An External-Memory Work-Depth Model and Its Applications to Massively Parallel Join Algorithms

Xiao Hu      Ke Yi      Paraschos Koutris
Hong Kong University of Science and Technology  (xhuam,yike)@cse.ust.hk  University of Wisconsin-Madison  paris@cs.wisc.edu

ABSTRACT
The PRAM is a fundamental model in parallel computing, but it is seldom directly used in parallel algorithm design. Instead, work-depth models have enjoyed much more popularity, as they relieve algorithm designers and programmers from worrying about how various tasks should be assigned to each of the processors. Meanwhile, they also make it easy to study the fundamental parallel complexity of the algorithm, namely work and depth, which are irrelevant to the number of processors available.

The massively parallel computation (MPC) model, which is a simplified version of the BSP model, has drawn a strong interest in recent years, due to the widespread popularity of many big data systems based on such a model. However, the current state of art for MPC algorithm design is still akin to that of PRAM, i.e., one has to specify, in each round, what each processor should compute and to which other processors messages should be sent. Being a coarse-grained parallel model, we observe that the MPC/BSP can be viewed as a combination of a certain variant of the external memory (EM) model and the PRAM. Seeing the impact of work-depth models on parallel algorithm design, we propose an EM work-depth (EWD) model, by incorporating elements from the EM model into an internal memory work-depth model (we choose the multi-thread model). We show that algorithms designed in the EWD model can be optimally simulated in the MPC/BSP model, and illustrate how it can be used to more easily design MPC/BSP algorithms by parallelizing the corresponding EM algorithms. In particular, we show how many of the EM join algorithms can be converted fairly easily to the EWD model, which allows us to either recover or improve the best known MPC/BSP results.

KEYWORDS
External memory model, BSP model, work-depth model

1 INTRODUCTION
The PRAM has played a fundamental role in the history of parallel computing. Nevertheless, it is seldomly directly used for parallel algorithm design. Instead, work-depth models such as circuits [44], language models [11], and multi-thread models [15], have enjoyed much more popularity. There are a number of reasons why a work-depth model is preferred over the barebone PRAM. First, in a work-depth model, the algorithm designer can focus on the algorithmic ideas instead of worrying about the details of how exactly tasks are assigned to each of the processors. Second, a work-depth model measures the complexity of an algorithm by work $W$ and depth $d$, which are irrelevant to $p$, the number of processors available. This allows us to better focus on the inherent parallel complexity of the algorithm. On the other hand, given a particular $p$, an algorithm with work $W$ and depth $d$ can be optimally simulated on the PRAM in $O(d + W/p)$ steps [11–13]. Thirdly, work-depth models are syntactically similar to many programming languages supporting parallel computing, so algorithms designed in these models can be more easily implemented in practice.

The PRAM can be considered as a fine-grained parallel model, where in each step, each processor carries out one unit of work. However, today’s massively parallel systems, such as MapReduce and Spark, are better captured by a coarse-grained model. The first coarse-grained parallel model is the bulk synchronous parallel (BSP) computer proposed by Valiant in 1990 [42]. The BSP has $p$ processors connected by a communication network, each with some private memory for local computation. Initially, input data is partitioned on the $p$ processors arbitrarily. Subsequent computation proceeds in rounds. In each round, each processor first collects all messages sent to it from other processors in the previous round (if there is one), does some local computation, and then sends messages to other processors. The rounds are synchronous in the sense that all messages being exchanged have to arrive at their destinations before the next round can start.

The initial BSP model described in Valiant’s paper tried to measure a number of costs with many parameters. Subsequently, most theoretical studies on BSP algorithms have adopted a simpler model [9, 16, 18–21, 30], characterized by three parameters: the number of processors $p$, each processor’s local memory size $M$, and the number of rounds $r$. The total message size sent out and received by any processor in each round cannot exceed $M$. The main complexity problem in this simplified model is therefore the tradeoff among the three parameters. In this simplified model, the local memory contents do not persist across rounds; if a processor wants to retain some of the local data in memory, it can just send the data to itself.

