Throughput Optimization for Pipeline Workflow Scheduling with Setup Times

Anne Benoit[1], Mathias Coqblin[2], Jean-Marc Nicod[2], Laurent Philippe[2], and Veronika Rehn-Sonigo[2]

[1] LIP, ENS Lyon and Institut Universitaire de France
[2] FEMTO-ST Institute, CNRS/UFC/ENSMM/UTBM, Besançon

Abstract. We tackle pipeline workflow applications that are executed on a distributed platform with setup times. Several computation stages are interconnected as a linear application graph, and each stage holds a buffer of limited size where intermediate results are stored and a processor setup time occurs when passing from one stage to another. In this paper, we focus on interval mappings (consecutive stages mapped on a same processor), and the objective is the throughput optimization. Even when neglecting setup times, the problem is NP-hard on heterogeneous platforms and we therefore restrict to homogeneous resources. We provide an optimal algorithm for constellations with identical buffer capacities. When buffer sizes are not fixed, we deal with the problem of allocating the buffers in shared memory and present a $b/(b+1)$-approximation algorithm.

1 Introduction

In this paper, we consider pipeline workflow applications mapped on a distributed platform such as a grid. This kind of applications is used to process large data sets or data that are continuously produced by some source and produce some final results. The first stage of the pipeline is applied to an initial data to produce an intermediate result that is then sent to the next stage of the pipeline and so on until the final result is computed. Examples of such applications include image set processing where the different stages may be filters, encoders, image comparison or merging and video capture processing and distribution where codecs must be applied on the video flow before being delivered to some device. In this context, a first scheduling problem is to map the pipeline stages on the processors. Subhlock and Vondran [13,14] show that there exists an optimal interval mapping for a given pipeline and a given platform when communications and processors are homogeneous. An interval mapping is defined as a mapping where only consecutive pipeline stages are mapped on the same processor. However, the cost of switching between stages of the application on one processor is not taken into account. When a new data set arrives on the processor, the first local stage starts to process it as soon as the previous data set is output. Then this data set moves from stage to stage until the last local stage, and it is sent to the processor in charge of the following stage. So,

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 57–67, 2013.

at each step of the execution, we switch from one stage to the next one. As a result, if the cost of switching cannot be neglected, several setup times must be added to the processing cost. Benoit and Robert [4] prove that the basic interval mapping problem is NP-hard as soon as communications or computations are heterogeneous, even without setup times. For this reason, we restrict this work to homogeneous platforms.

The problem of reconfiguration that requires a setup time has been widely studied, and covers a lot of domains (see survey [1]). For instance, Zhang et al. [15] address the problem of wafer-handling robot calibration in semiconductor factories. They propose a low-cost solution to reduce the robot end effector tolerance requirements and thus the calibration times. A solution based on ant colony optimization is proposed in [9,10] to reduce the setup costs in batch processing of different recipes of semiconductors. In the scope of micro-factories, due to the cost of design and production of micro-assembly cells, micro-assembly cells are being designed with a modular architecture that can perform various tasks, at the cost of a reconfiguration time between them [8]. In the domain of pure computing, setup times may appear when there is a need to swap resources, or to load a different program in memory, e.g., to change the compiler in use [2]. Some authors have also shown interest in using buffers to stock temporary results after each stage of the pipeline, in order to reduce the amount of performed setups. Bryan and Norman [7] consider a flowshop wherein a job consists of m stages mapped on m processors, and a processor must be reconfigured after each job to process the next one (in their example, the clean-out of a reactor in a chemical processing facility). They acknowledge that the problem of sequence-dependent setup times in which a setup time depends on the previous stage and the next one is NP-hard and they propose several heuristics. Luh et al. [11] study scheduling problems in the manufacturing of gas insulated switchgears. The problems involve significant setup times, strict local buffer capacities, and few possible processing routes.

