Theoretical Informatics and Applications
Theoret. Informatics Appl. **36** (2002) 43–65
DOI: 10.1051/ita:2002004

PIPELINED DECOMPOSABLE BSP COMPUTERS *

Martin Beran[1]

Abstract. The class of weak parallel machines is interesting, because it contains some realistic parallel machine models, especially suitable for pipelined computations. We prove that a modification of the bulk synchronous parallel (BSP) machine model, called decomposable BSP (dBSP), belongs to the class of weak parallel machines if restricted properly. We will also correct some earlier results about pipelined parallel Turing machines.

Mathematics Subject Classification. 68Q05, 68Q10.

Introduction

The bulk synchronous parallel (BSP) model, introduced by Valiant [15], is an example of the so-called bridging models of parallel computers. A good bridging model should allow portable parallel algorithms to be developed easily and also implemented on the really existing computers efficiently. The BSP model fulfils these requirements – BSP algorithms have been designed and analyzed [6, 7, 10, 13], and BSP implementations exist [3, 9]. Hence, the BSP model is a realistic model of parallel computation. The decomposable BSP model (dBSP) is an extension of the BSP model [1]. It enables exploitation of communication locality of parallel algorithms in order to achieve an additional speedup.

Models of computation can be assigned into machine classes according to their computational power. The first machine class $\mathcal{C}_1$ [14] contains the Turing machine (TM) and other sequential models, *e.g.*, RAM (random access machine). Given an algorithm, its time complexity differs only up to a polynomial factor when

Keywords and phrases: BSP, complexity theory, models of compuation, parallel computing, pipelining.

* *This research was partially supported by the GA ČR grant No. 201/00/1489.*

[1] Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic; e-mail: `beran@ms.mff.cuni.cz`

© EDP Sciences 2002

executed on different models from the first class. The second machine class $\mathcal{C}_2$ [5] is the class of massively parallel computers, *e.g.*, PRAM (parallel random access machine). The time complexity of a problem on computers from the second class is polynomially related to the space complexity of the Turing machine. Machines from the second class offer an exponential speedup in comparison with the first class but, unfortunately, they are infeasible if physical laws of nature are taken into account. Hence we would like to have a class of parallel computers which compute faster then members of the first machine class, yet being realistic, *i.e.*, physically feasible. One such class is the class of weak parallel machines $\mathcal{C}_{\text{weak}}$, defined by Wiedermann [17]. Members of class $\mathcal{C}_{\text{weak}}$ provide fast pipelined computation, *i.e.*, fast solution of many instances of the same problem. A typical feature of weak parallel models is restricted communication. For example, in a single step of a parallel Turing machine, information can be transferred from a tape cell to its immediate neighbours only. In dBSP, any communication pattern is allowed, but local communication (limited to small clusters of processors) runs faster and is therefore preferred. This characteristic – communication locality – is exhibited by many really existing parallel computers. This observation supports the hypothesis that weak parallelism of $\mathcal{C}_{\text{weak}}$ members could be a good characterization of realistic parallel computers.

In this paper, we will answer the question whether dBSP computers belong to the class of weak parallel machines. In Section 1, we will define the class $\mathcal{C}_{\text{weak}}$ and present a representative weak parallel machine – the pipelined parallel Turing machine (PPTM). We will correct some claims from [17] about the relation of PPTMs and class $\mathcal{C}_{\text{weak}}$. Section 2 contains an analysis of the membership of the dBSP model in the class of weak parallel machines. Results presented in this paper are contained also in the author's doctoral thesis [2].

1. Weak parallel models

The class of weak parallel machines $\mathcal{C}_{\text{weak}}$ was defined in [17]. In addition to time and space, a new complexity measure – *period* – was introduced.

Definition 1.1. *Period* $P(n)$ of a computation over a sequence of inputs (instances of the same problem $\mathcal{P}$) of the same size n is the upper bound on time between beginnings of reading two subsequent inputs of the sequence or between ends of writing two subsequent outputs.

Note that the period depends only on the size of an individual input, not on the number of inputs in the sequence. The time complexity is defined like in non-pipelined case, *i.e.*, it is the time needed to process a single input and produce the corresponding output. In a pipelined computation, the time is the interval between start of reading the i-th input and end of writing the i-th output, for any i. Intuitively, each input instance should pass the same processing stages. This concept is formalized by the notion of uniform pipelined computation.

Definition 1.2. A pipelined machine is *uniform* iff it eventually starts cycling (repeating the same configuration) with period $P(n)$ on a sufficiently long sequence of identical inputs.

Now we define the class of efficient pipelined machines. The definition is similar to the definition of the second machine class. The respective machines are "weak parallel", because they provide fast processing of many instances (small period), but not of a single input (which would lead to small time).

Definition 1.3. The *class of weak parallel machines* $\mathcal{C}_{\text{weak}}$ contains machines with their periods $P(n)$ polynomially equivalent to the space complexity $S(n)$ of a DTM.

"Polynomial equivalence" means that there are some constants k, l such that $P(n) = O(S^k(n))$ and $S(n) = O(P^l(n))$. Since space complexity of all models in the first machine class is linearly equivalent (*i.e.*, polynomially equivalent with $k = l = 1$) the DTM in the above definition can by substituted by any first class machine, *e.g.*, a RAM.

In the paper [17], the parallel Turing machine (PTM) is defined, analyzed, and its pipelined version (PPTM) is shown to be a member of $\mathcal{C}_{\text{weak}}$. In this section, we briefly repeat important definitions and facts from that paper (in Sect. 1.1) and present some new results: two restrictions of the PPTM model with proofs of their membership in the class of weak parallel machines and a theorem saying that the original nonrestricted PPTM is too strong to belong into $\mathcal{C}_{\text{weak}}$ (in Sects. 1.2, 1.3, and 1.4). These results will be used later in Section 2 when dealing with pipelined versions dBSP.

1.1. Parallel Turing machines

A parallel Turing machine is a modification of the standard Turing machine [4]. It achieves parallelism by making copies of its finite control unit and head. Its important property (which restricts the degree of possible parallelism) is the fact that all heads operate on the same set of tapes.

Definition 1.4. A (non-pipelined) d-dimensional k-tape *Parallel Turing Machine* – (d, k)-PTM for short – is based on a nondeterministic sequential Turing machine (NTM) having k tapes of dimension d. The computation starts with only one processor (a set of k heads with the control unit realizing the transition relation is called the processor). If the NTM would do a nondeterministic choice of performing one of instructions $i_1, i_2, \ldots, i_b$, the PTM creates $b - 1$ new processors and each of b processors executes a different instruction from the alternatives. Then all the processors work independently, but they share the same set of tapes. The processors run synchronously, performing one step per time unit. It is forbidden for several processors to write simultaneously different symbols into the same tape cell.

Whenever any processor encounters a choice in the transition relation, it generates new copies of itself. One would suspect that there could emerge $2^{O(T(n))}$ processors during $T(n)$ steps, but it is not the case. Any processors in the same

state with heads at the same positions are indistinguishable and effectively act as a single processor. Assuming $S(n)$ cells occupied on each of k tapes and q possible states, at most $q.(S(n))^k = O(S^k(n))$ distinguishable processors can exist. The limited number of processors causes that the PTM does not belong to the second machine class. As for membership in the first class, $(1, 1)$-PTM $\in \mathcal{C}_1$. A multi-tape PTM and a sequential TM can simulate mutually each other, but the sequential TM can be simulated by a multi-tape PTM in sublinear space [17], thus (d, k)-PTM $\notin \mathcal{C}_1$ for $k > 1$.

Definition 1.5. A *pipelined parallel Turing machine* (d, k)-PPTM has a two-dimensional read-only input tape, and a one-dimensional write-only output tape. The i-th input word w_i is written from the beginning of the i-th row of the input tape. The machine prints its i-th output to the i-th cell of the output tape. The input is read in order $w_1, w_2, \dots$ and the output is printed in the same order. The number of steps made by the PPTM between reading the first symbol of w_i and printing the i-th output depends only on the length of w_i. The PPTM halts after printing the last output.

