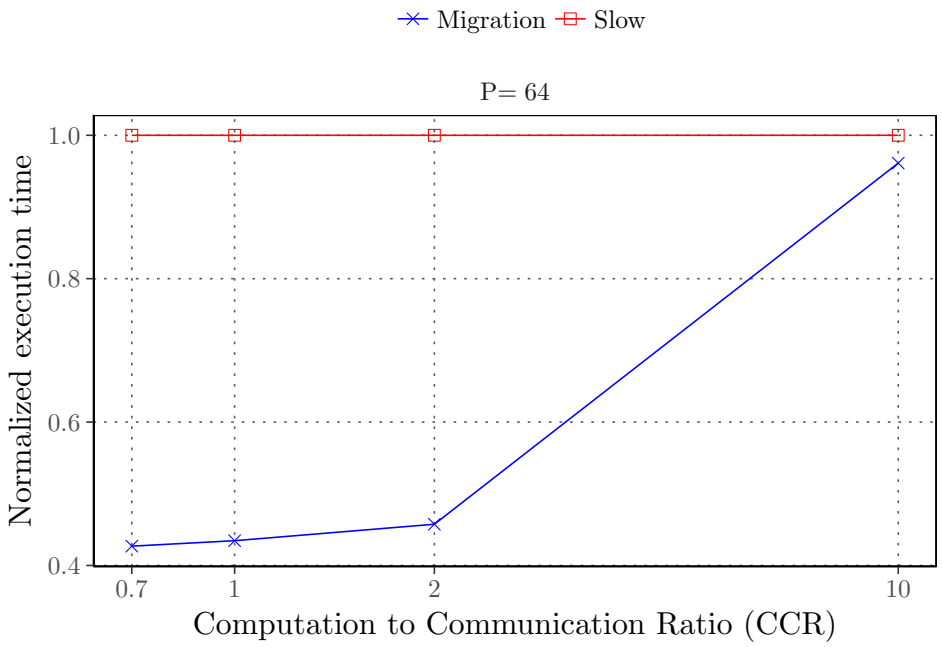


Figure 13: Performance of a 1D stencil running on a KNL with 64 threads.

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 to gain 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 to decide which 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 actual experiments on a KNL platform with a 1D Gauss-Seidel computational kernel, and compared the actual performance of a tuned memory mapping with that of the heuristics in simulation, thereby demonstrating the accuracy of the model and bringing an-

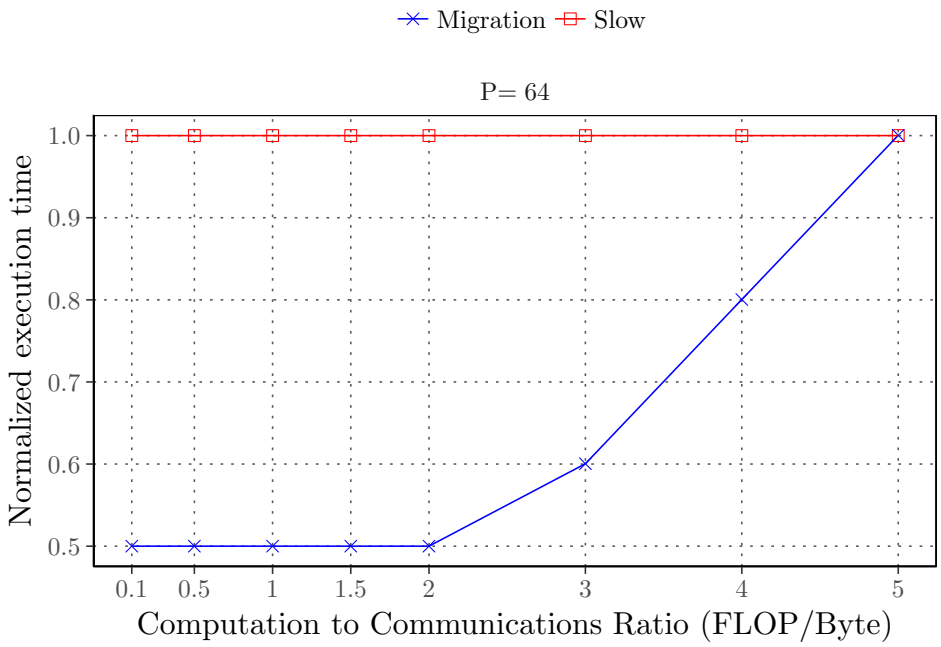


Figure 14: Performance of a 1D stencil according to the model, with 64 threads.

other 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 to extend the model in order to allow for moving data across both memory types. This is a challenging endeavor, because it requires to decide 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 towards a full understanding of scheduling problems on deep-memory architectures.

