1-optimality of static BSP computations: scheduling independent chains as a case study.

Alfredo Goldman*, Gregory Mounie, Denis Trystram
ID – IMAG
ZIRST, 51 avenue J. Kuntzmann
F-38330 Montbonnot Saint Martin, France

October 10, 2001

Abstract

The aim of this work is to study a specific scheduling problem under the machine-independent model BSP. The problem of scheduling a set of independent chains in this context is shown to be a difficult optimization problem, but it can be easily approximated in practice. Efficient heuristics taking into account communications are proposed and analyzed in this paper. We particularly focus on the influence of synchronization between consecutive supersteps. A family of algorithms is proposed with the best possible load-balancing. Then, a strategy for determining a good compromise between the two opposite criteria of minimizing the number of supersteps and a good balance of the load is derived. Finally, a heuristic which considers the influence of the latency is presented. Simulations of a large number of instances have been carried out to complement the theoretical worst case analysis. They confirm the very good behavior of the algorithms on the average cases.

keywords: Scheduling, synchronization, chains, BSP

*Supported by CNPq. On leave from DCC-IME-USP Brazil

1 Introduction

This paper studies the problem of scheduling independent chains on identical parallel processors. The chains are composed of tasks that have to be executed sequentially. This problem is motivated by the practical determination of the allocation of processes on a parallel distributed-memory machine. Each process may be a list of non-preemptive components that have to be executed sequentially. The processors are organized as a parallel distributed-memory machine, consisting of m identical processors. It is well established that in such systems, the communications are the most predominant parameters which influence the performances. Until recently, most of the works considered a standard computational model (the *delay* model) where the communications are taken into account explicitly by the time for transmitting an elementary piece of data from one processor to another [21]. Unfortunately, such a model is unrealistic and too difficult to be used practically for parallelizing actual applications. We consider here a machine-independent programming model based on *BSP* (Bulk Synchronous Parallel) [24]. Two main parameters can be considered in BSP, namely the global communication-synchronization overhead and the latency between two consecutive communication-synchronization events. One of the main motivations for introducing BSP was to separate the problems of load-balancing and optimization of communications.

The goal of this work is to show that efficient scheduling algorithms can be designed under the BSP model. We developed a theoretical analysis which is confirmed by practical simulation experiments at the end of the paper.

1.1 Related works

The identification and scheduling of potential parallelism is one of the main research fields in Parallel Computing. Just few years ago, with the lack of a standard and realistic cost model, most scheduling works were restricted to the analysis of virtual parallelism or addressed a specific parallel architecture. There are mainly two kinds of works related to this paper: scheduling chains on theoretical idealized models, and scheduling other graphs under realistic new parallel programming models.

The scheduling of chains considering theoretical models was studied a long time ago. The close problem of scheduling a set of independent tasks of any duration with preemption has been treated for uniform processors with zero cost communications [13] (the uniform model is the natural extension of the classical delay model where the processors have different speeds varying by a multiplicative factor). Assuming uniform processors with integer speed ratio and communication occurrences at any integer time, scheduling independent chains has been showed to be NP-hard in the strong sense for an arbitrary number of processors [19] even if the communication times are neglected.

The problem of scheduling under BSP has been investigated in particular by people in the Oxford and Paderborn groups: [20, 23] and [17, 1]. The scheduling of uniform directed acyclic graphs (also known as tightly-nested loops) was studied in [5, 6]. BSP algorithms for several classical problems involving matrices were presented in [20]. A new model (BSPRAM) and corresponding algorithms for butterfly directed acyclic graph, cube directed acyclic graph, dense matrix multiplication and sorting were presented in [23]. Note that some other works studied the scheduling problem of specific task graphs like FORK (flat trees) or general tree structures under computational models close to architectural constraints like LogP [18, 25, 26] or CGM [4, 8].

In this work, we focus our attention on scheduling specific precedence task graphs which consist of a set of independent chains of unit execution time tasks. This paper is an extension of the paper [11], where an algorithm with $\lceil \frac{m+1}{2} \rceil$ communications was proposed.

1.2 Organization of the paper

The basic problem of scheduling a set of independent chains and some definitions about BSP are introduced in Section 2. A discussion about the trade-off between load balancing and communication, and some preliminary results for solving this problem under the basic computational model (the *delay model*, described later in this paper) are also presented. Then, some specific instances of this problem are solved in Section 3 (for two extreme cases, namely for a two processors system and for an unlimited number of processors). The general case corresponding to an arbitrary number of processors is tackled in Section 4. Complexity results and worst case about communications are analyzed. A first approximation algorithm presented in [11] is recalled. Then, a family of algorithms with any fixed number of supersteps is proposed. We deduce a general algorithm which determines a trade-off between a low communication overhead and a good balance of the load. The influence of the latency is discussed in Section 5. Section 6 is devoted to experiments with simulations, where we evaluate how to perform the supersteps for obtaining good performances in average. Finally, some perspectives for future work are discussed in the conclusion.

2 Preliminaries

In this section the target problem of scheduling independent chains is presented and the definition of BSP together with its main properties are recalled. Some preliminary results assuming standard computational models, useful for the understanding of the further algorithms, are also described.

2.1 Description of the problem

The problem that we consider in this paper is to schedule k independent chains of tasks $\{ch_1, \ldots, ch_k\}$ on m identical processors under the BSP model. We will shortly call this problem *SIC* for Scheduling Independent Chains. The execution time of each task takes one unit of time (UET assumption). The length of chain ch_i will be denoted n_i (for $1 \leq i \leq k$). In the following, we will consider the case $k \geq 2$, since otherwise the problem is trivial and the best strategy is to schedule the entire single chain on one processor without communication. The total number of tasks is $n = \sum_{i=1}^{k} n_i$. Without loss of generality, the chains are assumed to be initially sorted in decreasing order of length, that is, $n_1 \geq \ldots \geq n_k$.

It is interesting to observe that in the SIC problem, task replication is not useful, as each task has at most one successor. This is usually not the case for general graphs.

According to most other studies related to scheduling in the context of Parallel Processing, we are interested in minimizing the *makespan*, which is the maximum completion time of the tasks of a parallel algorithm.

Definition 1 *The ideal makespan for SIC on m processors is defined by* $t^* = \max\{\lceil \frac{n}{m} \rceil, n_1\}$.

The ideal makespan is the maximum of the total workload distributed among the processors and the length of the longest chain. All the valid schedules have maximum completion time at least t^*.

Observe that the bound given by the ideal makespan is not tight, it cannot be accomplished in every case without splitting chains (which introduces communication between the different parts of a chain). Consider for instance the execution of three chains of same length on two processors as shown in figure 1 (in this figure and in all further figures, the dashed area represents idle times).

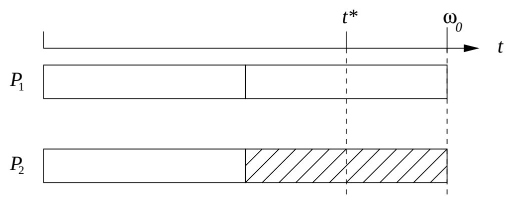

Figure 1: Scheduling three identical chains with no communication on two processors.

This remark motivates the following notation:

Notation 1 *ω_o denotes the best possible makespan to complete the execution of all the chains without communication.*

It is straightforward to show that $\omega_0 \geq t^*$.

2.2 Presentation of BSP

BSP (for Bulk Synchronous Parallel) is a computational model introduced for improving the scalability, the portability and the ease of developing application code on distributed-memory parallel systems [24]. It distinguishes between the two key factors of performance: computation and communication. BSP was introduced in order to separate the communication difficulties from the scheduling difficulties. It is becoming more and more popular for theoretical studies, providing a solid foundation for designing parallel algorithms. Moreover, some implementations of BSP library components have been developed on several platforms [16, 3].

2.2.1 Definition

Programming in BSP consists of a succession of supersteps. Each superstep may be divided into three activities:

- a computation phase where independent local computations run in parallel;
- a communication phase where the communications of data involve all the processors in a personalized all-to-all communication [12];
- a synchronization phase where a synchronization barrier allows the simultaneous starts of all the processors for the next superstep.