However, most of those researches consider that the number of processors is large enough to map each stage on only one processor (one-to-one mapping) and no reconfiguration is required before the next batch. Note that the one-to-one mapping problem can be solved in polynomial time provided that communications are homogeneous [4]. In our approach, we consider that the number of stages is greater than the number of processors. We therefore focus on interval mappings, where several consecutive stages are mapped onto the same processor.

The difficulty of the mapping problem is twofold. First, as in classical interval mapping one has to decide how to cut the different stages of the pipeline workflow into intervals, hence which stages are mapped onto the same processor. Second, the schedule inside a processor has to be fixed. Switching continuously between stages may lead to a drop in performance (due to the setup times), whereas buffering the data and defining a schedule for the processing of stages may limit the number of setups. Hence buffers are introduced to store intermediate results. This makes it possible to perform one stage several times before switching to the next one.

Starting from the interval mapping results, we tackle in this paper the problem of optimizing the cost of switching between stages mapped on the same processor depending on the buffer sizes. In a first step, we consider the single-processor scheduling problem where a single processor has to process several consecutive and dependent pipeline stages. Each stage is associated with a buffer. Usually, these buffers are limited by the available memory in the system and the buffer size hence influences the possible schedules as it limits the number of repetitions. Several other parameters are also taken into account as the duration of each stage's setup, the homogeneity or heterogeneity of buffers, and the available memory. Once the single-scheduling problem has been dealt with, we study in a second step the overall execution of the pipeline (in terms of throughput). Because of buffer utilization, data is treated and forwarded in batches, which leads to a data flow in waves. This particular behavior has to be taken into account in the solution.

We formally define the optimization problem in Section 2. The main contributions follow: (i) we provide optimal algorithms when buffers are of fixed size (Section 3); and (ii) we discuss how to allocate memory to buffers on a single processor in Section 4, both from a theoretical perspective (optimal algorithm in some cases), and from a practical point of view ($b/(b+1)$-approximation algorithm). Finally, we conclude and discuss future work in Section 5.

2 Framework

The application is a linear workflow application, or pipeline. It continuously processes a large amount of consecutive data sets. Formally, a pipeline is expressed as a set S of n stages: $S = \{S_1, \ldots, S_n\}$. Each data set is fed into the pipeline and traverses the pipeline from one stage to another until the entire pipeline is passed. A stage S_i receives a task of size δ_i from the previous stage, treats the data set which takes a number of w_i computations, and outputs data of size δ_{i+1}. The output data of stage S_i is the input data of the next stage S_{i+1}.

The target platform is a set P of p homogeneous processors $P = \{P_1, \ldots, P_p\}$ fully interconnected as a clique. Each processor P_u has a processing speed (or velocity) v, expressed in instructions per time unit, and a memory of size M. It takes X/v time units for P_u to execute X floating point operations. Each processor P_u is interconnected with a processor P_v via a bidirectional communication link of bandwidth β (expressed in input size units per time unit). We work with a linear cost model for communications, so it takes X/β time units to send or receive a message of size X from processor P_u to processor P_v. Furthermore communications are based on the bi-directional one-port model [5,6], where a given processor can send and receive at the same time, but for both directions can only support one message at a time. Distinct processor pairs can however communicate in parallel. Communications are non-blocking, i.e., a sender does not have to wait for its message to be received as it is stored in a buffer, and the communications can be covered by the processing times provided that a processor has enough data to process.

Each processor can process data sets from any stage. However to switch from an execution stage S_i to the next stage S_j, the processor P_u has to be recon-

figured. This induces setup times, denoted as st. The level of heterogeneity in setup times leads to different models: *uniform setup times* (st), where all setup times are fixed to the same value, *sequence-independent setup times* (st_i), where the setup time only depends on the next stage S_i to which the processor is reconfigured, and *sequence-dependent setup times* ($st_{i,j}$) that depend on both the current stage S_i and the next stage S_j. The problem with sequence-dependent setup times requires to look for the best setup order in a schedule to minimize the impact of setup times. This has already been proven to be NP-hard, and can be modeled as a Traveling Salesman Problem (TSP) [12]. Hence we will not study this problem in this paper, and we focus on st and st_i instead.