A PPTM $\mathcal{M}$ solves a decision problem $\mathcal{P}$ in *time* $T(n)$ iff for a sequence of instances of the same length n the machine $\mathcal{M}$ prints the i-th output after at most $T(n)$ steps after reading the first symbol of the i-th input.

A PPTM $\mathcal{M}$ works in *space* $S(n)$ iff any sequence of arbitrarily many inputs of the same length n is processed using at most $S(n)$ cells on working tapes.

A PPTM $\mathcal{M}$ solves $\mathcal{P}$ with *period* $P(n)$ iff it reads the first symbol of the input w_i after at most $P(n)$ steps after reading the first symbol of w_{i-1} and prints the i-th output at most $P(n)$ steps after printing the $(i-1)$-st output.

Typically, the input words of a PPTM are *instances* of the same problem $\mathcal{P}$. Hence, the PPTM is especially suitable to solve not just one instance of $\mathcal{P}$, but a (long) sequence of instances.

The *Pipelined Computation Thesis* (PCT) says that for a problem $\mathcal{P}$, there is a pipelined algorithm for $\mathcal{P}$ with period polynomially equivalent to the sequential space complexity of $\mathcal{P}$. The class $\mathcal{C}_{\text{weak}}$ consists of exactly those machines which satisfy the PCT. The uniform $(1, 1)$-PPTM satisfies half of the PCT, as is proved in the following lemma.

Lemma 1.6. *A Turing machine computing in time $T(n)$ and space $S(n)$ can be simulated by a $(1, 1)$-PPTM with period $P(n) = O(S(n))$, time $O(T(n))$ and space $O(T(n))$.*

Proof. Each instance, which is being computed, occupies $S(n)$ consecutive tape cells. There are $T(n)/S(n)$ such instances at any time, occupying $T(n)$ cells in total, thus the PPTM needs space $O(T(n))$. The pipelined machine simulates one step for each instance and moves the whole content of its working tape one cell to the right. This can be done in a constant number of steps with enough processors. After $S(n)$ steps, there is enough room for a new instance at the beginning of the working tape. At the same time, the oldest instance on the tape is finished and its output printed. Hence the period of computation is $O(S(n))$. After $T(n)/S(n)$

periods, $T(n)$ steps is performed on an instance and its processing is finished. The PPTM time complexity is $O(T(n))$, because each period comprises $S(n)$ steps. Arrangement of the instances on the working tape is drawn in Figure 1. □

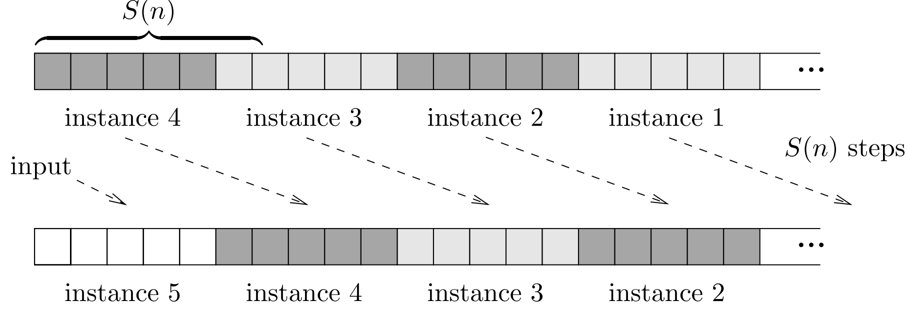

FIGURE 1. Pipelined PTM with period $P(n) = O(S(n))$.

As we will see later, the second part of the PCT – simulation of PPTM with period $P(n)$ in sequential space $O(P^k(n))$ – does not hold for a general uniform PPTM. Nevertheless, a single period can be simulated in small space.

Lemma 1.7. *Let $\mathcal{M}$ be a uniform $(1,1)$-PPTM computing with period $P(n)$. Then there exists a sequential Turing machine $\mathcal{M}'$ with a separate input tape computing in space $S(n) = O(P(n))$, which gets a configuration of $\mathcal{M}$ on its input tape and checks that $\mathcal{M}$ returns into the same configuration after $P(n)$ steps.*

Proof. Simulation machine $\mathcal{M}'$ uses the fact that only processors with heads at cells $i-t, \dots, i+t$ can influence the content of cell i after t steps. Figure 2 shows which cells have to be remembered and which may be forgotten to be able to simulate $P(n)$ steps. First, symbols in cells $0, \dots, P(n)$ are copied from input to the working tape, content of the cell 0 after $P(n)$ steps is computed and checked, then the symbol in the cell $P(n)+1$ is copied to the working tape and the cell 1 after $P(n)$ steps is evaluated, and so on until the whole working tape is processed. At any time, only a segment of $O(P(n))$ cells of the original PPTM $\mathcal{M}$ tape has to be stored on the working tape of $\mathcal{M}'$. Machine $\mathcal{M}'$ is a deterministic sequential Turing machine working in space $S(n) = O(P(n))$. See [17] for technical details of the simulation. □

The above algorithm allows to simulate a single period in polynomially related sequential space. This is not sufficient to claim that a whole computation can be simulated in polynomially related space, because the simulation does not cover the startup phase of the pipelined algorithm, before the cycle (required by uniformity) is entered. The cost of the startup phase of a uniform pipelined computation has no a priori upper bound and may contain a major part of the whole computation (*cf.* Th. 1.13).

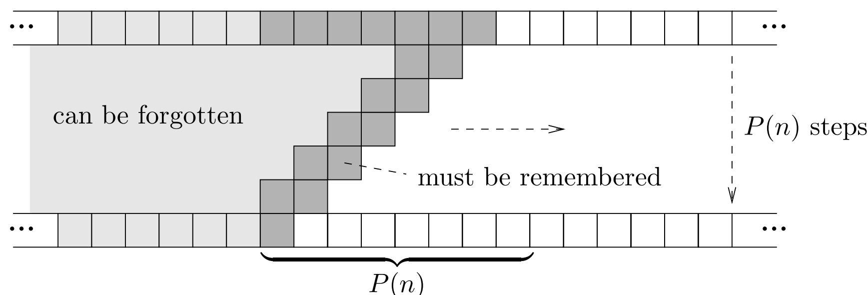

FIGURE 2. Sequential simulation of a pipelined PTM.

Now we present two restrictions of a uniform PPTM, namely the *restricted PPTM* and the *strictly pipelined PTM*. We prove that both models belong to the class of weak parallel machines.

1.2. LIMITED PPTM

The limited PPTM places an upper bound on the space complexity of the startup phase, *i.e.*, the initial part of a computation before the machine enters a cycle required by uniformity. The desired space upper bound is $O(P^k(n))$ for some constant $k > 0$. This allows, together with Lemma 1.7, to space-efficiently sequentially simulate the whole PPTM computation.

Definition 1.8. A uniform (d, k)-PPTM with period $P(n)$ is called a *limited* (d, k)*-PPTM* iff there exists a sequential Turing machine algorithm $\mathcal{A}$, working in space $S(n) = O(P^k(n))$ for some constant $k > 0$, which for each input generates configuration C belonging to the computation cycle of the PPTM (which the pipelined computation is required to enter due to uniformity). $\mathcal{A}$ gradually prints the cells of C from left to right (assuming the working tape starts in the left and stretches to the right) to its output tape, given the PPTM's input of size n.

Theorem 1.9. *Limited* $(1, 1)$*-PPTM is a member of the class of weak parallel machines.*

Proof. A simulation of a space $S(n)$ sequential computation on a PPTM with period $P(n) = O(S(n))$ is provided by Lemma 1.6. The algorithm is limited, because the PPTM tape contains configurations of a space bounded sequential machine. Therefore parts of the tape corresponding to individual instances can be generated by the sequential algorithm in space $O(S(n))$.