Depending on whether broadcasts are allowed and the parameter ranges considered, the (simplified) BSP model can have a number of variants. The situation has been made muddier by the fact that some papers on this topic did not explicitly relate their models to the BSP. In the first part of this paper (Section 2), we try to clarify the relationships among the BSP variants. In particular, we point out that the massively parallel computation (MPC) model [7], which has received a lot of attention in the database theory community, is the same as the CREW-BSP. We also study the relationship between the BSP and other related models such as the PRAM, the external memory (EM) model [4], and a parallel EM model [5].

In Section 3, we define the multi-way join problem, and review its complexity results in these models. In examining the results, we observe a high level of resemblance in the complexity bounds of this problem in the EM model and the BSP model, as well as many
commonalities in the algorithmic techniques used in both models. Essentially, both are coarse-grained parallel models, although the EM is a sequential model while the BSP is a parallel one. In fact, it is known how to simulate a BSP algorithm using the EM model \cite{17,32}, by serializing the algorithm and using the external memory as a medium for message passing. However, this simulation result does not offer any new algorithmic results since most EM algorithms predate their BSP counterparts. A reverse simulation would be much more useful, which is unfortunately impossible according to the current belief that NC ⊊ P.

On the other hand, we observe that many multi-way join algorithms in the EM model, although described as sequential algorithms, in fact contain a high level of parallelism, which is not surprising each join result only depends on a constant number of tuples. But to formalize this intuition, we would need a principled approach to converting the algorithms to the BSP. The main proposal of this paper is thus an EM work-depth (EWD) model (Section 4), which incorporates elements from the EM model into a PRAM work-depth model (we choose the multi-thread model). We first demonstrate how existing EM join algorithms can be easily rewritten in the EWD model (Section 5), and then show that any EWD algorithm can be optimally simulated by the BSP (Section 6). This has let us either recover or improve the best known BSP results. We also note that the model and simulation results are not restricted to the join algorithms; as an extra example, we show how Strassen’s algorithm can be easily written in the EWD model and optimally simulated in the BSP model. Interestingly enough, our work implies that EM algorithms, which were traditionally designed to optimize disk I/O, can also be used in today’s RAM-only systems, as long as they contain enough parallelism.

Finally, we hope that the EWD would become a standard model of BSP algorithm design, taking the role work-depth models are playing for the PRAM. The EWD inherits all the nice features of a PRAM work-depth model mentioned at the beginning. Most importantly, an EWD algorithm is very similar to a traditional shared-memory multi-threading program. Algorithm designers would not have to worry about all the low-level details of the BSP, such as how to assign tasks to each of \( p \) processors and how messages are passed around, so that they can more easily focus on the inherent parallel complexity of the algorithm, namely, work and depth.

2 MODELS

In this section, we formally define all the models studied in this paper, which are classified along two dimensions (see Table 1). Then we study their relationships, as summarized by the diagram in Figure 1. The RAM model requires no further explanation. We describe the other models below. We will measure space and communication costs in terms of words. A word is an \( O(\log N) \)-bit integer, where \( N \) is the input size.

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine-grained</td>
<td>RAM</td>
<td>PRAM</td>
</tr>
<tr>
<td>Coarse-grained</td>
<td>EM</td>
<td>PEM, BSP</td>
</tr>
</tbody>
</table>

Table 1: Classification of computation models

2.1 PRAM

The most well known parallel model of computation is perhaps the parallel RAM (PRAM). It is a natural generalization of the standard RAM model to \( p \) processors. The PRAM features a shared memory, which all processors have concurrent access to. In every parallel step, each processor reads \( O(1) \) words from the shared memory, does some local computation, and writes out \( O(1) \) words. Depending on whether concurrent reads/writes are allowed, there are different versions of PRAM. In the case that concurrent writes are allowed, various conflict resolution policies are possible, resulting in even more variants. In this paper, we will not consider these variants that allow concurrent writes, but only consider the following two versions of the PRAM: in the concurrent-read exclusive-write (CREW) model, two or more processors can read the same memory word concurrently, while in the exclusive-read exclusive-write (EREW) model, no concurrent read or write to the same memory word is allowed.

The PRAM has a rich yet controversial history. It is often criticized for being unrealistic, for essentially two reasons. First, building a large shared memory architecture is extremely difficult, if not impossible. Second, the model is too “fine-grained”, with each processor only processing \( O(1) \) amount of data in each step. Nevertheless, simplicity and elegance of the model had stimulated extensive research. Although algorithms designed in the PRAM model may not be directly implemented, many ideas underlying these PRAM algorithms have proved useful in other more realistic parallel models.