To execute a pipeline on a given platform, each processor is assigned an interval of consecutive stages. Hence, we search for a partition of $[1..n]$ into $m \leq p$ intervals $K_k = [I_k, J_k]$ such that $I_k \leq J_k$ for $1 \leq k \leq m, I_1 = 1,\ I_{k+1} = J_k + 1$ for $1 \leq k \leq m-1$ and $J_m = n$. Interval K_k is mapped onto a processor P_u. Once the mapping is fixed, the processor internal schedule has to be decided, since it influences the global execution time. Each processor is indeed able to perform sequentially its allocated stages. However, setup times are added each time a processor switches from one stage to another. To reduce setup times a processor may process several consecutive data sets for a same stage. The intermediate results are stored in buffers, and each stage S_i mapped on P_u has an input buffer B_i of size $m_{i,u}$.

The sizes of these input buffers depend on the memory size M available on P_u and on the number of allocated stages, as well as on the input data sizes. The capacity $b_{i,u}$ of buffer B_i is the number of input data sets that the buffer is able to store within the allocated memory $m_{i,u}$. Hence, a processor is able to process data sets for a stage S_i as long as B_i is not empty, and B_{i+1} is not full. Actually if S_i is the last stage of the interval mapped on P_u, we allocate an output buffer BO_u of size mo_u with a capacity bo_u.

The objective function is to maximize the throughput ρ of the application, $\rho = \frac{1}{\mathcal{P}}$, where $\mathcal{P}$ is the average period of time between the output of two consecutive data sets. Therefore, we aim at minimizing the period of the application. Since our framework model allows us to cover communication time by computation time, $\mathcal{P}$ is formally defined by: $\mathcal{P} = \max_u \Big(\max \big(in(u), cpu(u), out(u)\big)\Big)$, where $in(u)$, $cpu(u)$, $out(u)$ are respectively the mean time to input, process and output one data set onto $P_u \in P$. In the next two sections, we explicitly evaluate the application period depending on fixed or variable buffer sizes.

3 Fixed Buffer Sizes

In this section, we deal with the scheduling problem with fixed buffer sizes for both single and multiple processors. We consider that buffers that are allocated on the same processor P_u are homogeneous, i.e., they have the same capacity b_u.

Single Processor Scheduling ($b_i = b$). With a single processor, the mapping is known, since stages S_1 to S_n form a single interval. We propose a polynomial time greedy algorithm to solve the problem of single processor scheduling and

prove its optimality. The idea is to maximize the number of data sets that are processed for a stage between each setup. This is done by selecting a stage for which the input buffer is full and the output buffer is empty, so that we can compute exactly b data sets, where b is the number of data sets that fits in each buffer. Therefore, we compute b data sets for stage S_1, hence filling the input buffer of S_2, and then perform a setup so that we can compute b data sets for stage S_2, and so on, until these b data sets exit the pipeline. Then we start with stage S_1 again. We call the proposed algorithm GREEDY-B in the following.

To prove the optimality of GREEDY-B, we introduce a few definitions: during the whole execution, for $1 \leq i \leq n$, $nbout$ is the total number of data sets that are output; $nbst_i$ is the number of setups performed on stage S_i; $nbst = \sum_{i=1}^{n} nbst_i$ is the total number of setups; and $nbcomp_i$ is the average number of data sets processed between two setups on stage S_i. We have for $1 \leq i \leq n$: $nbcomp_i = \frac{nbout}{nbst_i}$, $nbst_i = \frac{nbout}{nbcomp_i}$, and $nbst = \sum_{i=1}^{n} \frac{nbout}{nbcomp_i}$.

Proposition 1. *For each stage S_i ($1 \leq i \leq n$), $nbcomp_i \leq b$.*

Proof. For each stage S_i, the number of data sets that can be processed after a setup is limited by its surrounding buffers. Once a setup is done to any stage S_i, it is not possible to perform more computations than there are data sets or than there is room for result sets. Since all buffers can contain exactly b data sets, we have $nbcomp_i \leq b$.