The synchronization guarantees the completion of communication and computation of the superstep. There are mainly two ways to perform the communication: asynchronously or as an efficient h-relation [12]. In the rest of the paper, we will shortly use CS for communication-synchronization.

The original BSP model [24] allows supersteps where the synchronization is done only among subsets of processors. However, the actual current implementations only provide global synchronization. See [22] for a complete discussion. According to most existing related works, we assume here a simplified BSP model where the communications involve all the processors, that is, the synchronization is always global. This is a usual practical assumption.

There are two intrinsic restrictions in the BSP model: each processor can send/receive a bounded number of messages during a superstep, and the messages have a limited size. With these restrictions, the cost of a single superstep is at worst the sum of three terms: the maximum cost of the local computations, the cost of a h-relation and the cost of the synchronization barrier. An h-relation is a message exchange operation where each processor receives or sends at most h messages. It should be noted that each message may be delivered in more than one communication packet. BSP considers two main parameters: g and l. g is known as the communication throughput ratio, it takes hg units of time to deliver a h-relation. l is defined as the cost of a synchronization barrier. In all the algorithms presented in this paper, each processor sends and receives at most one communication, so if b is the size of the largest communication (b-relation) we can assume that a CS takes a constant time denoted by C (equal to $bg + l$).

In the original BSP model proposed by Valiant [24], a periodicity parameter called latency (L) was introduced. In his model, in a superstep, after each period of L time units, a global check was made to determine whether the superstep was completed by all processors. In the variation of BSP model proposed by McColl [20], there is no reference to the periodicity, this approach was probably chosen because in actual computers the periodicity can be as small as the synchronization costs. But as periodicity can be interesting in the theoretical point of view, or even for future computers, in this work we chose to present a short discussion (Section 5) taking into account a minimum gap between two consecutive supersteps. So, instead of assuming a time multiple of L, this study is restricted to a minimum gap between consecutive supersteps.

To summarize, we will consider two main parameters: the time to communicate the data and synchronize the processors, C (a positive real number usually greater than the execution time, $C > 1$) and in addition the minimum time between two consecutive supersteps, denoted by λ (of course, $\lambda \geq C$).

In order to give an insight about these parameters, we have reported some practical values in table 1[1]. We verify that the synchronization cost has the main influence on parameter C.

	(Mflop/s)	g (flop/word)	l (flop)
8, 400Mhz PII, 100Mb ether.	88	30.9	18347
Cray T3D, 150Mhz (256 procs.)	12	2.4	387
T3E, 300Mhz(20 procs.)	47	1.63	880
IBM SP2, 66.7Mhz (8 procs.)	26	11.4	5412
Parsytec (8 procs.)	19.3	25.4	29080

Table 1: Floating point rate by processor, and normalized values of g and l for several machines.

In order to simplify our analysis, we suppose w.l.o.g. that the last superstep of a program does not require CS. So a program with s supersteps has a CS time equal to $(s-1)C$.

2.2.2 Example

The example depicted in figure 2 shows an execution under BSP for a simple precedence task graph (which is the usual representation of oblivious programs: the instructions are represented by nodes, and the precedence relations among the instructions are represented by arcs). On the Gantt chart, the communications are represented in grey, the synchronization phases are in black (as pointed out before, in actual machines the synchronization phase can be about two orders of magnitude greater than the communication phase for small messages) and the idle times are in dashed areas.

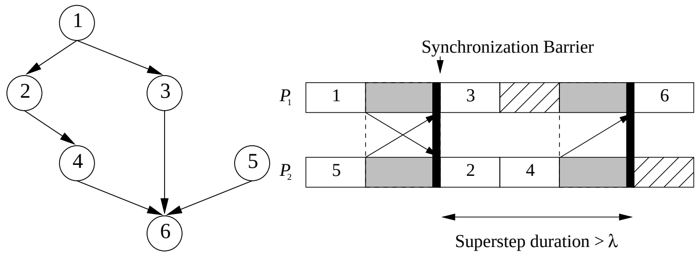

Figure 2: Schedule of a UET graph under BSP.

[1] These values were found in `www.BSP-Worldwide.org/implmnts/oxtool/params.html`

2.3 Load balancing versus communication

In a parallel distributed memory machine with m processors, the parallel execution time for executing a program can be estimated [2] as:

$$t_m = \frac{t_1}{m} + (t_I + t_c)$$

where t_1 is the sequential time of the program (in our case equal to the total number of tasks), t_I corresponds to the sum of idle time over all processors divided by m, and t_c is the time for performing communication and synchronization. Given m, we are interested in minimizing t_m. As the sequential time and the number of processors are fixed, this corresponds to minimizing the overhead $(t_I + t_c)$.

If $t_c = 0$, that is just one superstep in the BSP model (on the BSP model t_c is discrete), it will be shown later in this paper that the minimization problem is NP-hard. Otherwise, increasing t_c (and thus, the number of supersteps on BSP) might reduce t_I. In this paper, we will study how to minimize the idle time by increasing the communication time. We will give closed formulas to minimize (globally) the idle time, and to minimize the idle time for a given number of supersteps (thus, bounding the communication time). We will also discuss the trade-off between a good balance of the load and a low communication overhead.

2.4 SIC under basic computational models

Some papers addressing the problem of scheduling a set of independent chains have been published before. This problem has some practical interests since it corresponds to distributing the execution of sequential work (for instance, while using library components like BLAS [9]).

Most of these papers assume simple computational models. Either communications are simply ignored, or they are taken into account with variations around the standard delay model introduced by Rayward-Smith [21], in which there is no synchronization among the communications, and the transmission of messages can be overlapped by local computations. Communications are assumed to be constant and equal to the elementary computational time (UET-UCT assumption). We can easily derive a BSP algorithm from an asynchronous algorithm, as follows:

Claim 1 *It is easy to obtain a feasible BSP schedule from an asynchronous algorithm designed for the delay model, just by replacing any (asynchronous) communication by a global one.*

2.4.1 A basic asynchronous algorithm

In this section, a simple algorithm for SIC under the standard asynchronous delay model is presented. The principle consists in filling the processors one after the other, from time zero to time t^*, that is from left to right on the figures (all details can be found in algorithm 1).

When a chain does not fit entirely on a processor (for instance, P_1), it is split and a communication is introduced between its two parts.

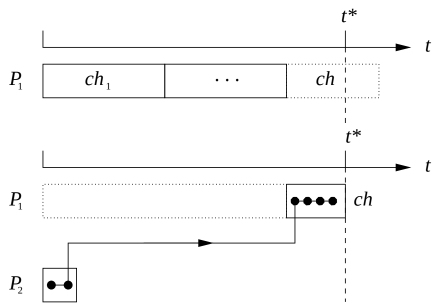

Figure 3: Greedy fill-in and allocation of a split chain.

Notation 2 *A chain that is executed in more than one processor will be denoted* split chain.

According to the precedence constraints: the first part starts on P_2 and the last part finishes on P_1, each part is allocated at extreme sides of the Gantt chart in order to respect the precedence constraints. Figure 3 illustrates the principle.

Algorithm 1 Asynchronous algorithm.

Require: $load(P_j)$ computes the number of tasks allocated to processor P_j
  $i \leftarrow 1$, $j \leftarrow 1$
  while $i \leq k$ **do** {it remains some chains to allocate}
    while $load(P_j) + n_i \leq t^*$ and $i \leq k$ **do**
      allocate ch_i to P_j
      $i \leftarrow i + 1$
    end while
    if $load(P_j) \neq t^*$ **and** $i \leq k$ **then**
      allocate the first $n_i - (t^* - load(P_j))$ tasks of ch_i to P_{j+1}
      allocate the last $t^* - load(P_j)$ tasks of ch_i to P_j
      $i \leftarrow i + 1$
    end if
    $j \leftarrow j + 1$
  end while

In the standard delay model, it is straightforward to show that the allocation given by this algorithm is valid, and the corresponding makespan is equal to t^*. The key point is that a communication can always be performed between the two parts of a split chain (because it has less than t^* tasks).

3 Two preliminary cases

In this section, some results for scheduling k independent chains under BSP are presented for two extreme cases, namely two processors, and an unlimited number of processors. Both cases have been already published in [11]. We only recall the main results which constitutes the basis of further developments.