2.2 EM and PEM

The external memory (EM) model \cite{4} is another well-studied model designed for handling massive amounts of data. In some sense, it addresses the “fine-grained” problem of RAM/PRAM. The model consists of an internal memory of size \( M \) and an unlimited external memory, which is partitioned into blocks of size \( B \). Each step of the algorithm reads one block of data from external memory into internal memory, does some computation with data currently in internal memory, and writes one block back to external memory. The complexity of the algorithm is measured by the number of such I/O steps. For reasons that will be clear later, in this paper we will consider a degenerated version of the EM model where \( M = \Theta(B^2) \), i.e., the internal memory can only store a constant number of blocks. We just point out that most existing EM algorithms still work in this degenerated model, except for a few that require the “tall cache” assumption \( M = \Omega(B^2) \). Thus, the degenerated EM model has only one parameter \( M \). When \( M = O(1) \), the EM model essentially degenerates into the RAM model\(^1\). So \( M \) is always assumed to be large, making it a coarse-grained model.

The EM model can be extended to a parallel version in a natural way. In the parallel EM (PEM) model \cite{5}, there are \( p \) processors, each having a private internal memory of size \( M \). The external memory is still divided into blocks of size \( B = \Theta(M) \) (we still consider a degenerated PEM model), and is shared among all processors.

\(^1\)The formal definition of the RAM specifies exactly the set of primitive operations that take one unit of time. Since the EM model does not consider CPU time, an EM with \( M = O(1) \) is equivalent to the RAM when any operation on \( O(1) \) words is considered to take \( O(1) \) CPU time.
Similar to the PRAM model, each processor, in each parallel I/O step, reads one block of data from external memory, does some local computation, and writes one block back. As with the PRAM, there can be different variants regarding whether concurrent access to the same block on the shared external memory is allowed. In this paper, we only consider the CREW version of the PEM.

To be consistent with the BSP models, we use \( r \) to denote the number of I/O steps used by an algorithm, in the EM model (parallel I/O steps in the PEM model).

### 2.3 BSP

The bulk-synchronous parallel (BSP) model [42] addresses both the issues of fine-grained computation and the unrealistic shared memory of the PRAM. In modern terminology, it has a “shared-nothing” architecture, where data sharing can only be done through communication. More precisely, the BSP model consists of \( p \) processors, each having \( M \) local memory. In each round, each processor receives up to \( M \) words of data from other processors, does some local computation, and then sends out up to \( M \) words of data to other processors, which will be received by those processors in the next round. This is actually the EREW version of the BSP, which was the one described in Valiant’s original paper [42]. However, it is also natural to consider a CREW version [7, 20], where outgoing communication is not bounded. More precisely, a processor can broadcast a message to multiple processors with the same cost as sending it to a single processor. Furthermore, some have considered a weak-CREW BSP [18, 20], in which the processors are labeled from 1 to \( p \), and the destinations of a broadcast must be a range of consecutive processors.

The original BSP model not only considered \( r \), the number of rounds, as a measure of complexity, but also the local computation times of the processors in each round. By introducing more parameters, a “combined running time” is introduced as the final measure of complexity of the BSP model. This complicates the model, making it less clean than the EM/PRAM models, which could be one of the reasons why the BSP model had not received the popularity it should deserve.

With more experience with share-nothing architectures in the past decade, both in theory and practice, the community has generally agreed that the communication cost is more important than local computation time, and cleaner models have emerged. These models often did not identify themselves as the BSP model, but in fact they can be viewed as certain specializations of the BSP. In particular, the MapReduce model defined in [30] is a special case of the CREW BSP model with \( p = M = \Theta(N^{1-\epsilon}) \) for some small constant \( \epsilon > 0 \). The coarse grained multicomputer (CGM) model [18] is a special case of the weak-CREW BSP where \( M = N/p > p \). The massive parallel computation (MPC) model [7] is the same as the CREW BSP model, except that the parameter \( M \) is referred to as load.