Proposition 2. *On a single processor with homogeneous buffers, the period can be expressed as $\mathcal{P} = \sum_{i=1}^{n} \frac{w_i}{v} + \sum_{i=1}^{n} \frac{st_i}{nbcomp_i}$.*

Proof. The period is the total execution time divided by the total number of processed data sets $nbout$. The execution time is the sum of the time spent computing, and the time to perform the setups. The computation time is the time to compute each stage once (w_i/v for stage S_i), multiplied by the number of data sets $nbout$. The reconfiguration time is the sum of the times required to perform each setup: $nbst_i \times st_i$. Therefore, the period can be expressed as $\mathcal{P} = \frac{1}{nbout}\left(\sum_{i=1}^{n} \frac{w_i}{v} \times nbout + \sum_{i=1}^{n} st_i \times nbst_i\right)$, and we conclude the proof by stating that $nbst_i = \frac{nbout}{nbcomp_i}$.

Lemma 1. *On a pipeline with homogeneous buffers, the lower bound of the period on a processor is $\mathcal{P}_{min} = \sum_{i=1}^{n} \frac{w_i}{v} + \sum_{i=1}^{n} \frac{st_i}{b}$.*

Proof. The result comes directly from Propositions 1 and 2: $\mathcal{P} = \sum_{i=1}^{n} \frac{w_i}{v} + \sum_{i=1}^{n} \frac{st_i}{nbcomp_i} \geq \sum_{i=1}^{n} \frac{w_i}{v} + \sum_{i=1}^{n} \frac{st_i}{b} = \mathcal{P}_{min}$.

Theorem 1. *The scheduling problem on a single processor can be solved in polynomial time, using the GREEDY-B algorithm.*

Proof. It is easy to see that GREEDY-B is always performing b computations between two setups, and therefore $nbcomp_i = b$ for $1 \leq i \leq n$. Therefore, the period obtained with this algorithm is exactly $\mathcal{P}_{min}$, which is a lower bound on the period and hence it is optimal.

Multi Processor Scheduling ($b_i = b_u$). The interval mapping problem on fully homogeneous platforms without setup times can be solved in polynomial time using dynamic programming [13,14]. We propose the use of this dynamic programming algorithm for homogeneous platforms, taking into account the setup times in the calculation of a processor's period. To be precise, the calculation of the period is the one obtained by the GREEDY-B algorithm.

Let $c(j,k)$ be the optimal period achieved by any interval mapping that maps stages S_1 to S_j and that uses at most k processors. Let $per(i,j)$ be the average period of the processor on which stages S_i to S_j are mapped. Note that $per(i,j)$ takes the communication step into account. We have:

$$c(j,k) = \min_{1 \leq l \leq j-1} (\max(c(l,k-1), per(l+1,j))),$$

with the initial condition $c(j,k) = +\infty$ if $k > j$. Given the memory M, we can compute the corresponding buffer capacity $b(i,j) = \left\lfloor \frac{M}{\sum_{k=i}^{j+1} \delta_k} \right\rfloor = b_u$, since we assume identical buffer capacities. Therefore:

$$per(i,j) = \max \left(\frac{\delta_i}{\beta}, \sum_{k=i}^{j} \left(\frac{w_k}{v} + \frac{st_k}{b(i,j)} \right), \frac{\delta_{j+1}}{\beta} \right)$$

The main difference with the ordinary use of the dynamic programming algorithm is that P_u consumes b_u input data sets or outputs b_u data sets in waves because of GREEDY-B. So $c(n,p)$ returns the optimal period if and only if the period is actually dictated by the period of the slowest processor, i.e., the slowest processor cannot be in starvation or in saturation because of intermittent access to the input/output buffers. The following theorem ensures that this is true:

Theorem 2. *On a pipeline with inner-processor homogeneous buffer capacities b_u, the period $\mathcal{P}$ is dictated by the period of the slowest processor.*

The proof can be found in the companion research report [3]. It is a proof by induction, and several cases need to be discussed considering a pipeline of processors: we prove that the slowest of the processors is never slowed down either by a lack of data inputs or by a saturation of its output buffer.