3.1 SIC on two processors in BSP

First the problem of scheduling independent chains on two processors with a makespan equal to t^* is shown to be NP-hard. Then, an algorithm that builds a schedule with a makespan not greater than $t^* + C$ (in other words, it has no more than two supersteps) is presented and analyzed.

3.1.1 Theoretical analysis

Lemma 1 *Determining the existence of a schedule of UET independent chains on two identical processors within the time t^* is an NP-complete problem.*

Proof: In the presence of communications, any schedule will have a makespan greater than t^*, so we look for a schedule with only one superstep.

The problem is in NP, since a non-deterministic algorithm only needs to guess a subset CH of the set of chains, and to check in polynomial time that the sum of the chain lengths in CH is equal to t^*. Recall that the following *Partition* problem is NP-Complete [10]:

- Instance : a set X of elements of integer length n_i, an integer B such that $\sum n_i = B$.
- Question : does a subset of total length $\frac{B}{2}$ exist?

It is straightforward to reduce this problem into the SIC problem on two identical processors. Given a partition problem, consider the problem of scheduling k independent chains of length $n_i, 1 \leq i \leq k$, if $t^* \geq \sum_{i=2}^{k} n_i$, the problem is trivial. Otherwise, when $t^* = \frac{B}{2}$, a schedule with makespan t^* exists if and only if there exists a set of tasks whose total length is $\frac{B}{2}$.
❑

Moreover, a schedule without communication can be "far" from t^*. For instance, consider again the example of Section 2.1 consisting of three chains of the same length. The length of the schedule without communication is $\omega_o = \frac{4}{3}t^*$. Anyway, the well-known Graham's bound [14] $(2 - \frac{1}{m})t^*$ on list schedules remains always valid.

3.1.2 Algorithm for two processors

The algorithm given below determines a schedule for SIC on two processors. The principle is based on the general asynchronous algorithm of Section 2.4.1 using the systematic derivation stated in claim 1. The first processor is filled exactly until t^*. If a chain is shared between both processors, a CS is introduced between its two parts (for instance when the first part finishes).

Note that there is no need to sort the chains before the algorithm. The algorithm analysis is straightforward.

Proposition 1 *The time of the schedule generated by the previous algorithm is at most $t^* + C$.*

Proof: By construction, it is obvious to remark that at most one chain is split in the previous algorithm. Thus, there is at most one communication. As the largest chain has at most t^* tasks, a CS can be introduced between the two parts of the split chain, if it exists. So the previous algorithm gives a valid schedule with processing time less than $t^* + C$. ❑

3.1.3 Example

We detail an example for SIC with 4 chains of respective lengths $10, 10, 5$ and 5 on two processors. The time of a CS is $C = 2$ (cf. figure 4).

We first compute $t^* = 15$. Initially, the first processor is filled up to at least time t^*, which corresponds to two chains with a makespan of 20. Due to the addition of one CS, 5 tasks of the second chain are moved to the second processor. With one CS, the makespan becomes $17 = t^* + C$, and both processors are fully occupied.

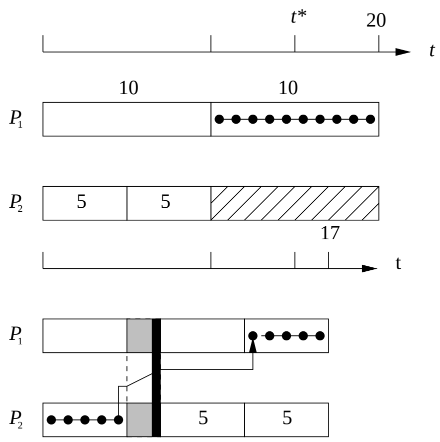

Figure 4: Allocation with only one superstep and BSP algorithm on two processors.

Remark: The proposed algorithm does not always generate an optimal schedule. In some cases, a schedule with makespan t^* does exist (see figure 5 with the same example as before). But, as proved before, the question of the existence of such a schedule is an NP-complete problem.

3.2 Unlimited number of processors

We can design an oblivious algorithm for this case. Given k chains, ch_i, for $1 \leq i \leq k$, each one is allocated to a separate processor. The makespan is given by the length of the longest chain, n_1. It is unnecessary to use more than k processors because each chain has to be executed sequentially.

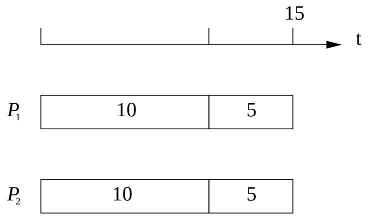

Figure 5: Example of a schedule that reaches t^* with no communication

There may be a solution with fewer processors, but it is not possible to improve the makespan. Finding the optimal schedule without communication with the minimum number of processor is also NP-Hard. It is equivalent to the problem of solving a $1D$-Bin-Packing problem [10], with bins of length n_1.

4 Fixed number of processors

In this Section, the extension to the case of a fixed number of processors is studied. We showed in the last section that SIC is already a difficult problem for two processors, but there exists an almost optimal algorithm which guarantees only an additional term of C from the optimal.

The complexity of the problem on $m > 2$ processors is first discussed (of course, $k > m$, otherwise the problem is straightforward). We study how to minimize the idle time t_I, we present a worst case analysis and a schedule in this case. The worst case analysis is developed to show that $\lceil \frac{m+1}{2} \rceil$ supersteps may be required for obtaining a schedule in which each processor has to execute at most t^* tasks. A schedule is proposed with processing time t^* and at most $\lceil \frac{m}{2} \rceil + 1$ supersteps. Finally, a family of algorithms where the number of communications is fixed and equally spaced in time is introduced.

In all these algorithms, a CS (which costs time C) can occur at any time. The influence of the latency term λ will be studied separately in section 5.

4.1 Perfect load balance

Since minimizing the parallel time t_m is equivalent to minimizing the overhead $t_I + t_c$, the prime and natural question is to study what happens when both terms are minimal. First, the problem of minimizing the idle time t_I without communication ($t_c = 0$) is shown to be difficult. Then, the problem of minimizing the idle time with communication is studied. We present an instance which requires at least $\lceil \frac{m+1}{2} \rceil$ supersteps for any schedule, and an algorithm which requires at most $\lceil \frac{m}{2} \rceil + 1$ supersteps, thereby achieving the lower bound for even m and exceeding it by only one for odd m.

4.1.1 Theoretical analysis

In this section, we are interested in SIC with the ideal makespan t^* on m processors ($m \geq 3$). Given a set of chains, it is possible that there is no schedule without communication which reaches the bound. But, even if such a schedule exists, it is difficult to find.

Proposition 2 *Finding the minimum makespan of the SIC problem without communication for an arbitrary number of processors is NP-hard in the strong sense.*

Proof: We use a reduction from the numerical 3-dimensional matching problem [10] which is recalled below.

N3DM problem:

- Instance : Given three sets, $X = \{x_1, \ldots, x_N\}$, $Y = \{y_1, \ldots, y_N\}$ and $Z = \{z_1, \ldots, z_N\}$. Each element $x \in X$ (respectively $y \in Y$ and $z \in Z$) has a positive integer weight $s(x)$ (respectively $s(y)$ and $s(z)$). Let B be a positive integer, such that $\sum_{i=1}^{N} s(x_i) + s(y_i) + s(z_i) = NB$.
- Question : Find N disjoint 3-partitions of $X \cup Y \cup Z$ such that in each partition $\{x_i, y_j, z_k\}$ the sum of the weights is equal to B.

This problem is known to be NP-hard in the strong sense [10]. In the presence of CS (i.e. more than one superstep), any schedule will have a makespan larger than t^*, so we look for a schedule without CS.

It is easy to show that SIC is in NP, since a non-deterministic algorithm needs only to guess an m-partition $(S_1, \ldots, S_m)$ of the set of chains, and to check in polynomial time that the sum of the chain lengths of each subset $S_j, 1 \leq j \leq m$ is at most t^*.