There is an issue with regard to each processor’s local memory size, and whether its memory contents can persist across rounds. Indeed, Valiant’s original paper did not put a specific bound on the local memory size. Our definition of the BSP model above assumes that the memory size is the same as the communication bound, both of which are parameterized by \( M \). This implies that memory persistence is not necessary, since a processor can just send any data that it wants to persist to itself. We choose to do so because (1) it simplifies the model (otherwise we need one more parameter); (2) across \( r \) rounds, a processor receives at most \( rM \) data, so putting a local memory bound does not make any difference for constant \( r \), which is the most important case; and (3) none of the BSP algorithms that we are aware of needs local memory asymptotically larger than the incoming message size of each round.

Finally, in this paper we assume that \( M = \Omega(N^\epsilon) \) for any small constant \( \epsilon > 0 \). The same assumption has been adopted in most recent work [3, 25, 27]. We believe this is almost the minimum requirement for a model to be “coarse-grained”; some other papers use stronger requirements such as \( M = \Omega(\sqrt{N}) \) [8, 18] or even \( M = \Omega(N^{1-\epsilon}) \) [30, 34]. Smaller values for \( M \) would make the BSP close to the PRAM, losing its “coarse-grained” characteristic. This assumption is also well justified by practical values of \( N \) and \( M \), where the input size is on the order of terabytes while local memory size is in the gigabytes.

Figure 1: The relationships among parallel computation models for massive data. The notation \( x \uparrow y \) means that \( x \) blows up by a factor of \( O(y) \) in the simulation.
Meanwhile, we also need the obvious assumption that \( M = \Omega(N/p) \), so that the input data can at least fit in memory, all machines combined. Essentially, the BSP models a “memory-only” system, where the combined memory size is \( Mp \). Note that other computation models do not put any limit on the space usage of the algorithms. This issue will arise when we discuss the simulation relationships among these models.

### 2.4 Known relationships

Some simulation relationships among these models are known or trivial (see Figure 1). First, it is trivial to use the RAM to simulate the PRAM, and the EM to simulate the PEM, by simply serializing the algorithm. Of course, the number of steps increases by a factor of \( p \). Meanwhile, it is also known how to simulate BSP algorithms in the EM model \([17, 32]\), which is not surprising as it is also a parallel-to-sequential simulation. On the other hand, it is generally believed that an optimal reverse simulation, i.e., one that always reduces the running time by a factor of \( p \), is not possible. This is essentially the \( NC \subseteq P \) conjecture.

A similar situation happens between the fine-grained models and their coarse-grained counterparts. It is trivial to use the RAM to simulate an EM algorithm, where each I/O is simulated by \( B \) RAM steps. An optimal reverse simulation is believed to be not possible, either. More precisely, there are problems (e.g., shortest path) that are believed to be unsolvable in \( O\left(\frac{f(N)}{r}\right) \) I/Os where \( f(N) \) is the running time of the corresponding RAM algorithm.

It is also known \([21, 30]\) that any CREW PRAM algorithm using \( pM \) memory, \( pM \) processors, and running in \( r \) steps can be simulated in the CREW BSP model using \( O(p) \) processors, each with local memory \( O(M) \), \( O(r) \) rounds. The upper bound on the memory usage of the PRAM algorithm is necessary, since as mentioned the BSP is a model with limited memory. However, the number of rounds is sub-optimal. Similar to the RAM vs EM case, a coarse-grained model like the BSP is expected to use fewer steps to accomplish the same task. Take sorting as an example, which can be done on the PRAM in \( O(\log N) \) steps using \( N \) processors and \( O(N) \) space \([14]\). This simulation thus leads to a BSP algorithm using \( O(N/M) \) processors, each with \( O(M) \) memory, in \( r = O(\log N) \) rounds. However, the optimal BSP sorting algorithm requires only \( r = O(\log M N) \) rounds using the same number of processors \([20]\), which is \( O(1) \) in the parameter range \( M = \Omega(N^r) \) considered in this paper. Thus, \( r \) rounds of BSP can do a total of \( O(rpM \log M) \) comparisons. Equating this with \( N \log N \) gets us \( r = O(\log N/\log M) = O(1) \). On the other hand, the general PRAM- to-BSP simulation above uses one BSP processor to simulate \( M \) PRAM processors, which means that only \( O(pM) \) work can be done of work in each round, causing this \( O(\log M) = O(\log N) \) factor difference.