Single Processor Scheduling with Different Buffer Sizes. We complete the fixed buffer size study by considering buffers with different sizes. GREEDY-B chooses either a stage whose input buffer is full and we have enough space to fully empty it, or a stage whose output buffer is empty and we have enough data sets to compute in order to fully fill it. That way, we still maximize the amount of data sets processed after each setup: we are limited by the lowest capacity buffer, which is either a fully emptied input buffer, or a fully filled output buffer. It may not return an optimal schedule in the general case, but we can prove its optimality in the case of *multiple buffers*, i.e., each buffer capacity is a multiple of the capacities of both its predecessor and its successor: for $1 \leq i \leq n$, $\min(b_i, b_{i+1}) | \max(b_i, b_{i+1})$.

Theorem 3. *The scheduling problem with multiple buffers on a single processor can be solved in polynomial time, using the GREEDY-B algorithm.*

The proof of this theorem can be found in the companion research report [3]. Note that GREEDY-B is not optimal for multiple processor scheduling with multiple buffers.

4 Variable Buffer Sizes

In this section, we tackle the problem of allocating the buffers for all stages on a single processor P from an available memory M. We first focus on platforms with homogeneous data input sizes ($\delta_i = \delta$) and setup times ($st_i = st$).

Allocation Algorithm. If n stages are mapped on one processor then it needs $n+1$ buffers. Given the memory M and the size of the data δ, if we want all buffers to contain the same number of data sets, then the maximum number of data sets that can fit in each buffer can be computed as $b = \left\lfloor \frac{M}{(n+1)\delta} \right\rfloor$.

The ALL-B algorithm allocates memory for each buffer according to this uniform distribution. The actual memory allocated for each buffer is $m_i = m = b\delta = \left\lfloor \frac{M}{n+1} \right\rfloor$. The memory used by this allocation is then $(n+1)\delta \times b \leq M$, and we call $\mathcal{R} = M - (n+1)\delta \times b$ the *remainder* of memory after the allocation, i.e., the unused part of the memory. We prove that this allocation algorithm is optimal if the remainder is lower than δ.

Theorem 4. *The algorithm ALL-B is optimal on a single processor (i.e., the period is minimized with this allocation) when* $\mathcal{R} = M - (n+1)\delta \times \left\lfloor \frac{M}{(n+1)\delta} \right\rfloor < \delta$.

The proof can be found in the companion research report [3]. It is a proof by induction on n, and by expressing the general period (with any buffer sizes), we prove that the minimum is reached when all buffers are identical. The idea behind this proof is that, starting from a uniform allocation (same buffer sizes), raising the size of a buffer means reducing the size of another. The period is based on the amount of computations done before a setup (the $\frac{st}{\min(b_i, b_{i+1})}$ part of the period), and this value depends on the minimum of two consecutive buffers. Therefore we would need to raise more buffers than we lower to balance this value.

Memory Remainder. If there is a remainder in the memory after the allocation of buffers ALL-B, it is under certain conditions possible to use this remainder to increase the size of some buffers. It may also be possible to have another allocation, not based on ALL-B, that would make better or full use of the memory. In both cases, the period achieved by some scheduling algorithm may be lower than the one we have.