To do the reduction, we solve a N3DM problem by solving the schedule of $3N$ chains on N processors. Given an instance of N3DM, the length of each chain is defined as follows:

$n_{x_i} = 8B + s(x_i), 1 \leq i \leq N,$
$n_{y_i} = 4B + s(y_i), 1 \leq i \leq N,$
$n_{z_i} = 2B + s(z_i), 1 \leq i \leq N.$

The total number of tasks is $N(8B + 4B + 2B) + \sum_{i=1}^{N} s(x_i) + s(y_i) + s(z_i) = 15NB$.

The ideal makespan is $15B$. We will show that if there exists a schedule of length at most $15B$, then, it corresponds to a solution for the N3DM problem.

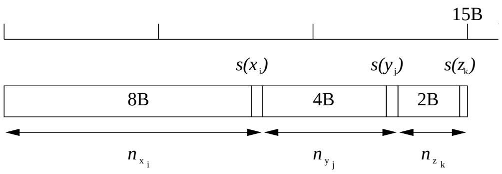

Figure 6: Principle of the allocation on one processor.

There is exactly one chain of each type (x, y and z) on each processor: each processor owns one chain of length n_{x_i} (it is not possible to allocate more than one such chain per processor, without mapping more than $15B$ tasks on a processor). Taking into account the chains of length n_{x_i}, the number of remaining available time slots is less than $7B$ (see figure 6). Thus, there is at most one chain of length n_{x_i} per processor. Obviously, the same argument holds also for the chains of type y and z.

Thus, each processor has exactly three chains, one of each type x_i, y_j, z_k, and $n_{x_i} + n_{y_j} + n_{z_k} = 15B$, that is $s(x_i) + s(y_j) + s(z_k) = B$. In any optimal schedule without communication, there is exactly one element from each set on each processor, and the sum of these elements is always equal to B. Thus, the solution of SIC is a solution for the N3DM problem. ❑

4.1.2 Worst case on the number of supersteps

We present an instance for which any schedule needs $\lceil \frac{m+1}{2} \rceil$ supersteps (or, equivalently $\lceil \frac{m-1}{2} \rceil$ CS) for SIC on m processors in order to minimize the idle time on the processors. The considered instance has $m+1$ chains of the same length. The number of supersteps which minimizes the idle time has been proven by constructing valid schedules with, respectively, 1, 2, ..., $\lceil \frac{m+1}{2} \rceil$ supersteps. The construction is quite technical, but not difficult, and more details can be found in [11].

4.1.3 Algorithm

We present now an algorithm which minimizes the idle time t_I, this algorithm can require up to $\lceil \frac{m}{2} \rceil + 1$ supersteps (that is $\lceil \frac{m}{2} \rceil$ CS, as we do not consider the CS of the last superstep). The algorithm is based on a transformation (described on claim 1) of the asynchronous algorithm (which has a perfect load balance) of Section 2.4.1. The key idea is to show that it is possible to group the communications placed by the asynchronous algorithms, at least two by two, into the same CS. To group asynchronous communications within the same CS, we introduce them after the last asynchronous communication sent and before the first asynchronous communication received.

The main steps of the analysis are sketched as follows. Again, the details of the proofs may be found in [11].

The chains are partitioned into three disjoint sets according to their length which will be scheduled one after the other:

- Set A of chains of length t^*,
- Set B of chains of length between $\lceil \frac{t^*}{2} \rceil$ and t^*,
- Set C of chains of length at most $\lceil \frac{t^*}{2} \rceil$.

Claim 2 *All the chains of set A can be scheduled without CS. All the chains of set C can be scheduled with a single CS at time $\lfloor \frac{t^*}{2} \rfloor$.*

It is more difficult to schedule chains from set B. The idea for reducing the number of supersteps is to gather some asynchronous communications into a single CS using the argument described in the following lemma. When the chains of set B are allocated in decreasing order of length we can state the following lemma:

Lemma 2 *The asynchronous communications of consecutive split chains in set B can be partitioned into subsets of size at least two (only the last subset can have one communication), in such a way that the communications on each subset use the same CS.*

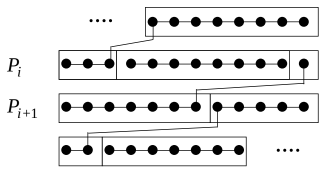

Figure 7: Allocation of chains on set B.

Sketch of the proof: The proof of the previous lemma is technical but not difficult. Two cases can occur (see figure 7), namely processors with two or three chains of set B. On processor P_{i+1} there are two different chains which can use the same CS (see figure 8). On processor P_i there are three different chains, in this case, there are again two sub-cases. Let ch_j be the chain completely allocated on P_i. Either chain ch_{j-1}, which starts its execution on P_i, can use the same CS as P_{i+1}. Or, it cannot share this CS, but can share another CS with chain ch_{j-2}.❑

Set A does not need a CS, if set C needs a CS, set B will need at most $\lceil \frac{m-2}{2} \rceil$ CS (as set C has at least a split chain), otherwise set B will need at most $\lceil \frac{m-1}{2} \rceil$ CS. So the total number of CS is bounded by $\lceil \frac{m}{2} \rceil$.

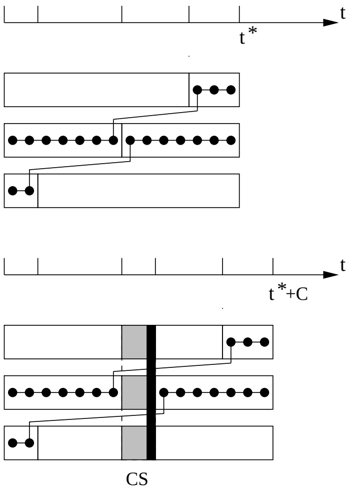

Figure 8: How to perform two asynchronous communications into a single CS.

Algorithm 2 given below groups the chains according to the receive time of the asynchronous communications. This algorithm minimizes the number of CS for a fixed chain allocation.

Algorithm 2 Algorithm with a limited number of supersteps

Apply the asynchronous algorithm and tag all communications
#cs ← 0
while there remains an untagged AC (asynchronous communications) **do**
  #cs ← #cs+1
  compute synch[#cs] as the smallest receive time of all the untagged AC
  tag all the AC that can be grouped within the same CS at time synch[#cs]
end while
for $i \leftarrow 1$ **to** #cs **do**
  introduce a CS after the processing of synch[i] tasks
end for

4.2 Fixed number of supersteps

With the previous algorithm, at most $\lceil \frac{m}{2} \rceil + 1$ supersteps are required to perfectly balance the load among the processors. In this section, instead of minimizing only the idle time t_I we are looking for a trade-off of the overhead $t_I + t_c$. Indeed, t_I may increase if t_c decreases.

Without communication, that is $t_c = 0$, there are several well-known approximation algorithms like LPT [15] and multi-fit [7]. Here we choose to present LPT as it is easy to understand. The performance guarantee of the makespan of LPT is: $w_{LPT} \leq (\frac{4}{3} - \frac{3}{m})w_o$. Recall that LPT (which stands for Largest Processing Time first) is the policy where the tasks are allocated in decreasing order of length, at the earliest possible time (see figure 9 for an example). Now, we are interested in approximation algorithms where a fixed number of CS may be considered.

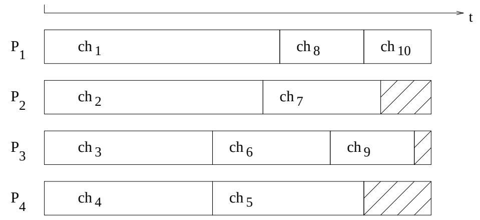

Figure 9: LPT principle of allocation on four processors.

Without loss of generality, let us suppose that the chains are strictly smaller than t^*, otherwise these chains are simply allocated one per processor, and the problem is solved with the remaining chains on the free processors. The goal of this section is to design an optimal algorithm for a fixed number of supersteps s that are evenly distributed ($s > 1$). To introduce this optimal algorithm, we present first an algorithm which depends on s and on an integer (denoted by α between n_1 and t^*) which is a parameter for tuning the distance between two consecutive supersteps. Then, we analyze the behavior of this first algorithm when the number of supersteps s varies. Finally, we present a refinement of this algorithm, and we choose the value of α for which it reaches the best performance.