The reverse simulation uses the algorithm from Lemma 1.7 to check that the PPTM returns into the same configuration after a period. The special configuration C (required to exist by the definition of the limited PPTM) is tested, because it can be generated in space $S(n) = O(P^k(n))$. If a new cell is needed by the cycle testing algorithm, it is obtained by running the C generating algorithm until one new cell is printed. □

1.3. Strictly pipelined PTM

Another possible approach is not to restrict the complexity of the startup phase, but instead to prevent usage of its results during subsequent computation. More generally, we forbid sharing of information among instances. Thus, all necessary data must be computed for each instance separately. The definition of a *strictly pipelined parallel Turing machine* formalizes the intuitive notion of the instances not interacting with each other. For each instance, there are tape cells which are "owned" by the instance. Only contents of these cells and processors with heads on them may have an influence on the computation of that instance.

Definition 1.10. For a chosen input word w, a partitioning of tape cells of a $(1,1)$-PPTM into two sets is a partitioning into set $P_{w,t}$ of cells *pertinent* to w in step t and set $I_{w,t}$ of cells *independent* on w in step t, iff:

1. $P_{w,t} \cap I_{w,t} = \emptyset$;
2. $P_{w,t} \cup I_{w,t}$ contains the whole rewritten part of the working tape;
3. in the beginning of a computation ($t = 0$), there is only one processor, which is in the initial state and is scanning the cell number 0. At that moment, the cell $0 \in P_{w_1,0}$;
4. if the cell number $0 \in I_{w_i,t} \cap P_{w_i,t-1}$ then $0 \in P_{w_{i+1},t}$ and a new processor in the initial state scanning cell 0 is created;
5. a processor with the head on cell $c \in P_{w,t}$ in step t must not have its head on cell c' in step $t' > t$ if $c' \in I_{w,t'}$;
6. only a processor with the head on cell $c \in P_{w_i,t}$ can read a symbol from the i-th input word w_i in step t;
7. cell $c \in I_{w,t} \cap P_{w,t-1}$ contains the blank symbol λ in step t;
8. if a processor with its head on cell $c \in P_{w,t}$ creates new processor p then cell c' scanned by the head of p will belong to $P_{w,t+1}$. An exception is a processor created according to (4);
9. if a processor enters a terminal state, it disappears.

Definition 1.11. A uniform $(1,1)$-PPTM is *strictly pipelined*, iff in the beginning of a computation the working tape contains the blank symbol λ in every cell and the sets of cells pertinent to any pair of input words w_i and w_j, $i \neq j$, are disjoint at every step t.

The definition of the strictly pipelined PTM ensures that computations of individual instances − corresponding to individual input words − are completely separated from each other. Due to (3, 4), and (5), every processor is bound to a particular instance during all time of the processor's existence. A processor pertinent to an instance can neither become independent, nor pertinent to another instance. The feature (4) provides a means for starting computation of new instances. Conditions (7) and (8) guarantee that no instance can exploit information produced by another instance i, because information can be neither left in a cell which becomes independent on the instance i, nor passed outside the cells pertinent to i by a newly created processor. Passing information to the cells independent on i by an already existing processor is forbidden by (5). A processor pertinent to an

instance i cannot look at an input word other than w_i, due to (6). Feature (9) prevents processors pertinent to finished instances from becoming obstacles in further computation. The cells pertinent to an input word are not marked in any way. The notion of pertinent and independent cells is just an abstract characteristic of the PTM. Thus, there could be many ways of partitioning the cells, all satisfying the conditions of Definition 1.10. A parallel Turing machine is strictly pipelined if and only if there exists at least one such partitioning.

Theorem 1.12. *Strictly pipelined* $(1,1)$*-PPTM is a member of the class of weak parallel machines.*

Proof. The strictly pipelined PTM is uniform, therefore it starts cycling with period $P(n)$ after some time. During one cycle, a new input is read and computation of a new instance begins. Only a processor scanning cell 0 can start computation of a new instance. A processor pertinent to a problem instance (an input word), *i.e.*, a processor with its head on a cell pertinent to the instance, is either the initial processor created for the instance according to (4) in Definition 1.10, or it has been created by another processor pertinent to the same instance. Thus no information about other instances can be passed to the processor at hand *via* its state upon creation (which is determined by the state of the creator). Requirement (7) ensures that no information about an instance can be transferred to other instances via tape contents. Hence, processors pertinent to an instance have no information about other instances. In particular, there is no information about how many instances have already started processing before, therefore processing of any new instance must enter the uniformity cycle. Otherwise the cycling behavior would not be guaranteed. During one period, *i.e.*, $P(n)$ steps, all processors of the instance can allocate (made pertinent to it) up to $O(P(n))$ tape cells. As cycling is required, these cells must be made free (independent of this instance) and available for usage by the next instance in subsequent $O(P(n))$ steps. During this deallocation, only $O(P(n))$ new cells can be allocated. The instance cannot have more than $O(P(n))$ pertinent cells at any time and hence the computation of one instance can be simulated by a nonpipelined PTM in space $O(P(n))$. The individual instances share no information, thus each instance can be simulated separately, assuming there are no active processors and no non-blank cells other than those pertinent to the currently simulated instance. Since the nonpipelined $(1,1)$-PTM $\in \mathcal{C}_1$, space $S(n) = O(P^k(n))$ for some constant $k > 0$ suffices for a sequential Turing machine to simulate the computation.

The reverse simulation of a space $S(n)$ Turing machine on the pipelined PTM with period $P(n) = O(S(n))$, as described in Lemma 1.6, satisfies the restriction of the strictly pipelined PTM. □

1.4. Unrestricted PPTM

We can ask a natural question whether the restrictions of limited and strictly pipelined PPTMs are necessary or not. We prove that general or even uniform PPTMs are too powerful for being weak parallel machines (this fact disproves the

claim of [17] that uniform PPTMs are members of $\mathcal{C}_{\text{weak}}$). The problematic part is a space bounded sequential simulation of a PPTM. Although we are able to simulate a single period (and thus all periods due to uniformity) in small space using Lemma 1.7, it is also necessary to somehow obtain one configuration belonging to the cycle. If we just nondeterministically guess a configuration and test that it will repeat after every $P(n)$ steps, we are not sure whether the PPTM would ever reach such a configuration. We could guess a perfectly cycling configuration unreachable from the initial configuration of the PPTM. Therefore, we must check that the guessed (or otherwise obtained) cycling configuration would be ever reached by the simulated PPTM machine. Generally, the PPTM can get into the cycle after an arbitrarily complex precomputation, which cannot be simulated using only $O(P(n))$ space. The precomputation phase is a major obstacle, as is shown in the following theorem and its corollaries:

Theorem 1.13. *Every sequential Turing machine (TM) algorithm that halts on each input can be simulated by a uniform* $(1,1)$*-PPTM algorithm with period* $P(n) = O(n)$.

Proof. A PPTM computation is divided into two phases. In the precomputation phase, results of TM computations are computed for all inputs of length n. During the computation phase, a simple table lookup is performed for each input. The PPTM uses a two-track tape. The table is stored in one track and the inputs are put into the second track.