References

[1] Erich Strohmaier et al.: The top500 benchmark (2017) `https://www.top500.org/`.

[2] Intel: Intel Xeon Phi Processor: Performance Monitoring Reference Manual – Volume 1: Registers. Technical report, Intel (03 2017)

[3] Dongarra, J.: Report on the sunway taihulight system. Research report UT-EECS-16-742, Univ. Tennessee (2016) Available at `www.netlib.org`.

[4] Computing, P.: Zettascaler-2.0 configurable liquid immersion cooling system (2017)

[5] Asai, R.: Clustering Modes in Knights Landing Processors: Developer's Guide. Technical report, Colfax International (05 2016)

[6] Vladimirov, A., Asai, R.: MCDRAM as High-Bandwith Memory (HBM) in Knights Landing Processors: Developer's Guide. Technical report, Colfax International (05 2016)

[7] Unat, D., Shalf, J., Hoefler, T., Schulthess, T., (Editors), A.D., et al.: Programming Abstractions for Data Locality. Technical report (04 2014)

[8] Corporation, I.: Memkind: A user extensible heap manager. `https://memkind.github.io` (2018)

[9] Laboratory, L.A.N.: Simplified interface to complex memory. `https://github.com/lanl/SICM` (2017)

[10] Oden, L., Balaji, P.: Hexe: A toolkit for heterogeneous memory management. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS). (2017)

[11] Voskuilen, G., Rodrigues, A.F., Hammond, S.D.: 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. (2016) 204–207

[12] Servat, H., Peña, A.J., Llort, G., Mercadal, E., Hoppe, H., Labarta, J.: 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. (2017) 126–136

[13] Perarnau, S., Zounmevo, J.A., Gerofi, B., Iskra, K., Beckman, P.: Exploring data migration for future deep-memory many-core systems. In: IEEE Cluster. (2016)

[14] Chandrasekar, K., Ni, X., Kalé, L.V.: A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications. In: IEEE Int. Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA. (2017) 1293–1300

[15] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198

[16] Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. (May 2013) 1299–1308

[17] Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-aware task scheduling on multi-accelerator based platforms. In: IEEE Int. Conf. on Parallel and Distributed Systems. (Dec 2010) 291–298

[18] Aba, M.A., Zaourar, L., Munier, A.: Approximation algorithm for scheduling a chain of tasks on heterogeneous systems. In: European Conference on Parallel Processing, Springer (2017) 353–365

[19] NVIDIA: CUDA: Unified memory programming. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd` (2018)

[20] Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in cuda. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC). (Sept 2014) 1–6

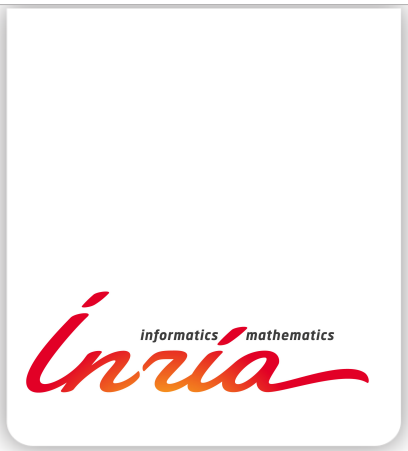
[21] Garey, M.R., Johnson, D.S.: Computers and Intractability, a Guide to the Theory of NP-Completeness. W.H. Freeman and Company (1979)

[22] Micheli, G.D.: Synthesis and Optimization of Digital Circuits. 1st edn. McGraw-Hill Higher Education (1994)

[23] Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems **13**(3) (Mar 2002) 260–274

[24] Benoit, A., Perarnau, S., Pottier, L., Robert, Y.: A performance model to execute workflows on high-bandwidth memory architectures. Research report RR-9165, INRIA (2018) Available at `hal.inria.fr`.

[25] Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling **5**(5) (2002) 379–394



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399