4.2.1 A first algorithm for fixed supersteps

The algorithm 3 detailed below constructs a schedule for any set of chains within s supersteps. The principle is first to compute where the CS will be introduced, then to allocate the chains.

The $(s-1)$ CS are introduced from the beginning of the schedule in successive regular intervals of length $\Delta_\alpha = \frac{2}{2s-1}\alpha$. Let ϵ be the size of the last superstep. Note that by construction of the intervals $\epsilon \geq \Delta_\alpha$.

Algorithm 3 Given a number of supersteps s and an integer α

Require: $load(P_j)$ computes the number of tasks allocated to processor P_j
  $i \leftarrow 1, j \leftarrow 1, \Delta_\alpha = \frac{2}{2s-1}\alpha$, $\max_{load} = t^* + \frac{\Delta_\alpha}{2}$
  Introduce up to $(s-1)$ CS in regular intervals of length Δ_α
  while $(i \leq k)$ **do**
    if $(load(P_j) + n_i \leq \max_{load})$ **then**
      allocate ch_i to P_j
    else
      allocate first $n_i - load(P_j) + \max_{load}$ tasks of ch_i on P_{j+1}
      allocate the remaining tasks of ch_i to P_j
      if there is no CS between the split parts of ch_i **then**
        transfer tasks from P_j to P_{j+1} from left to right until a CS is found
      end if
      $j \leftarrow j+1$
    end if
    $i \leftarrow i+1$
  end while

The idea of this algorithm is to add some idle times (up to $\frac{\Delta_\alpha}{2}$ units by processor) to obtain a valid schedule. The chains are scheduled one after the other as in the asynchronous algorithm. A chain is split when a processor reaches the limit of $t^* + \frac{\Delta_\alpha}{2}$ tasks.

If a chain does not have a CS between its two split parts, it is delayed (as shown in figure 10). For the split chains smaller than $(s-1)\Delta_\alpha$, it is easy to verify that there always exists a CS between its two parts. For the other chains, if there is a split chain ch, it can be delayed in order to use one of the CS. In the case where a chain is delayed, the tasks on the first processor where the chain is allocated are transfered, until a CS is found, to the following processor. As ch has at most n_1 tasks, and n_1 is smaller than $(s-1)\Delta_\alpha + \frac{\Delta_\alpha}{2}$, this delay is at most $\frac{\Delta_\alpha}{2}$. Of course, this process is not cumulative since it does not affect the CS locations.

Proposition 3 *Given a SIC instance to be scheduled within s supersteps and an integer α such that $n_1 \leq \alpha \leq t^*$, the time of algorithm 3 is bounded by $t^* + \frac{1}{2s-1}\alpha + (s-1)C$.*

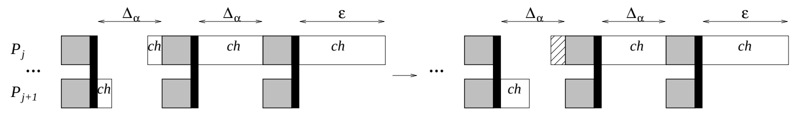

Figure 10: Allocating chains longer than $(s-1)\Delta_\alpha$. If there is no CS between the two parts of a split chain on an interval, the tasks belonging to this interval on P_j are transfered to P_{j+1} and the rest of the chain starts after CS.

Proof: Each processor, except maybe the last one which can have less than t^* tasks, have at least t^* tasks to process, and a chain that is split and allocated on two different processors uses one of the introduced CS in order to communicate. So, the schedule generated by algorithm 3 is feasible.

According to the allocation of chains in the algorithm, at most $(s-1)$ CS are done and as we start from an algorithm in time t^* by adding at most a time $\frac{\Delta_\alpha}{2} = \frac{1}{2s-1}\alpha$, we obtain the result. ❑

From this general result, we can derive some specialized cases depending on the length of the chains.

Corollary 1

1. *When the chains are small ($n_1 < \lceil \frac{t^*}{2} \rceil$), the makespan achieved by algorithm 3 is $t^* + C$.*
2. *Given $s_o = \max\{s | n_1 > \frac{s-1}{s} t^*\}$. For all s, $s > s_o$, the makespan achieved by algorithm 3 is $t^* + (s-1)C$.*

Proof:

1. The proof is straightforward by using a simplified version of the previous algorithm. One CS is introduced at time $\lceil \frac{t^*}{2} \rceil$. Then, the chains are allocated one after the others as in the asynchronous algorithm. As the chains are small, all the split chains can use this CS to communicate.
2. Introducing the CS in successive intervals of length $\frac{t^*}{s}$ (which corresponds to Δ_α for $\alpha = t^*$), the condition on s_o guarantees that the longest chain is smaller than $t^* - \frac{t^*}{s}$ (that is $(s-1)\Delta_\alpha$). So, if the chains are allocated as in proposition 3¿ There always exists a CS between the parts of a split chain. ❑

We study now, for an instance of SIC, the number of supersteps which leads to a schedule of minimal idle time, and then we study the number of supersteps which minimizes the overhead of the schedule produced by algorithm 3 (defined as the additive factor of the lower bound t^*: $\frac{\Delta_\alpha}{2} + (s-1)C$).

In Proposition 3, the overhead is given by the sum of two terms, one proportional to the number of CS, and the other inversely proportional to the number of CS. To find a good compromise between the load-balancing and the number of supersteps, we present the following result.

Proposition 4 *The optimal number of supersteps to minimize the overhead for the schedule produced by algorithm 3 is given by* $s^*_{\max} = \left\lceil \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rceil$ *or* $s^*_{\max} = \left\lfloor \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rfloor$.

Proof: The length of the schedule is t^* plus the sum of two positive expressions: namely, $\frac{\Delta_\alpha}{2}$ which is a decreasing function on s, and an increasing one, $(s-1)C$. So, the makespan is minimized when the derivative on s is equal to zero. However, as s is an integer, the minimum makespan is obtained by $s^*_{\max}$, equal to the floor, or the ceiling functions of $\sqrt{\frac{\alpha}{2C}} + \frac{1}{2}$. ❑

So, the best compromise algorithm is easy to derive:

Algorithm 4 Best compromise algorithm for a fixed number of supersteps.

Choose α such that $n_1 \leq \alpha \leq t^*$
Compute $s^*_{\max}$
Apply algorithm 3 with $s^*_{\max}$ supersteps

4.2.2 Refinement

Algorithm 4 clearly depends on the distribution of the chain lengths. As its performance is guaranteed by a worst case bound, it can be refined in some cases and the bound can be improved. First we propose some improvements on the average behavior of the algorithm, and then we discuss how to determine the value of α.

Algorithm improvement Algorithm 4 can be adapted in order to obtain a better average behavior. The chain placement can be slightly changed in order to fill-in the processors as close as possible to t^*, so the chain placement is done only until t^*. During the allocation, if a chain is split, and there is no CS between its two parts, then the less costly of the following procedures (depicted in figure 11) is chosen:

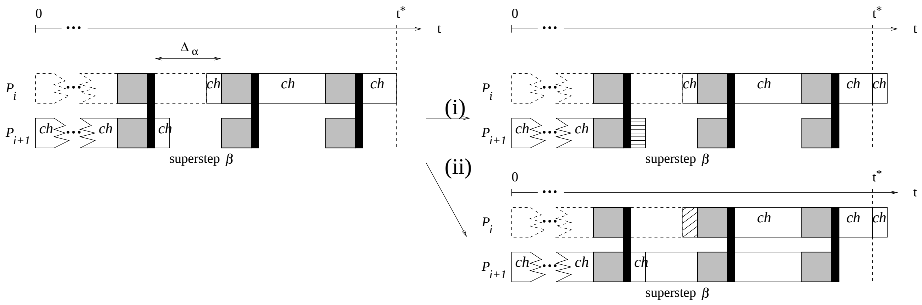

Figure 11: Alternatives to rearrange a chain with no extra CS between its two parts.

For the following description, for a split chain ch let p_i be the processor where the last part of ch is allocated, and p_{i+1} be the processor where the first part of ch is allocated. Let β be the superstep where the communication should occur.