The gap can be more than logarithmic. Next consider the problem of enumerating\(^2\) the Cartesian product of two sets, each of size \( N \). A trivial optimal PRAM algorithm uses \( N^2 \) processors to solve this problem in \( O(1) \) rounds. The PRAM-to-BSP simulation above thus needs to use \( p = O(N^2/M) \) processors each with \( M \) memory. The BSP algorithm (see Section 3), on the other hand, can solve this problem in \( O(1) \) rounds with only \( p = O(N^2/M^2) \) processors each with \( M \) memory. This time, the \( O(M) \)-factor gap is caused by the fact that a BSP processor can produce \( O(M^2) \) results in a round, while the general PRAM-to-BSP simulation above still uses one BSP processor to do only \( O(M) \) amount of work in each round.

Fundamentally, the reason why a general optimal simulation from the PRAM to the BSP is impossible is that the BSP is a coarse-grained parallel model. While the processors are parallel, within a processor, some arbitrary (possibly highly sequential) computation can take place. Depending on the nature of the problem, this can lead to a gap that is logarithmic or polynomial in \( M \).

It is also known that the CREW BSP can be simulated by the EREW BSP with an \( O(\log M N) = O(1) \) factor increase in the number of rounds \([29]\). Thus, we will have the freedom of choosing whichever version of the BSP that is most convenient for our purpose. For example, we use CREW BSP for designing algorithms and simulating other models, while use EREW BSP if it is to be simulated by other models.

### 2.5 PEM vs. BSP

Next, we study the relationship between the (CREW) PEM and the BSP model. Although the processors in these two models use different forms of communication, we show that they are equivalent up to an \( O(\log p) \) factor. The intuition is that the shared external memory in the PEM model can be used for the point-to-point communication in the BSP model, and vice versa.

**Theorem 1.** Any algorithm in the EREW BSP model that runs in \( r \) rounds using \( p \) processors with local memory \( M \) can be simulated in the PEM model using \( p \) processors with local memory \( M \) in \( O(r \log p) \) I/O steps.

**Proof.** We show how each round of an EREW BSP algorithm can be simulated in \( O(\log p) \) I/O steps in the PEM model. Each PEM processor will simulate one BSP processor. At the beginning of the round, each PEM processor has the same memory contents as its BSP counterpart. The simulation consists of the following steps:

1. Each PEM processor \( i \) counts the number of words its corresponding BSP processor wants to send to processor \( j \), for \( j = 1, ..., p \). The PEM processor writes a list of \( (j, c_i(j)) \) pairs to consecutive blocks on disk, where \( c_i(j) > 0 \) is the number of words to be sent to processor \( j \). Note that there are at most \( M \) such nonzero counts, which can be written to disk in \( M/B = O(1) \) I/O steps.
2. Sort the \( (j, c_i(j)) \) pairs by \( j \) in \( O(\log p) \) parallel I/O steps \([5]\).
3. For each \( j \), compute \( c_j = \sum_i c_i(j) \). This can be done as a binary merge in \( O(\log p) \) parallel I/O steps: In each step, a PEM processor reads two consecutive blocks, adds up the counts \( c_i(j) \) for the same \( j \), and writes back the partial sums in one block. If the sum \( c_j \) is already fully computed for some \( j \), it does not have to be written back. So the block written back contains at most two \( (i, c_i(j)) \) pairs.
4. Each PEM processor writes all its outgoing messages to disk, as a list of \( (\sigma, t(\sigma)) \) pairs, which means that the corresponding BSP processor wants to send a word \( \sigma \) to processor

\(^2\)Formally, in an enumeration problem, the algorithm is required to call an emit function on each result, but the function call itself has no cost.
t(σ). Since we use the EREW BSP, the total message size out of any processor is at most M, which needs \( O(1) \) I/O steps to write to disk.

(5) For each \( j = 1, \ldots, p \), write \( M - e(j) \) pairs of \((null, j)\) to disk, so that each processor \( j \) has exactly \( M \) words addressed to it.

(6) Sort all the \( pM \) pairs of \((σ, t(σ))\) by \( t(σ) \) in \( O(\log(p)) \) I/O steps.