- Precomputation phase: Run the sequential TM algorithm for all $2^{O(n)}$ possible inputs and write a table of pairs $\langle i, o\rangle$ to the first track of the tape. For each input i occupying n tape cells there is a single cell o which holds the information about acceptance or rejection of this input. The table can be stored in space $n2^{O(n)} = 2^{O(n)}$. As the definition of the pipelined PTM requires that reading of a new input word is started after every $P(n)$ steps, the input is continuously read and stored on a separate track of the PTM's working tape.
- Computation phase: Create one head for each tape cell to be able to move information along the tape in parallel. The leftmost head reads one new input symbol in every step. Simultaneously, the contents of the second track is shifted one cell to the right. After $O(n)$ steps, one input word is read and stored at the tape so that it is aligned with the first entry of the lookup table created in the precomputation phase. Now one head compares – sequentially, in time $O(n)$ – the input and part i of the first table entry. If it matches, the corresponding o is attached to the input. Then a new period begins. When a new input is read, the input from the first period is shifted to the right, aligned with the second entry of the table, and compared. After $2^{O(n)}$ periods the input gets to the end of the table. As the table contains all the possible inputs, exactly one match must have occurred by that time. Therefore, the result for the input is known and can be printed to the output. At the same time, $2^{O(n)}$ inputs can be simultaneously compared to different entries of the table. After every $O(n)$ steps, one new input is read and one output is

produced. First, the inputs read and buffered on the working tape during the precomputation phase are processed. Simultaneously, the new inputs are read and put to the buffer in place of already processed ones.

□

Corollary 1.14. *There is a uniform* $(1,1)$*-PPTM computing with period* $P(n)$ *such that it cannot be simulated by a Turing machine in space* $O(P^k(n))$ *for a constant* $k > 0$.

Proof. Consider a problem with TM space complexity $S(n) = \omega(n^k)$ for all $k > 0$. It means that there is no TM which solves the problem in space less than $S(n)$. A TM algorithm for this problem can be simulated by a PPTM with period $P(n) = O(n)$ according to Theorem 1.13. A simulation of the PPTM on the TM in space $O(P^k(n)) = O(n^k)$ yields a contradiction to the assumption about $S(n)$. □

We showed that computations of a sequential algorithm on all inputs of fixed length can be performed in the precomputation phase and consequently a short period is achieved. Although it is possible to simulate one period in small sequential space, the complete simulation of the pipelined computation may need much more space. The mutual simulation between a sequential TM and a uniform $(1,1)$-PPTM with polynomially equivalent period and sequential space – which is required by the Parallel Computation Thesis – is possible from TM to PPTM, but not *vice versa.* Thus, we refuted the PCT for the uniform $(1,1)$-PPTM.

Corollary 1.15. *The uniform* $(1,1)$*-PPTM is not a model belonging to the class of weak parallel machines.*

Note that Corollary 1.15 is not in contradiction to Theorems 1.9 and 1.12, because the table lookup algorithm from Theorem 1.13 is neither limited nor strictly pipelined. The table of results needs space $2^{\Theta(n)}$ which exceeds the upper bound of a limited algorithm with polynomial period. Moreover, the table is reused by all instances which is forbidden by strict pipelining.

2. Pipelined decomposable BSP

Now we turn our attention from Turing machines to models based on RAM. We assume that the reader has some knowledge about RAM and PRAM models, their mutual simulations, and the related complexity theory (an overview can be found in [4, 11]).

2.1. Definitions

Throughout this paper, all RAM, PRAM, and BSP computers have logaritmic cost, *i.e.*, an instuction with n-bit operands takes n time units, and an n-bit value stored in register with r-bit address takes $n + r$ space units. Moreover, we assume that only a polynomial (in the size of input) number of PRAM or BSP processors can be actively computing from the beginning of a computation. An

inactive processor p starts computing only after explicitely activated by sending an activation message to it, or by calling a special instruction $activate(p)$ which costs $\log p$ time units.

A *Bulk Synchronous Parallel Computer* [13, 15] consists of p processors with local memories. Every processor is essentially a RAM with logarithmic cost. The processors can communicate by sending messages via a router (some communication and synchronization device). The computation runs in supersteps, *i.e.*, the processors work asynchronously, but are periodically synchronized by a barrier. A superstep consists of three phases: computation, communication, and synchronization. In the computation phase, the processors compute with locally held data. The communication phase consists of a realization of so-called h-relation, *i.e.*, processors send point-to-point messages to other processors so that no processor sends nor receives more than h bits. Data sent in one superstep are available at their destinations from the beginning of the next superstep. In the final phase of each superstep, all the processors perform a barrier synchronization.

Performance of the router is given by two parameters: g (the ratio of the time needed to send or receive one bit to the time of one elementary computational operation – the inverse of the communication throughput) and l (the communication latency and the synchronization overhead). Both g and l are non-decreasing functions of p. If a BSP computation consists of s supersteps and the i-th superstep is composed of w_i computational steps in every processor and of an h_i-relation, then the time complexity of the computation is defined by $T = \sum_{i=1}^{s}(w_i + h_i g + l) = W + Hg + sl$, where $W = \sum_{i=1}^{s} w_i$ and $H = \sum_{i=1}^{s} h_i$. We always assume the logarithmic cost of individual instructions. Communication contributes not only to H, but also to the computational cost W. Cost of a communication (send or receive) instruction which reads/writes k registers at addresses $a, a+1, \ldots, a+k-1$ is $\sum_{i=0}^{k-1}(\log(a+i) + |r_{a+i}|)$, where $|r_{a+i}|$ is size (number of bits) of value stored in register $a+i$.

A *Decomposable Bulk Synchronous Parallel Computer* with p processors and communication parameters g and l – denoted $dBSP(p, g, l)$ – is a $BSP(p, g, l)$ computer with some modifications and additional instructions `split` and `join`.

During a superstep, a processor can issue instruction `split(i)`, where $i \in \{0, \ldots, p-1\}$. If a processor calls `split`, then all other processors must call `split` exactly once in the same superstep. Beginning from the next superstep, the machine is partitioned into clusters (submachines) $C_1, \ldots, C_{cl}$, where cl is the number of different values of i in instructions `split(i)`. Processors which specified the same i in the split instruction belong to the same cluster, while different values of i imply different clusters. Communication is restricted to processors in the same cluster. Sending a message from a processor in one cluster to a processor in another cluster is forbidden. Cluster C_i can be further recursively decomposed using instruction `split(j)`, which assigns the calling processor to subcluster $C_{i,j}$. When a cluster is recursively decomposed, all processors in the cluster must call `split`, but other clusters need not decompose at the same time (their processors need not call `split`). See Figure 3 for an example of dBSP partitioning.

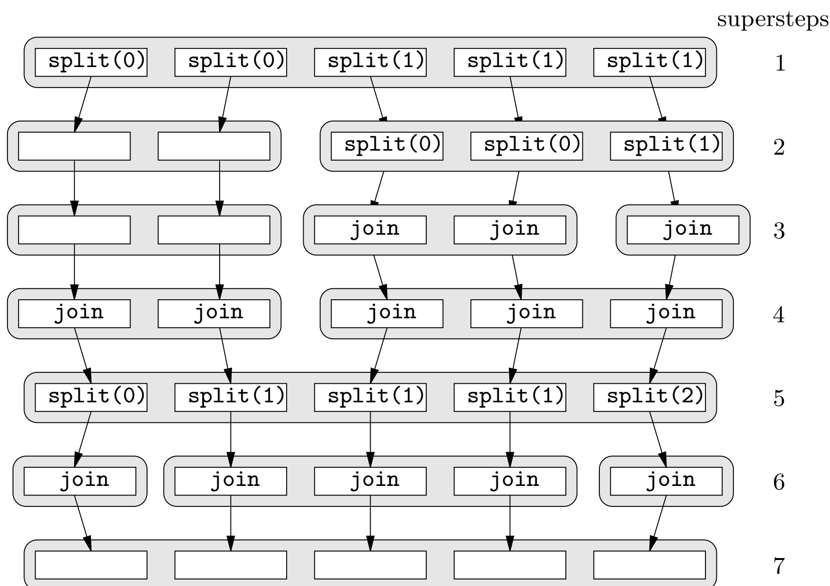

FIGURE 3. Example of dBSP instructions `split(i)` and `join`.

Instruction `join` called by a processor cancels the last level of decomposition which involved the calling processor. All the processors in all the sibling – originated from the same `split` operation – clusters must call `join` exactly once in the same superstep. Only one level of join is allowed in a single superstep. After a join, the machine can be decomposed again.