(i) Stop executing the tasks on superstep β of processor p_{i+1}. Transfer these tasks to processor p_i. In this case the processor time previously allocated to ch on p_{i+1} (filled with horizontal lines on figure 11) can be used for the next chain allocation.

(ii) Delay the tasks executed by processor p_i during superstep β until the beginning of the next superstep. That is, the execution of chain ch on processor P_i will start after superstep β.

Obviously the delay introduced by this procedure can be as large as a half of the computational part of a superstep. So, this algorithm has the same worst case bound as for the original algorithm for a given number of CS. However, the improved algorithm has better average behavior.

With this improvement the number of supersteps can be reduced in the average case. To find the best makespan with the improvements we have to perform a search for the number of supersteps between 1 and $s^*_{\max}$.

Definition 2 *Given a SIC instance, we denote by s^* the number of supersteps that provide the best makespan using the improved algorithm.*

Bound improvement The idea is to take into account the differences between t^* and n_1, and between n_1 and the time slot of the last introduced CS.

Proposition 5 *Using the algorithm improvement, with a fixed number of supersteps s, the bound becomes:*

$$t^* + \max\left(\frac{\Delta_\alpha}{2} - \frac{t^* - n_1}{2}, n_1 - (s-1)\Delta_\alpha, 0\right) + (s-1)C.$$

Proof: In addition to $t^* + (s-1)C$, there is the maximum of two terms: the first term comes from the difference between t^* and n_1. This difference is a lower bound of the elapsed time between the two parts of a split chain. As $t^* - n_1$ increases, the time to add to t^* in order to schedule the chains in the worst case decreases. This time is proportional to $\Delta_\alpha - (t^* - n_1)$, but as we choose the less costly rearrangement, this time is divided by 2. This term increases with α.

The second term comes from the date of the last CS, as it is the last time a chain can be split. If the first part of a chain is greater than this time, it needs to be truncated in order to be scheduled. This term decreases with α. ❑

The minimum value of the bound is achieved for $\alpha_{\min} = \frac{t^*+n_1}{2}$.

5 Influence of the latency

The algorithms presented in the last section concentrated on the impact of CS. The specific structure of the SIC problem allows to limit the number of communications. We studied algorithms where CS are evenly distributed over the time. Practically, CS can occur at any time. The objective of this section is to study the impact on latency, i.e. the minimum delay between two successive CS.

5.1 Principle of an algorithm

The aim of this section is to show that it is possible to take into account the constraint of the latency with a small additional overhead proportional to λ. The main idea is to delay, or advance, close CS until they are spaced by a time interval greater than λ.

If a CS is delayed, an additional idle time may be introduced on the receiver side. The effect is to increase its completion time. If a CS is advanced, some idle times may be introduced at the sender side and the equivalent number of tasks can be migrated to the receiver side increasing the completion time (see figure 12).

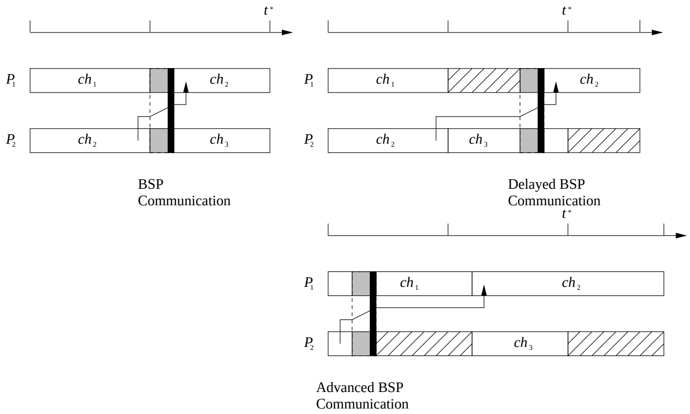

Figure 12: Moving CS forward or backward

In figure 12, on the initial scheduling, three chains are placed. Chain ch_2 starts on P_2 and finishes on P_1. If the CS is delayed, the end of ch_2 is delayed too. On the other hand, if the CS is advanced, the end of the first part of ch_2 must be moved to P_1 and increases its completion time. So, any change on a CS date will only increase the receivers completion time. The procedure is detailed on algorithm 5. This procedure is available for any BSP algorithm and transform it to a new (feasible) BSP algorithm with the latency constraint, but in general, we can not guarantee its performance.

5.2 Analysis

Given a SIC instance. let ω^* be the makespan of an algorithm that does not consider the latency, we state the following proposition.

Algorithm 5 Algorithm with CS spaced of at least λ

Apply any of the given BSP algorithms
for each CS (from the first one to the last one) **do**
  if CS is at less than $\frac{\lambda}{2}$ time units from the previous one **then**
    migrate the tasks of the split chains that use this CS
    merge it (so advanced it) with the previous one
  end if
  if CS is between $\frac{\lambda}{2}$ and λ time units from the previous one **then**
    delay it until it occurs at λ units from the previous one.
    merge with this CS all previous CS at less than $\frac{\lambda}{2}$ time units
    delay the tasks of the split chains that use these CS
  end if
end for

Proposition 6 *The additional cost from one of the proposed BSP algorithms, taking into account the latency as stated in algorithm 5 is at most an additive term equal to* $+\frac{\lambda}{2}$.

Proof: All the CS introduced by the algorithm are displaced (forward or backward) by at most $\frac{\lambda}{2}$ unit of times. Each processor increases its completion time only if the communication it receives is displaced. As each processor receives only one communication, each processor finishes at most $\frac{\lambda}{2}$ time units after ω^*. As each processor is the origin of only one communication, each merged CS still corresponds to a 1-relation. ❑

6 Average case analysis

In this paper we proposed some algorithms, and variations, to solve the SIC problem. We already established the theoretical performance guarantee. In this section, we are interested in the average behavior of the algorithms, towards simulations. We used the curve t^* which is a lower bound of the optimal schedule to serve as a reference. The following algorithms have been implemented:

- LPT - (Largest Processing Time first) algorithm. Recall that it requires no communication.
- *Asynchronous* - (Algorithm 2) where there are no restriction on the CS placement.
- Three variations of the Algorithm 3 with a given number s of CS.

  All the implemented algorithms use the refinements of Section 4.2.2. The interval between two CS is proportional respectively to the ideal makespan t^*, the length of the longest chain n_1 and the average of both. We will denote the smallest makespan version, with the optimal number of supersteps, of the previous variations as, respectively $C/s^*/t^*$, $C/s^*/n_1$ and $C/s^*/(t^* + n_1)/2$.

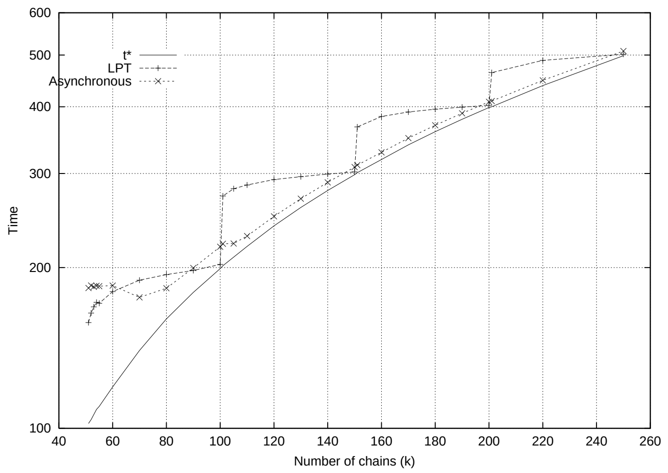

Figure 13: Average performance of LPT and *Asynchronous* on 50 processors with almost equal length chains.

6.1 Methodology

For each curve, the number of processors and the CS time (equal to C) are fixed. The chain lengths have been chosen according to a binomial law of average 100 and standard deviation 10. Only the chains of positive lengths are really scheduled. As the behavior of the algorithms does not depend significantly on the number of processors, we report only an average case of 50 processors. The CS cost on 50 processors is 10. This value corresponds to actual machines (for example T3E, in table 1 it corresponds to have tasks with 88 flop). Each plotted point is the average of 30 instances. As the measures obey a Gaussian distribution, the computed average is in a range of 5% of the real average.