(7) Each PEM processor reads all the messages destined to it in \( O(1) \) I/O steps. Since each processor has exactly \( M \) words to be read, their locations on disk are predetermined.

\[ □ \]

**Remark 1.** The \( O(\log(p)) \) factor cannot be removed in general, because sorting can be done in \( O(1) \) rounds in the BSP model \([20]\), while there is a lower bound of \( O(\log(p)) \) in the PEM model when \( B = \Theta(M) \) \([5]\). Fundamentally, this is because a BSP processor can read/write \( M \) words whose locations can be arbitrary in the system, while a PEM processor must read/write \( M \) words stored consecutively on disk.

**Remark 2.** We have used large blocks \( B = \Theta(M) \) in the PEM model. For smaller values of \( B \), the simulation will need \( O(\frac{M}{p} r \log \frac{M}{p}) \) I/O steps, because reading/writing a large block of size \( M \) can be done in \( O(\frac{M}{p}) \) I/Os on smaller blocks, while sorting can be done in \( O(\frac{M}{p} \log \frac{p}{M}) \) I/Os \([5]\).

**Remark 3.** This simulation supersedes previous BSP-to-EM simulations \([17, 32]\), we first convert the BSP algorithm to a PEM algorithm, and then just serialize the PEM algorithm to an EM algorithm, which incurs an extra factor of \( p \). The resulting I/O cost of the EM algorithm is thus \( O(\frac{M}{p} r \log \frac{p}{M}) \), matching the previous results.

From the observation in Remark 1, it is actually easier to simulate the PEM by the BSP, though we do need a limit on the amount of space used by the PEM algorithm, since the BSP only has \( O(pM) \) total space.

**Theorem 2.** Any algorithm that runs in the PEM model in \( r \) I/O steps using \( p \) processors with local memory size \( M \), occupying \( O(pM) \) space of external memory can be simulated by an algorithm in the CREW BSP model in \( O(r) \) rounds using \( O(p) \) processors each having local memory \( M \).

**Proof.** We use \( p \) BSP processors to simulate the \( p \) PEM processors. We use \( O(p) \) additional BSP processors to keep all data stored in external memory, each mapping to \( M \) consecutive words. Each I/O step in the PEM model consists of 3 phases: reading phase, local computation, and writing phase. We simulate each phase as follows.

1. **Reading phase.** If PEM processor \( i \) reads a block from external memory that is mapped a BSP processor \( j \) (i.e., one of the \( O(p) \) additional BSP processors), we ask processor \( j \) to send the block of data to processor \( i \).

2. **Local computation performed by each PEM processor can be simulated by the corresponding BSP processor straightforwardly.**

3. **Writing phase.** If PEM processor \( i \) writes a block that is mapped to BSP processor \( j \), we ask processor \( i \) to send this block of data to processor \( j \).

After an I/O step, if a PEM processor wants to persist any local data, we simply ask the corresponding BSP processor to send them to itself.

\[ □ \]

### 3 MULTI-WAY JOINS

In this section, we define the main computational problem we study under the various models defined in the previous section, the multi-way join problem, which is a central problem in relational databases. Formally, a (natural) join query is defined as a hypergraph \( Q = (V, E) \), where \( V \) is a set of \( n \) attributes and \( E \subseteq 2^V \) is a set of \( m \) relations. Let \( \text{dom}(v) \) be the domain of attribute \( v \in V \). An instance of \( Q \) is \( R(Q) = \{ \text{dom}(e) : e \in E \} \), where \( R(e) \) is a set of tuples specifying a value in \( \text{dom}(v) \) for each attribute \( v \in e \). We often omit \( Q \) when the context is clear. We use \( N = \sum_{e \in E} |R(e)| \) to denote the size of \( R \). The join results of \( Q \) on \( R \) consist of all combinations of tuples, one from each \( R(e) \), such that they share common values on their common attributes. In this paper, we focus on the data complexity, i.e., we assume that the query size, namely \( n \) and \( m \), is constant. Indeed, the problem is intractable in terms of query size.