The *time complexity* of a dBSP computation is $T = \sum_{i=1}^{s}(w_i + h_i g(p_i) + l(p_i)) = W + \sum_{i=1}^{s}(h_i g(p_i) + l(p_i))$, where p_i is the size (number of processors) of the largest non-partitioned cluster existing in superstep s. The synchronization cost of a decomposed machine is $l(p_i)$ and not $l(p)$, because synchronization among processors from different clusters is not necessary. They could work independently and synchronize only at the time of a `join`, when they become a single cluster again.

A PPTM is constructed from a non-pipelined PTM by adding input and output tapes capable of holding many inputs and outputs. Similarly, input and output arrays of registers can be added to the PRAM and dBSP models yielding pipelined PRAM and pipelined dBSP models, respectively.

Definition 2.1. A *pipelined PRAM (pipe-PRAM)* is a PRAM equipped with additional two-dimensional array I of read-only registers and two-dimensional array O of write-only registers. The j-th value of the i-th input word w_i can be read from register $I_{i,j}$ and the j-th value of the i-th output is written to register $O_{i,j}$. Special register *io_select*, local to each processor, is used to choose the particular

input or output. The initial value of all registers *io_select* is 0 and the only operation permitted for *io_select* is `inc` to increment the value of *io_select* by one in the processor executing the instruction. The increment is performed in unit time regardless the value of *io_select*. The only way to access the input is by the instruction `r:=in(i)` which copies the content of $I_{io_select,i}$ into read-write PRAM register r. Only the number of bits of index i, the index of register r, and the value of $I_{io_select,i}$ contribute to the time cost of the `r:=in(i)` instruction. Particularly, the time cost of the instruction does not depend on the value of *io_select*. Similarly, the output is written by the instruction `out(i,r)` which copies the content of PRAM register r into output register $O_{io_select,i}$. The cost of the instruction depends again only on the number of bits in index i, the index and the value of r, rather than on the value of *io_select*. The input is read in order $w_1, w_2, \ldots$ and the output is written in the same order. Only the first $O(n^k)$, where $k > 0$ is a constant, processors may access the I/O registers.

Definition 2.2. A *pipelined dBSP computer (pipe-dBSP)* is a dBSP machine equipped with the same input/output register arrays as the pipe-PRAM.

Since individual instructions of a PRAM or a BSP computer can take different numbers of time units and the BSP computes – and thus can read input or write output – only during a part of each superstep, we slightly relax the definition of time and period of a pipelined computation (*cf.* Def. 1.1).

Definition 2.3. A computation of a pipelined machine on inputs of length n runs in *time* $T(n)$ and with *period* $P(n)$ iff the N-th output is printed after at most $T(n) + (N-1)P(n)$ time units since the beginning of the computation.

Before we answer the question about membership of the pipelined dBSP model in the class of weak parallel machines, we present several easy technical lemmas first.

Lemma 2.4. *On both a Turing machine and a RAM, time and space complexities are interrelated by the inequality*

$$S(n) \le T(n) \le 2^{O(S(n))} .$$

Proof. The first inequality follows from the fact that in $T(n)$ steps a Turing machine cannot use more than $T(n)$ cells and a RAM cannot allocate more than $T(n)$ bits in its registers. After $2^{O(S(n))}$ steps, all possible configurations in space $S(n)$ are exhausted and the machine has either to stop or to cycle forever. □

Lemma 2.5. *For the maximum number of processors* $p_{\max}$ *used in the first* T *time units of a PRAM computation on an input of size* n*, it holds*

$$p_{\max} \le 2^{O(\log n + T)} .$$

Proof. In the beginning of the computation there are up to $p_{\text{init}} \le O(n^k) \le 2^{O(\log n)}$ processors. The logarithmic time cost puts an upper bound on the value

of the argument given to the `activate` instruction, because this instruction must be finished in time T: $\log p_{\text{activate}} \le T$ yields $p_{\text{activate}} \le 2^{O(T)}$. □

Lemma 2.6. *Time and space complexities of a PRAM computer are related by the inequality*

$$S(n) \le pT(n) \le 2^{O(T(n))} .$$

Proof. During $T(n)$ time units, a single PRAM processor cannot allocate more than $T(n)$ bits in its local registers and in the global memory. See also the proof of Lemma 2.4.

Combination of the first part of this lemma with Lemma 2.5 yields $S(n) \le pT(n) \le 2^{O(T(n))}T(n) \le 2^{O(T(n)+\log T(n))} \le 2^{O(T(n))}$. □

2.2. Unrestricted pipe-PRAM and pipe-dBSP

Pipelined PRAM and dBSP computers, introduced in Definitions 2.1 and 2.2, are very powerful. There is a straightforward simulation of a space-bounded sequential computation in a polynomially related period.

Lemma 2.7. *Let $\mathcal{P}$ be a problem for which there is a sequential RAM algorithm running in time $T^{\text{RAM}}(n)$ and space $S^{\text{RAM}}(n)$. Let us have a pipelined dBSP$(p, g(p), l(p))$ computer with arbitrary parameter $g(p)$ and with $l(p) = O(p^k)$ for some constant $k > 0$. Then the pipelined version of $\mathcal{P}$ can be solved by the pipe-dBSP machine in time $T^{\text{dBSP}}(n) = O((T^{\text{RAM}}(n))^{k+1}S^{\text{RAM}}(n))$, space $S^{\text{dBSP}}(n) = O((T^{\text{RAM}}(n))^{k+2})$, with period $P^{\text{dBSP}}(n) = O((S^{\text{RAM}}(n))^2)$, and $p = T^{\text{RAM}}(n)/S^{\text{RAM}}(n)$ processors.*

Proof. Each processor stores q instances of $\mathcal{P}$ in its local memory. A superstep comprises computing the next $T^{\text{RAM}}(n)/p$ steps of the sequential algorithm for all qp unfinished instances. Then every processor sends its instances to the next processor. The oldest q instances are finished and their output written to the output registers. Simultaneously, q new inputs are read from the input registers.

Every processor uses $O(qS^{\text{RAM}}(n))$ registers. The maximum number of bits in an address is thus $\log(qS^{\text{RAM}}(n)) = \log q + \log S^{\text{RAM}}(n)$ instead of $\log S^{\text{RAM}}(n)$ in the RAM algorithm. Hence an instruction can be slowed down relatively to its RAM counterpart by factor $\log q$. Communication is done in two phases. The computer is twice repartitioned into pairs of processors. In the first phase, each processor with even *pid* sends its data to the successive odd processor. Data from odd to even processors are sent in the second phase.

From time needed by a single computational superstep and its related communication supersteps, *i.e.*, q periods, we obtain the length of the period.

$$\begin{aligned} qP^{\mathrm{dBSP}}(n) &= \frac{T^{\mathrm{RAM}}(n)}{p} q \log q + 2l(p) + S^{\mathrm{RAM}}(n) q \log q + \log p + \\ &\quad + qS^{\mathrm{RAM}}(n) g(2) + 2l(2) \;, \\ P^{\mathrm{dBSP}}(n) &= \frac{T^{\mathrm{RAM}}(n)}{p} \log q + 2\frac{l(p)}{q} + S^{\mathrm{RAM}}(n) \log q + \\ &\quad + \frac{\log p}{q} + S^{\mathrm{RAM}}(n) g(2) + 2\frac{l(2)}{q} \;. \end{aligned}$$

The first term corresponds to computation, the next one to partitioning, then two terms follow due to the cost of send/recv instructions. Communication and synchronization in clusters is covered by the last two terms. Now we use the assumption $l(p) = O(p^k)$. We also put $p = T^{\mathrm{RAM}}(n)/S^{\mathrm{RAM}}(n)$ and $q = (T^{\mathrm{RAM}}(n))^k$. This yields $P^{\mathrm{dBSP}}(n) = O(kS^{\mathrm{RAM}}(n) \log T^{\mathrm{RAM}}(n))$ and further using Lemma 2.4 we get $P^{\mathrm{dBSP}}(n) = O((S^{\mathrm{RAM}}(n))^2)$.