LPT vs *Asynchronous*

The behaviors of LPT and *Asynchronous* are compared on chains with almost equal length (average length 100, and standard deviation 10) to the trivial lower bound of SIC. These cases should behave badly for both LPT, as the chains have almost the same length, and *Asynchronous*, as the CS cost is large (10 percent of a chain length).

In figure 13, as expected, the execution time of LPT does not smoothly get closer to the optimal. The makespan of *Asynchronous* tends more smoothly to the optimal. Nevertheless, when the number of chains is close to the number of processors, *Asynchronous* uses a large number of supersteps (cf. figure 14), so in these tests *Asynchronous* has a worse performance than LPT (which requires only one superstep). However, the maximum number (11) and the average number (7) of CS used are far from the worst case (for 50 processors, the worst case bound is 25 CS).

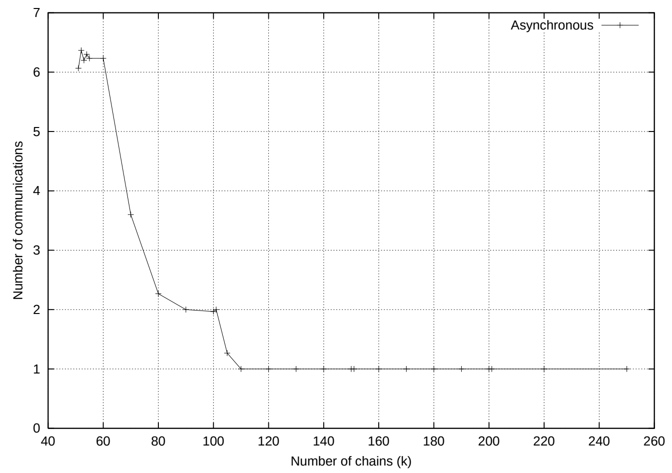

Figure 14: Average number of CS occurring on *Asynchronous*.

$C/s^*/t^*$ vs LPT

Let us compare the evenly distributed CS algorithm with LPT (cf. figure 15). The behavior of $C/s^*/t^*$ is close to the behavior of *Asynchronous*, whose makespan gets closer to t^* when the number of chains increases.

Variations of the fixed number of CS algorithms

$C/s^*/t^*$ behaves as *Asynchronous* (cf. figures 15 and 13 respectively) when the number of chains increases. The achieved makespan gets smoothly close to the best makespan we can achieve $t^* + C$ (two supersteps).

$C/s^*/n_1$ does not get as close to $t^* + C$ as $C/s^*/t^*$ when the number of chains increases (cf. figure 16). The last CS occurs at a date smaller than the chain size thus, contrary to $C/s^*/t^*$, some delays are always introduced. However, when the number of chains is small $C/s^*/n_1$ performs better than $C/s^*/t^*$, this is due to a smaller interval between two CS, thus to the introduction of less idle time.

$C/s^*/(t^* + n_1)/2$ combines the advantages of the two previous algorithms. It reaches the minimal bound of $t^* + C$ as the $C/s^*/t^*$ algorithm. When the number of chains is close to the number of processors, it is as good as $C/s^*/n_1$.

$C/s^*/(t^* + n_1)/2$ vs *Asynchronous*

Let us compare now $C/s^*/(t^* + n_1)/2$ with *Asynchronous* (cf. figure 17).

The former reaches t^*+C as fast as *Asynchronous*, but with a smoother behavior and a much better performance when the number of chains is close to the number of processors. The number of supersteps used by each algorithm is depicted by figure 18.

$C/s^*/(t^* + n_1)/2$ performs better than *Asynchronous* in almost all situations, but it uses less supersteps than *Asynchronous* even for a relative small number of chains. Conversely, when the communication time is

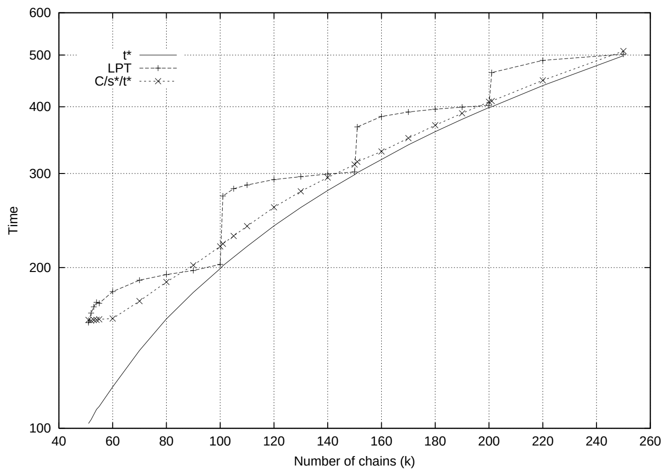

Figure 15: Average performance of LPT and $C/s^*/t^*$ on 50 processors with chains of almost equal length.

small compared to the average chain size, *Asynchronous* has a better performance as the communication weight becomes negligible. An example where the CS corresponds to 1% of the average chain size can be found on figure 19, again the average chain size is 100 and the number of processors is 50.

Influence of the number of communications

In this last experiment, we are interested on the behavior of $C/s/(t^* + n_1)/2$ when the number of CS varies. We fixed the number of processors (equal to 50), and we generated several SIC instances with 65 chains (the chain average and the standard deviation are the same as in the previous examples). As we can see in figure 20, the average of the makespan, produces a unimodal convex function, when the number of CS varies.

It is interesting to remark that even when the number of chains is close to the number of processors, the optimal number of supersteps s^* is small (for example on the instance of figure 20 this number is only 2).

6.2 Summarize

Let us now summarize the main conclusions of the experiments.

- The *Asynchronous* algorithm is the simplest one, and was not especially designed for BSP. It performs well when the number of chains is reasonably large compared to the number of processors. When the communication cost is very low, *Asynchronous* behaves well and has the best performances among all the algorithms.

  Theoretically, solving one SIC instance can require up to $\frac{m}{2}$ supersteps for a perfect balance of the load on the processors. However, even for instances which have a behavior close to the worst case (we

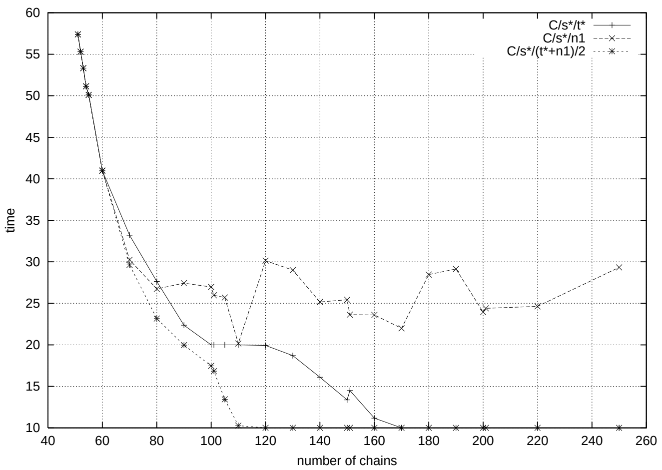

Figure 16: Average performance of the overhead for $C/s^*/t^*$, $C/s^*/n_1$, $C/s^*/(t^* + n_1)/2$ on 50 processors with chains of almost equal length.

saw that this corresponds to chains of about the same length), the number of communications used by *Asynchronous* is lower than this worst case.

- Let us now look at the algorithms derived from *Algorithm 3* which uses evenly distributed communications. $C/s^*/n_1$ has the best theoretical bound, but it does not have the best behavior when the number of chains increases. On the other hand $C/s^*/t^*$ has a good behavior when the number of chains increases but does not have a good performance ratio when the number of chains is close to the number of processors.

  Choosing the compromise $\alpha = \frac{t^*+n_1}{2}$, will guarantee the best behavior of this class of algorithms. $C/s^*/(t^* + n_1)/2$ performs better than *Asynchronous*, even when the number of chains is close to the number of processors since the number of introduced supersteps is a function of the achieved improvement.

  Finally, $C/s^*/(t^* + n_1)/2$ does not use more than a few supersteps. It is well suited for all instances when the communication cost is not negligible.