Proposition 3. *Given an application with homogeneous setup times st and input sizes δ, ALL-B may not give an optimal solution if $\mathcal{R} \geq \delta$.*

Proof. Let us consider a single processor, with a memory $M = 20$, and a speed $v = 1$. A total of $n = 6$ stages are mapped on this processor, and we have $\delta = w = st = 1$. There are seven buffers, and therefore ALL-B returns buffers of size $b = 2$, and the remainder is $\mathcal{R} = 20 - 2 \times 7 = 6$. The optimal period using this distribution is obtained by scheduling the stages with the GREEDY-B algorithm (see Theorem 1), and therefore:
$\mathcal{P} = \sum_{i=1}^{6} \frac{w_i}{v} + \sum_{i=1}^{6} \frac{st}{b} = 6 + \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) = 9.$

However, let us consider the following allocation: $b_1 = b_2 = b_3 = b_4 = 2$ and $b_5 = b_6 = b_7 = 4$. This allocation uses all the memory, and it corresponds to the definition of *multiple buffers*. Therefore, the optimal period is obtained by scheduling the stages with the GREEDY-B algorithm, and:
$\mathcal{P} = \sum_{i=1}^{6} \frac{w_i}{v} + \sum_{i=1}^{6} \frac{st}{\min(b_i, b_{i+1})} = 6 + \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4}\right) = 8.5.$
This allocation leads to a smaller period than ALL-B, which concludes the proof.

We propose an heuristic to deal with the memory remainder created by ALL-B ($\forall 1 \leq i \leq n+1, b_i = b$). In some cases, it is possible to use $\mathcal{R}$ to increase the size of several (but not all) buffers. According to Proposition 3, the use of this remainder may lead to a decrease of the period. We restrict to the construction of *multiple buffers* as defined above, so that we are able to find optimally the period thanks to the GREEDY-B algorithm. Hence, if there is enough memory to increase the size of buffers by steps of b, and if there is as least $2b\delta$ memory left, then the size of two consecutive buffers can be doubled, resulting in halving the number of setups for the corresponding stage.

The heuristic, that we call H-REMAIN, starts off by doubling the size of the two last buffers if there are $2b\delta$ memory units left, then will continue to increase the capacity of the adjacent buffers by b as long as $b\delta$ memory units are still available. Note that since $\mathcal{R} < (n+1)\delta$, the algorithm is guaranteed to end before having doubled the size of all buffers.

Given the available memory M, $\mathcal{P}_b(M)$ is the period obtained if $\forall i \in [1, n+1], b_i = b$; $\mathcal{P}_{algo}(M)$ is the period obtained by our heuristic; and $\mathcal{P}_{opt}(M)$ is the optimal (minimal) period that can be achieved with memory M.

We compute the value of b obtained by the ALL-B algorithm, and therefore $M = b(n+1)\delta + \mathcal{R}$, with $\mathcal{R} < (n+1)\delta$. It has already been proved (see Theorem 4) that if there is no remainder after ALL-B, $\mathcal{P}_b(M)$ is optimal. More formally, $M = b(n+1)\delta \iff \mathcal{P}_b(M) = \mathcal{P}_{opt}(M)$. We define $M^* = (b+1)(n+1)\delta = M + (n+1)\delta - \mathcal{R}$. With a memory M^*, there is also no remainder and $\mathcal{P}_{b+1}(M^*) = \mathcal{P}_{opt}(M^*)$. We first prove that both $\mathcal{P}_{algo}(M)$ and $\mathcal{P}_{opt}(M)$ can be bounded by $\mathcal{P}_b(M)$ and $\mathcal{P}_{b+1}(M^*)$ respectively:

Lemma 2. *We have* $\mathcal{P}_b(M) \geq \mathcal{P}_{algo}(M) \geq \mathcal{P}_{opt}(M) \geq \mathcal{P}_{b+1}(M^*)$.

Proof. By definition, we have $\mathcal{P}_{algo}(M) \geq \mathcal{P}_{opt}(M)$. For the upper bound, H-REMAIN is potentially improving $\mathcal{P}_b(M)$ by exploiting the remainder, and the period cannot be increased by the allocation of the remainder of the memory.

For the lower bound, note that $\mathcal{P}_{b+1}(M^*)$ is the optimal period with memory $M^* > M$, and therefore $\mathcal{P}_{opt}(M)$ cannot be better, otherwise we would have a better solution with M^* that would not use all memory.