The first output is produced after the first instance travels through all processors, thus $T^{\mathrm{dBSP}}(n) = pqP(n) = O((T^{\mathrm{RAM}}(n))^{k+1} S^{\mathrm{RAM}}(n))$. Sum of all occupied registers over all processors, with addresses possibly increased by the factor of $\log q$ gives space $S^{\mathrm{dBSP}}(n) = pqS^{\mathrm{RAM}}(n) \log q = O((T^{\mathrm{RAM}}(n))^{k+1} S^{\mathrm{RAM}}(n)) = O((T^{\mathrm{RAM}}(n))^{k+2})$. □

It is possible to achieve a shorter period using the technique from Section 1.4. We show that results analogous to Theorem 1.13 and Corollary 1.15 can be proved.

Theorem 2.8. *Every sequential $T^{\mathrm{RAM}}(n)$-time and $S^{\mathrm{RAM}}(n)$-space bounded RAM algorithm that halts on each input can be simulated by a pipelined PRAM in time $T^{\mathrm{PRAM}}_{\mathrm{pipe}}(n) = 2^{O(n)} T^{\mathrm{RAM}}(n)$, space $S^{\mathrm{PRAM}}_{\mathrm{pipe}}(n) = 2^{O(n)} + O(S^{\mathrm{RAM}}(n))$, and period $P(n) = O(n^2)$, using only a single processor.*

Proof. The theorem uses the table lookup technique from the proof of Theorem 1.13. The table of results for all possible inputs of size n can be generated during a precomputation phase and stored using $2^{O(n)}$ registers with addresses of $\log(2^{O(n)}) = O(n)$ bits, yielding the total space occupied by the table to be $2^{O(n)} O(n) = 2^{O(n)}$. Additional space $O(S^{\mathrm{RAM}}(n))$ is needed as the workspace for the precomputation. The total time of the precomputation phase is $2^{O(n)} T^{\mathrm{RAM}}(n)$, *i.e.*, $2^{O(n)}$-times repeated sequential algorithm. One input consisting of $O(n)$ values of up to $O(n)$ bits is transformed to index i by multiplication of all the input values in time $O(n^2)$. The index is used to obtain the i-th element of the table containing the output value in time $O(n)$ because the index has $O(n)$ bits. □

Corollary 2.9. *The pipelined PRAM does not belong to the class of weak parallel machines.*

Proof. See also Corollaries 1.14 and 1.15. Consider problem $\mathcal{P}$ with RAM space complexity $S(n)$ such that $\forall k > 0 : S(n) = \omega(n^k)$. There is a pipe-PRAM

algorithm for $\mathcal{P}$ working with period $P(n) = O(n^2)$. Assuming pipe-PRAM $\in \mathcal{C}_{\text{weak}}$, we obtain a RAM algorithm for $\mathcal{P}$ with space complexity $O(n^k)$ for some constant $k > 0$. This is a contradiction to space complexity lower bound $S(n)$ for problem $\mathcal{P}$. □

Corollary 2.10. *The pipelined dBSP is not a member of the class of weak parallel machines.*

Proof. The claim follows immediately from the previous corollary, because a RAM (single processor PRAM) algorithm is essentially equivalent to a dBSP algorithm which uses only one processor. □

2.3. Limited pipe-PRAM and pipe-dBSP

Limited pipe-PRAM or pipe-dBSP do not allow an arbitrarily complex precomputation phase. They are restricted in the same way as limited PPTMs (*cf.* Def. 1.8), *i.e.*, there is a configuration in their computation cycle which is computable in small space.

Definition 2.11. A pipe-PRAM with period $P(n)$ is called a *limited pipe-PRAM* iff there exists a RAM algorithm $\mathcal{A}$, working in space $S(n) = O(P^k(n))$ for some constant $k > 0$, which for each input generates configuration C belonging to the computation cycle of the pipe-PRAM. $\mathcal{A}$ computes the contents of the i-th register of the j-th processor in configuration C, given a pipe-PRAM input of size n and numbers i, j.

Lemma 2.12. *Let there exist a sequential RAM algorithm for problem $\mathcal{P}$ running in time $T^{\text{RAM}}(n)$ and space $S^{\text{RAM}}(n)$. Then the pipelined version of $\mathcal{P}$ can be solved by a limited pipe-PRAM with period $P^{\text{PRAM}}(n) = O((S^{\text{RAM}}(n))^2)$.*

Proof. We run the algorithm from Lemma 2.7 and use one register of the global memory per processor for communication. The communication, synchronization, and partitioning time is avoided, but the cost of send/recv instruction may increase by the factor of $\log p$. Only one instance is handled by a processor in any time ($q = 1$).

$$P^{\text{PRAM}}(n) = \frac{T^{\text{RAM}}(n)}{p} + S^{\text{RAM}}(n) \log p + \log p \; .$$

Substitution of $p = T^{\text{RAM}}(n)/S^{\text{RAM}}(n)$ and $\log T^{\text{PRAM}}(n) = O(S^{\text{RAM}}(n))$ into the above formula produces $P^{\text{PRAM}}(n) = O((S^{\text{RAM}}(n))^2)$.

Clearly, the algorithm cycles with period $P(n)$. Every configuration consists of RAM configurations in different stages of processing and therefore can be generated by a sequential algorithm in space $S^{\text{RAM}}(n)$. Hence the simulation can be performed by a limited pipe-PRAM. □

Lemma 2.13. *A limited pipe-PRAM computing with period $P(n)$ can be simulated by a RAM in space $S(n) = O(P^k(n))$.*

Proof. The RAM machine generates configuration C which exists in the cycle by definition. Then it simulates one period and checks that pipe-PRAM returns to C. The largest value manipulated – and thus the highest addressable register and the number of processors – during the period is bounded by $2^{O(P(n))}$, see Lemmas 2.5 and 2.6. We assume that the simulated PRAM runs in SIMD mode, *i.e.*, all its processors perform always the same instruction. This is not a major restriction, because the SPMD mode (processors have the same program, but each has own program counter) can be simulated with only a constant-factor overhead. If the program has I instructions (note that I is a constant), a single SPMD step is realized by I SIMD substeps. It the i-th SIMD substep, processors with program counter equal to i perform the i-th instruction. Other processors do nothing.

Configuration C is gradually generated one register at a time, for each possible index of a processor and of a register. For every generated register value, the RAM checks whether the register will contain the same value after the single period. Checking runs recursively back in time from the last to the first step in the period. Sketch of the algorithm:

1. nondeterministically guess and write down the sequence of instructions executed ... space $S_1(n) = O(P(n))$; the SIMD mode ensures that the number of instructions to be guessed does not depend on the number of processors;
2. guess and write down the output ... space $S_2(n) = O(P(n))$;
3. let us denote `instr(t)` the instruction executed in step t. We define function `mem(t,p,m)` which returns the content of register (p, m) — local register `m` of processor p (or the global register m in if $p = -1$) in step t. The function handles all kinds of PRAM instructions.

function `mem(t,p,m)`
    { stop recursion }
    if $t = 0$ **then** obtain register (p, m) from C;
    { irrelevant instruction }
    if `instr(t)` does not change `(p,m)` **then** return `mem(t-1,p,m)`;
    { conditional assignment }
    if `instr(t)`=(`if z>0 then x:=y`) & `mem(t-1,p,z)`> 0 **then**
        return `mem(t-1,p,y)`;
    { unconditional assignment }
    if `instr(t)`=(`x:=y`) **then** return `mem(t-1,p,y)`;
    { binary operation }
    if `instr(t)`=(`z:=x`$\circ$`y`) **then**
        return `mem(t-1,p,x)`$\circ$`mem(t-1,p,y)`, where $\circ \in \{+, -, *, /\}$
end

The output guessed in step 2 is checked by calling `mem(`$P(n)$`,`o`,`-1`)` for each output register o. The recursion depth is $S_{3'}(n) = O(P(n))$ and we need to store up to $S_{3''}(n) = O(P(n))$ bits per recursion level. Thus we get $S_3(n) = S_{3'}(n).S_{3''}(n) = O((P(n))^2)$ for the space complexity of function `mem(t,p,m)`;

4. check the sequence of instructions guessed in step 1. Given the label (index in the program) of the instruction executed in the t-th step, check that the label of the next executed instruction was correctly guessed. Call of function `instr(`$P(n)$`)` does the checking.

```
function instr(t)
    if t = 0 then
        return 0;
    if instr(t-1) is not a jump then
        return instr(t-1)+1;
    ifinstr(t-1)=(goto l) then
        return l; { unconditional jump }
    if instr(t-1)=(if global_zero(z) goto l) &
    mem(t-1,-1,z)= 0 then
        return l { conditional jump }
    else
        return instr(t-1)+1;
end
```

Checking of the instructions needs a counter from 0 to $P(n)$ plus a space for function `mem(t,p,z)` ... $S_4(n) = O(\log P(n) + S_3(n)) = O(S_3(n))$.