7 Conclusion

In this paper, we studied the influence of the computational model BSP on a simple scheduling problem, namely SIC. We focused first on the synchronization criterion. We proved that finding an optimal solution with any supersteps number is still NP-hard, but near-optimal heuristics can be found. We showed that there exist some instances which may require up to $\lceil \frac{m+1}{2} \rceil$ supersteps to obtain a perfect load balance. We proposed a general algorithm that obtains a perfect load balance with

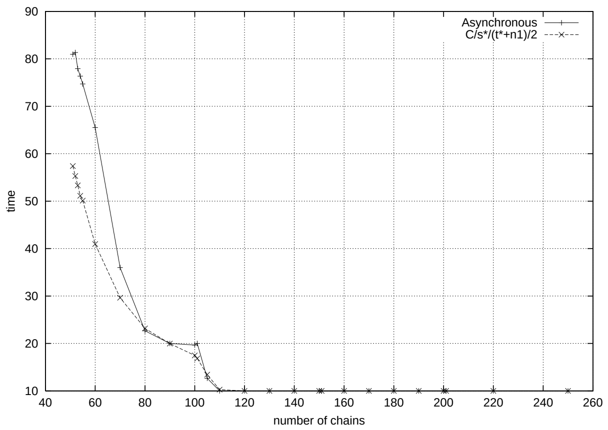

Figure 17: Average performance of the overhead (over t^*) for *Asynchronous* and $C/s^*/(t^*+n_1)/2$ on 50 processors.

only one more superstep ($\lceil \frac{m}{2} \rceil + 1$). We proposed also algorithms which achieve a trade-off on the overhead, that is both good load balance and small number of supersteps.

Some simulation experiments have been carried out to show that in practice the general algorithm needs only a few supersteps. The experiments showed also that the algorithm with the best worst case bound are not so good in average, especially for a large number of chains.

We hope that this work contributes to a better understanding on the design of BSP algorithms. A good BSP algorithm should achieve a trade-off between pure load-balancing and number of CS. For the SIC problem, the latency has no significant influence. This should be studied for other algorithms.

References

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Tenth Annual ACM Symposium on Parallel Algorithms And Architectures (SPAA 98)*, Puerto Vallarta, Mexico, June 28 - July 2 1998.

[2] R.J. Anderson, P. Beame, and W.L. Ruzzo. Low overhead parallel schedules for task graphs (extended abstract). In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990.

[3] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university bsp (pub) library - design, implementation and performance. In *Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, April 1999. url: `http://www.uni-paderborn.de/~pub`

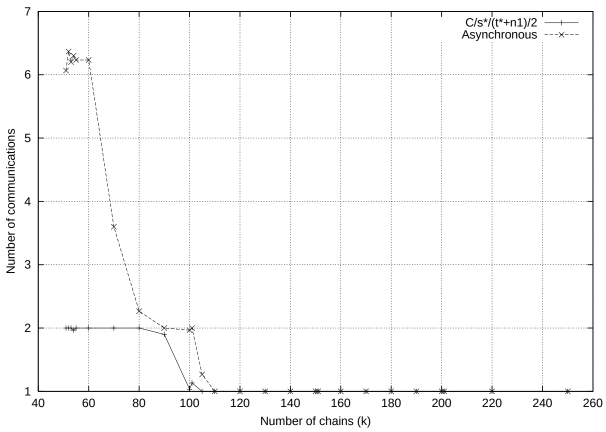

Figure 18: Average number of CS used by *Asynchronous* and $C/s^*/(t^*+n_1)/2$ on 50 processors.

[4] E. Caceres, F.K.H.A. Dehne, A.G. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In P. Degano and et al., editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[5] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In Luc Bougé et al., editors, *Euro-Par'96. Parallel Processing*, volume 2 of *Lecture Notes in Computer Science 1124*, pages 555–562. Springer-Verlag, 1996.

[6] R. Calinescu. A bsp approach to the scheduling of tightly-nested loops. In *Proc. 11th International Parallel Processing Symposium (IPPS'97), 1-5 April 1997, Geneva, Switzerland*, pages 549–553. IEEE Computer Society Press, 1997.

[7] E.G. Coffman, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[9] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An update notice on the extended BLAS. *ACM SIGNUM Newsletter*, 21(4):2–4, 1986.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, San Francisco, 1979.

[11] A. Goldman, G. Mounié, and D. Trystram. Near optimal algorithms for scheduling independent chains in bsp. In *HiPC'98*, December 1998. The 5th Int'l Conf on High Performance Computing.

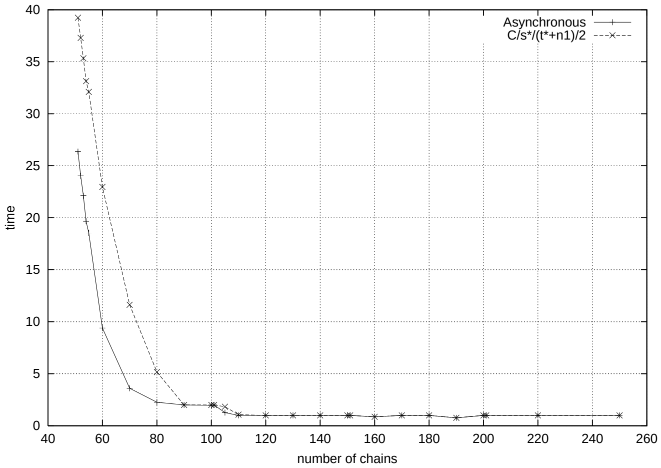

Figure 19: Average makespan of *Asynchronous* and $C/s^*/(t^* + n_1)/2$ on 50 processors with small communication times.

[12] A. Goldman, J. Peters, and D. Trystram. Exchange of messages of different sizes. In *Irregular 98*, pages 194–205, August 1998. LNCS 1457.

[13] T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the Association for Computing Machinery*, 25(1):92–101, January 1978.

[14] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[15] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[16] J. M. D. Hill, W. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[17] B.H.H. Juurlink, P. Kolman, F. Meyer auf der Heide, and I. Rieping. Optimal broadcast on parallel locality models. In *Proc. of 7th Colloquium on Structural Information and Communication Complexity (SIROCCO)*, L'Aquila, Italy, June 2000.

[18] I. Kort and D. Trystram. Some results on scheduling flat trees under LogP. *Information Systems and Operational Research INFOR*, 37(1):57–76, feb 1999.

[19] W. Kubiak, B. Penz, and D. Trystram. Scheduling chains on uniform processors with communication delays. Technical Report 3576, INRIA, France, December 1998.

[20] W.F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995.

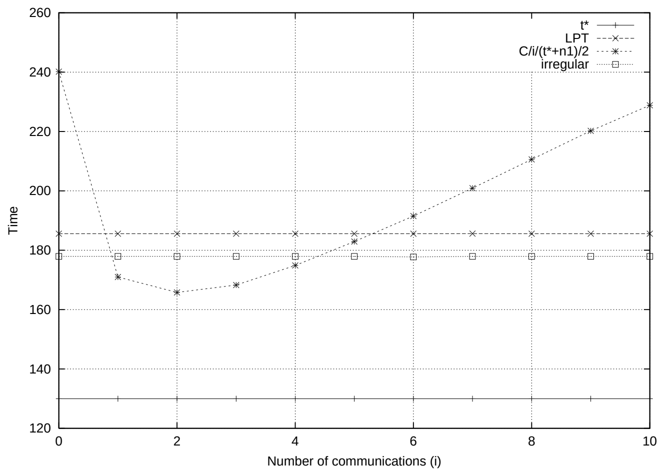

Figure 20: Influence of number of communications on $C/i/(t^* + n_1)/2$.

[21] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 1(18):55–71, January 1987.

[22] D. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[23] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196:109–130, 1998.

[24] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[25] J. Verriet. Scheduling tree-structured programs in the LogP model. Technical report UU-CS-1997-18, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, June 1997.

[26] W. Zimmermann, M. Middendoff, and W. Loewe. On optimal k-linear scheduling of tree-like task graphs for LogP-Machines. *Lecture Notes in Computer Science*, 1470:328–335, 1998.