Theorem 5. *The two algorithms ALL-B and H-REMAIN are $\frac{b+1}{b}$-approximation algorithms.*

Proof. Let $W = \sum_{i=1}^{n+1} \left(\frac{w_i}{v}\right)$. We have $\mathcal{P}_b(M) = W + \frac{(n+1)st}{b}$, and $\mathcal{P}_{b+1}(M^*) = W + \frac{(n+1)st}{b+1}$. Therefore,

$$\frac{\mathcal{P}_b(M)}{\mathcal{P}_{b+1}(M^*)} = \frac{W + \frac{(n+1)st}{b}}{W + \frac{(n+1)st}{b+1}} \leq \frac{\frac{(n+1)st}{b}}{\frac{(n+1)st}{b+1}} = \frac{b+1}{b} ,$$

since $W > 0$ and $\frac{(n+1)st}{b+1} \leq \frac{(n+1)st}{b}$. Finally, thanks to Lemma 2, we have:

$$\mathcal{P}_{algo}(M) \leq \mathcal{P}_b(M) \leq \frac{b+1}{b}\mathcal{P}_{b+1}(M^*) \leq \frac{b+1}{b}\mathcal{P}_{opt}(M) ,$$

which concludes the proof (recall that $\mathcal{P}_b(M)$ is the period obtained by algorithm ALL-B). Note that the worst approximation ratio is achieved for $b = 1$, and then we have 2-approximation algorithms. However, when b increases, the period achieved by the algorithms tend to the optimal solution.

With Heterogeneous Setup Times or Data Input Sizes (st_i, δ_i). The case of heterogeneous setup times (st_i) is kept for future work, since it turns out to be much more complex. Indeed, allocating buffers while taking setup times into account requires to prioritize higher setup times by allocating larger buffer capacities. However, this requires both the input and output buffers of the corresponding stage to be larger, and it will inevitably lead to side effects on surrounding stages.

For heterogeneous data input sizes (δ_i), we can use a variant of the ALL-B algorithm to allocate buffers of identical capacities, in terms of data sets: $b_i = \left\lfloor \frac{M}{\sum_{k=1}^{n+1} \delta_k} \right\rfloor = b$. In this case, the memory used is $\sum_{i=1}^{n+1} b \times \delta_i \leq M$, and the remainder is $\mathcal{R} = M - \sum_{i=1}^{n+1} b \times \delta_i$. However, even if there is no remainder, the allocation may not be optimal:

Let us consider a single processor, with a memory $M = 301$, speed $v = 1$. There are $n = 4$ stages with $w = st = 1$. The different input sizes are: $\delta_1 = 20, \delta_2 = 20, \delta_3 = 1, \delta_4 = 1, \delta_5 = 1$. ALL-B returns buffers of size $b = 7$, and the remainder is $\mathcal{R} = 301 - (20 \times 7 + 20 \times 7 + 1 \times 7 + 1 \times 7 + 1 \times 7) = 0$. The optimal period using this distribution is obtained by scheduling the stages with the GREEDY-B algorithm (see Theorem 1), and therefore:
$\mathcal{P} = \sum_{i=1}^{4} \frac{w_i}{v} + \sum_{i=1}^{4} \frac{st}{b} = 4 + \left(\frac{1}{7} + \frac{1}{7} + \frac{1}{7} + \frac{1}{7}\right) = 4.571$.

However, let us consider the following allocation: $b_1 = b_2 = 6$ and $b_3 = b_4 = b_5 = 18$. This allocation uses less memory, yet has way higher capacity buffers for b_3 to b_5, with the only trade-off being the reduction of the capacity of b_1 and b_2 by one. This allocation corresponds to the definition of multiple buffers. Therefore, the optimal period is obtained by scheduling the stages with GREEDY-B, and
$\mathcal{P} = \sum_{i=1}^{4} \frac{w_i}{v} + \sum_{i=1}^{4} \frac{st}{\min(b_i, b_{i+1})} = 4 + \left(\frac{1}{6} + \frac{1}{6} + \frac{1}{18} + \frac{1}{18}\right) = 4.444$.
This allocation leads to a smaller period than ALL-B.