The total space needed by the simulation algorithm is $S_{\text{nondet}}(n) = S_1 + S_2 + S_3 + S_4 = O((P(n))^2)$. Finally, we transform our nondeterministic algorithm into a deterministic one using Savitch's theorem. The space complexity of the deterministic algorithm is $S(n) = O((S_{\text{nondet}}(n))^2) = O((P(n))^4)$.

A call to $\texttt{mem}(0, i, j)$, stops the recursion and invokes the C generating algorithm to obtain the i-th register of j-th processor in time 0. The simulation needs space polynomial in the period of the PRAM algorithm plus space polynomial in $P(n)$ needed to generate C. The total space used by the RAM is thus $S(n) = O(P^k(n))$. □

Definition 2.14. A *pipelined dBSP* machine computing with period $P^{\text{dBSP}}(n)$ is *limited*, iff it can be simulated by a limited pipe-PRAM with period $P^{\text{PRAM}}(n) = O((P^{\text{dBSP}}(n))^j)$ for some constant $j > 0$.

We have defined the limited pipe-PRAM model first and derived the limited pipe-dBSP from it. The reason against a straightforward limited pipe-dBSP definition is a problem with uniformity. Recall (Def. 1.2) that a uniform machine eventually starts cycling with the cycle length $P(n)$. But any BSP-like machine can read input and write output only during the computational part of a superstep, not during communication and synchronization parts. Moreover, as we have seen in Lemma 2.7, simulating a sequential computation on a pipe-dBSP with a small period requires packing several periods into a single superstep. But then the BSP machine cannot be cycling with the cycle length $P(n)$, because the cycle length must be an integral multiple of the superstep length. According to our definition, the limited pipe-dBSP machine need not be uniform, we only require that it can be efficiently simulated by a limited pipe-PRAM (which is uniform).

Theorem 2.15. *Limited pipe-PRAM and limited pipe-dBSP with $l(p) = O(p^k)$ for some $k > 0$ are members of class $\mathcal{C}_{\text{weak}}$.*

Proof. Lemma 2.7 provides a pipelined dBSP algorithm for any problem with period polynomially equivalent to sequential space. A similar algorithm for the limited pipe-PRAM computer is given in Lemma 2.12. Its existence shows that also the dBSP computer is limited.

By definition, any limited pipe-dBSP machine can be simulated by a limited pipe-PRAM with period $P^{\text{PRAM}}(n) = O((P^{\text{dBSP}}(n))^j)$. This in turn is simulated by a RAM in space $S^{\text{RAM}}(n) = O((P^{\text{PRAM}}(n))^k) = O((P^{\text{dBSP}}(n))^{jk})$ according to Lemma 2.13. □

2.4. Strictly pipelined PRAM and dBSP

The limited pipe-dBSP allows for space-efficient sequential simulation, because the startup phase of its computation has limited space complexity. The strictly pipelined dBSP achieves the same goal by isolating individual instances (*cf.* strictly pipelined PTM in Sect. 1.3). The following rather technical definition ensures that computation of an instance cannot utilize any information produced by other instances. Moreover, the amount of work which can be spent in an instance during a single period is limited.

Definition 2.16. For a chosen input word w, a partitioning of PRAM registers into two sets is a partitioning into set $P_{w,t}$ of registers *pertinent* to w in step t and set $I_{q,t}$ of registers *independent* on w in step t, iff:

1. $P_{w,t} \cap I_{w,t} = \emptyset$;
2. $P_{w,t} \cup I_{w,t}$ contains all used local and global registers;
3. each register $r \in I_{w,t} \cap P_{w,t-1}$ contains value 0 in step t;
4. no instruction works with two registers r_1, r_2 such that $r_1 \in P_{w,t} \wedge r_2 \in I_{w,t}$;
5. if a processor works with registers from $P_{w,t}$ in step t and with registers from $I_{w,t}$ in step $t+1$, then its program counter contains 0 in step $t+1$, *i.e.*, the processor executes the same instruction as in the beginning of the computation.

Definition 2.17. A *pipelined PRAM* with period $P(n)$ is *strictly pipelined*, iff it is uniform, the sets of registers pertinent to any pair of input words w_i and w_j, $i \neq j$, are disjoint at every step t, and no more *work* (time multiplied by the number of processors) than $O(P^k(n))$, for some constant $k > 0$, is done on registers pertinent to a single input word during a period.

Lemma 2.18. *A strictly pipelined PRAM working with period $P(n)$ can be simulated by a RAM in space $S(n) = O(P^k(n))$.*

Proof. The strictly pipelined PRAM starts cycling with period $P(n)$, because it is uniform. A new input word is read and computation of the new instance begins during the cycle. A processor pertinent to an instance I, *i.e.*, a processor working with registers pertinent to I, can start working with another instance $I' \neq I$ only if it resets itself (sets 0, *i.e.*, the initial value, to its program counter), due to (5)

in Definition 2.16. Therefore, no processor can transfer information from I to I' *via* its internal state (program counter value). Requirement (3) ensures that no information about an instance can be transferred to other instances *via* values stored in registers. Hence, processors and registers pertinent to an instance have no information about other instances, in particular, they have no information about how many instances have already started processing before. Thus processing of a new instance must enter the cycle to ensure cyclic behavior. The computation may use only $2^{O(P(n))}$ processors and registers due to the maximum operand of an instruction possible in time $P(n)$. Only limited amount of work, namely $O(P^k(n))$, can be done on the instance in one period. Such work limits the number of processors pertinent to the instance to $O(P^k(n))$ and allows for allocation (making pertinent to this instance) of space up to $O(P^k(n))$ bits of memory. This memory must be made free (independent of the instance) for usage by the next instance during the next period. Hence, total memory consumed by a single instance is bounded by $O(P^k(n))$. The simulation algorithm takes the input and progressively executes all periods (cycles) with this input until the output is produced. As no information from the registers and processors pertinent to other instances can be utilized, only $O(P^k(n))$ registers and processors pertinent to the single instance being processed have to be stored from one cycle to the next. Other registers can be assumed to contain 0 and other processors can be assumed inactive (doing nothing). □

Definition 2.19. A *pipelined dBSP* machine computing with period $P^{\mathrm{dBSP}}(n)$ is *strictly pipelined*, iff it can be simulated by a strictly pipelined PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$ for some constant $j > 0$.

Theorem 2.20. *Strictly pipelined PRAM and strictly pipelined dBSP with* $l(p) = O(p^k)$*, for some* $k > 0$*, are members of class* $\mathcal{C}_{\mathrm{weak}}$.

Proof. The algorithm from Lemma 2.12 is a strictly pipelined PRAM algorithm. Consequently, the algorithm described in Lemma 2.7 is a strictly pipelined dBSP algorithm. Both algorithms run with period polynomially equivalent to the sequential space.

According to the definition, any strictly pipelined dBSP computer can be simulated by a strictly pipelined PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$. Lemma 2.18 then provides a RAM algorithm with space complexity $S^{\mathrm{RAM}}(n) = O((P^{\mathrm{PRAM}}(n))^k) = O((P^{\mathrm{dBSP}}(n))^{jk})$. □

3. Conclusion

Our goal was to analyze pipelined computations on the dBSP machine model. To do it, we have used a framework of weak parallel machines. We have presented a definition of class $\mathcal{C}_{\mathrm{weak}}$. Pipelined parallel Turing machines – defined and analyzed in [17] – are already known candidates for being weak parallel machines. Membership of PPTM in class $\mathcal{C}_{\mathrm{weak}}$ was claimed in paper [17]. The proof was based on ideas mentioned in Lemmas 1.6 and 1.7. In this paper, we identified a

problem in that proof: ability to simulate a single period in a space-efficient way does not yield simulation of the whole computation in small space. As we have shown in Corollary 1.15, a major part of a computation can be performed in its initial phase, before the machine starts cycling. Hence, a uniform PPTM is not a weak parallel machine, but a suitably restricted PPTM becomes a member of $\mathcal{C}_{\text{weak}}$. We have defined two possible restrictions – limited and strictly pipelined PPTMs.

The second type of machine models studied in this paper are pipelined decomposable BSP computers. The situation in this case is similar to PPTMs. A general dBSP machine is too powerful (even without any parallelism, *i.e.*, with only one processor used) and therefore is not a weak parallel machine. Two modifications of the pipelined dBSP model – limited and strictly pipelined dBSPs – have been introduced and their membership in $\mathcal{C}_{\text{weak}}$ proved. They are based on the same ideas as the limited and strictly pipelined PPTMs, respectively. This result indicates that dBSP is a viable realistic model of parallel computation. We have used dBSP and not BSP machines, because the algorithm for simulation of $S(n)$-bounded sequential computation in period $P(n)$ needs fast communication. A dBSP machines can be partitioned into clusters of size 2, thus the time of an h-relation is $hg(2) + l(p)$ instead of much larger BSP time $hg(p) + l(p)$. Note that $h = T^k(n)S(n)$ is very large in our case and thus the difference between using $g(2)$ and $g(p)$ is significant.

Negative results presented in this paper (Cors. 1.15, 2.9, and 2.10) pose a question whether the limited parallelism is a good notion at all. There are arguments for answering both "yes" and "no". Negative answer is motivated by the fact that class $\mathcal{C}_{\text{weak}}$ seems to be less robust than originally expected and necessary restrictions of limited and strictly pipelined machines are quite technical. A good machine class should contain some canonical members with simple and clear definitions. Examples are the Turing machine and RAM for $\mathcal{C}_1$ and PRAM for $\mathcal{C}_2$. We do not know any such model belonging to $\mathcal{C}_{\text{weak}}$ without further artificial restrictions. On the other hand, definition of the weak parallel machines was motivated by a need for characterization of realistic parallel models of computation. The trick with precomputing all possible outputs (proof of Th. 1.13) requires an exponential time and space for creating and storing the table of all results. But in real computers, we have typically only small (at most polynomial) amount of time, space, and processor resources. In such a realistic case, definition of a limited machine is a formalization of an intuitive requirement that a precomputation running exponentially longer and consuming an exponential amount of memory is infeasible. Note that in a polynomial space, only a small fraction of the whole solution table can be stored and there is no certainty that real inputs will make use of any such part of the table. Proof of Lemma 1.6 provides an algorithm for computing a series of inputs with period proportionally decreasing with the number of processors on realistic pipelined parallel computers (at least for $p \leq O(T(n)/S(n))$). Results for srictly pipelined machines show that this algorithm is usable even for cases where computations on individual inputs do not have anything in common and are totaly isolated from each other.

An important (but not surprising) observation is that if some data are computed once and then reused during a pipelined computation, then the total time needed to process all instances can be made significantly smaller. It could be interesting to study relations of pipelined computation and another recently emerging paradigm of computing – interactive and persistent machines [8, 12, 16]. Like a pipelined computer, an interactive machine processes a (potentially infinitely) long sequence of input data, produces corresponding output data, and is able to store information in its internal memory for later use.

References

[1] M. Beran, Decomposable bulk synchronous parallel computers, in *Proc. of SOFSEM '99.* Springer-Verlag, *Lecture Notes in Comput. Sci.* **1725** (1999) 349-359. `http://www.ms.mff.cuni.cz/~beran/publications.html`

[2] M. Beran, *Formalizing, analyzing, and extending the model of bulk synchronous parallel computer*, Technical Report V-829. Institute of Computer Science, Academy of Sciences of the Czech Republic (2000). `http://www.cs.cas.cz/research/library/reports_800.shtml`

[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 (1999). `http://www.uni-paderborn.de/~pub/`

[4] P. van Emde Boas, *Machine models and simulations*, edited by J. van Leeuwen. Elsevier Science Publishers, Amsterdam, *Handb. Theoret. Comput. Sci.* **A** (1990) 1-66.

[5] P. van Emde Boas, *The second machine class, model of parallelism*, edited by J. van Leeuwen, J.K. Lenstra and A.H.G. Rinnooy Kan. Centre for Mathematics and Computer Science, Amsterdam, *Parallel Computers and Computations, CWI Syllabus* **9** (1985) 133-161.

[6] A.V. Gerbessiotis and C.J. Siniolakis, *Primitive operations on the BSP model*, Technical Report PRG-TR-23-96. Oxford University Computing Laboratory, Oxford (1996).

[7] A.V. Gerbessiotis and L.G. Valiant, Direct bulk-synchronous parallel algorithms. *J. Parallel Distributed Comput.* **22** (1994) 251-267.

[8] D.Q. Goldin, S.A. Smolka and P. Wegner, *Turing machines, transition systems, and interaction* (submitted). `http://www.cs.umb.edu/~dqg/papers/mfcs.ps`

[9] J.M.D. Hill, W. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas and R. Bisseling, *BSPlib: The BSP programming library.* BSPlib reference manual with ANSI C examples (1998). `http://www.bsp-worldwide.org/implmnts/oxtool/`

[10] B.H.H. Juurlink and H.A.G. Wijshoff, Communication primitives for BSP computers. *Inform. Process. Lett.* **58** (1996) 303-310.

[11] R.M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, edited by J. van Leeuwen. Elsevier Science Publishers, Amsterdam, *Handb. Theoret. Comput. Sci.* **A** (1990) 869-941.

[12] J. van Leeuwen and J. Wiedermann, *The Turing machine paradigm in contemporary computing*, edited by B. Enquist and W. Schmidt. Springer-Verlag, *Mathematics Unlimited — 2001 and Beyond* (2001) 1139-1155.

[13] W.F. McColl, *Bulk synchronous parallel computing*, edited by J.R. Davy and P.M. Dew. Oxford University Press, *Abstract Machine Models for Highly Parallel Computers* (1995) 41-63.

[14] C. Slot and P. van Emde Boas, On tape versus core; an application of space efficient perfect hash function to the invariance of space, in *Proc. of STOC'84*. Washington D.C. (1984) 391-400.

[15] L.G. Valiant, A bridging model for parallel computation. *Comm. ACM* **33** (1990) 103-111.

[16] P. Wegner, *Models and paradigms of interaction*. OOPSLA Tutorial Notes (1995).

[17] J. Wiedermann, Weak parallel machines: A new class of physically feasible parallel machine models, in *Mathematical Foundations of Computer Science 1992, 17th Int. Symposium (MFCS'92)*, edited by I.M. Havel and V. Koubek. Springer-Verlag, Berlin, *Lecture Notes in Comput. Sci.* **629** (1992) 95-111.

Communicated by J. Hromkovic.
Received May 30, 2001. Accepted June 26, 2002.

To access this journal online:
www.edpsciences.org