5 Conclusion

In this paper, we present solutions to the problem of optimizing setup times and buffer use for pipeline workflow applications. For the problem of fixed buffer sizes of identical size within a same processor, we provide an optimal greedy algorithm for a single processor, and a dynamic programming algorithm for multiple processors. In the latter case, the application period is equal to the period of the slowest processor. In the case of variable buffer sizes, we tackle the problem of distributing the available processor memory into buffers such that the period is minimized. When the memory allocation results in no remainder (the whole memory is used), the algorithm turns out to be optimal, and we propose some approximation algorithms for the other cases.

In future work, we plan to consider sequence-dependent setup times ($st_{i,j}$), a problem that is already known to be NP-complete. We envisage the design of competitive heuristics, whose performance will be assessed through simulation. Furthermore, for the st_i case, we plan to investigate the memory allocation problem on a single processor. On the long term, we will consider the case of heterogeneous buffer capacities b_i. This case is particularly interesting, as the buffer allocation heuristics lead to heterogeneous buffer sizes.

References

1. Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M.: A survey of scheduling problems with setup times or costs. European J. of Op. Research 187(3), 985–1032 (2008)
2. Allahverdi, A., Soroush, H.: The significance of reducing setup times/setup costs. European Journal of Operational Research 187(3), 978–984 (2008)
3. Benoit, A., Coqblin, M., Nicod, J.M., Philippe, L., Rehn-Sonigo, V.: Throughput optimization for pipeline workflow scheduling with setup times. Research Report 7886, INRIA (2012), `http://graal.ens-lyon.fr/~abenoit/papers/RR-7886.pdf`
4. Benoit, A., Robert, Y.: Mapping pipeline skeletons onto heterogeneous platforms. J. Parallel and Distributed Computing 68(6), 790–808 (2008)
5. Bhat, P., Raghavendra, C., Prasanna, V.: Efficient collective communication in distributed heterogeneous systems. In: 19th ICDCS 1999, pp. 15–24 (1999)
6. Bhat, P., Raghavendra, C., Prasanna, V.: Efficient collective communication in distributed heterogeneous systems. JPDC 63, 251–263 (2003)
7. Norman, B.A.: Norman: Scheduling flowshops with finite buffers and sequence-dependent setup times. Comp. & Indus. Engineering 36(1), 163–177 (1999)
8. Gendreau, D., Gauthier, M., Hériban, D., Lutz, P.: Modular architecture of the microfactories for automatic micro-assembly. Journal of Robotics and Computer Integrated Manufacturing 26(4), 354–360 (2010)
9. Li, L., Qiao, F.: Aco-based scheduling for a single batch processing machine in semiconductor manufacturing. In: IEEE Int. CASE 2008, pp. 85–90 (2008)
10. Li, L., Qiao, F., Wu, Q.: Aco-based scheduling of parallel batch processing machines to minimize the total weighted tardiness. In: Int. CASE 2009, pp. 280–285 (2009)
11. Luh, P.B., Gou, L., Zhang, Y., Nagahora, T., Tsuji, M., Yoneda, K., Hasegawa, T., Kyoya, Y., Kano, T.: Job shop scheduling with group-dependent setups, finite buffers, and long time horizon. Annals of Operations Research 76, 233–259 (1998)

12. Srikar, B., Ghosh, S.: A milp model for the n-job, m-stage flowshop with sequence dependent set-up times. Int. J. of Production Research 24(6), 1459–1474 (1986)
13. Subhlok, J., Vondran, G.: Optimal mapping of sequences of data parallel tasks. ACM SIGPLAN Notices 30(8), 134–143 (1995)
14. Subhlok, J., Vondran, G.: Optimal latency-throughput tradeoffs for data parallel pipelines. In: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, p. 71. ACM (1996)
15. Zhang, M., Goldberg, K.: Calibration of wafer handling robots: A fixturing approach. In: IEEE Int. CASE 2007, pp. 255–260 (2007)