The trivial algorithm for computing the Cartesian product (i.e., the join problem with \( E = \{(v_1, \{v_2, v_3\})\} \) is the nested-loop join algorithm. Its EM version is known as blocked nested-loop join, which has I/O complexity \( \tilde{O} \left( \frac{N^2}{M^2} \right) \), or equivalently \( p = \tilde{O} \left( \frac{N}{M} \right)^2 \), where the logarithmic factor hidden in the \( \tilde{O} \) notation is due to randomized hashing. This algorithm has been derandomized and has its logarithmic factor removed \([25]\). In the BSP/MPC literature, one usually derives \( M \) as a function of \( p \), but for our purpose, it is more convenient to write \( p \) as a function of \( M \), since as we will see later, \( p \) will be equivalent to the I/O cost of an parallel EM algorithm.

For binary joins (i.e., \( E = \{(v_1, \{v_2, v_3\} \}) \), the classical algorithm is the sort-merge join. In the EM model, its I/O complexity is \( \tilde{O} \left( \frac{N}{M} + \frac{T}{M^2} \right) \), where \( T \) is the output size. Beame et al. \([8]\) designed a randomized BSP algorithm for binary joins with \( p = \tilde{O} \left( \frac{N}{M} + \frac{T}{M^2} \right) \), which has later been derandomized and has the logarithmic factor removed \([25]\).

The general multi-way join problem is much more challenging. In the RAM model, worst-case optimal algorithms have been recently discovered in a series of work \([6, 36, 37, 43]\) with running time \( \tilde{O}(N^{1+\rho^*}) \), where \( \rho^* \) is the fractional edge cover number of the hypergraph \( Q \). However, the current picture of the problem in the EM and the BSP model is far from complete, although the conjecture is \( \tilde{O} \left( \frac{N}{M} \rho^* \right) \) I/Os in the EM model, and \( p = \tilde{O} \left( \frac{N}{M} \rho^* \right) \) in the BSP model. This bound, if achievable, would be optimal in both models \([32]\). So far this bound has only been achieved for some special classes of joins.

The triangle join (i.e., \( E = \{(v_1, \{v_2, v_3\}) \}) \) was first considered in the EM model in \([24]\), which were later improved to the optimal I/O bound of \( \tilde{O} \left( \frac{N}{M} \right)^{3/2} \) by \([39]\). The result was
later generalized to LW-joins [23], for which \( \rho^* = \frac{N}{pT} \). No BSP algorithm is known\(^3\).

For Berge-acyclic joins, an EM algorithm with the desired I/O bound \( \tilde{O}\left(\frac{N}{M}\rho^*\right) \) was proposed in [26], but a BSP counterpart is currently unknown.

When the hypergraph \( Q \) is a graph (i.e., each relation has two attributes), a BSP algorithm with \( p = \tilde{O}\left(\frac{N}{M}\rho^*\right) \) has been recently proposed in [31]. This becomes the only case for which the BSP algorithm got ahead of its EM counterpart. Then by the BSP-to-EM simulation [32], an EM algorithm with the same I/O bound follows.

The aforementioned results have been summarized in Table 2, which shows a high level of agreement between the I/O complexity and \( p \), the number of BSP processors needed when each processor has local memory \( M \). When looking at the actual algorithms, although they look very different since they are designed for two different models, many of the algorithmic ideas are similar. Fundamentally, although the EM is a sequential model focusing on disk I/O while the BSP is a parallel model focusing on communication costs, the key common feature is that they are both coarse-grained models. In both models, an efficient algorithm should make best use of data locality. In the EM model, after fetching one block of tuples to memory, the algorithm should try to join them with as many tuples currently in memory as possible; in the BSP model, the algorithm tries to send as many joining tuples as possible to one machine, which can then emit their join results.

Therefore, an intriguing question is whether these two models are equivalent. However, this should not be true, since one is sequential and the other is parallel. To draw an equivalence, one has to inject parallelism into the EM model. One candidate is the PEM model reviewed above. However, this model has not received much attention in the literature with few algorithmic tools available. More importantly, it is not a work-depth model, thus not convenient for parallel algorithm design. Another technical issue is that there is still a logarithmic-factor gap between the PEM and the BSP as shown in Theorem 1, while we are most interested in constant-round BSP algorithms. In the next section, we describe our proposed model that addresses these issues.

### 4 AN EM WORK-DEPTH MODEL

Intuitively, our proposed EM work-depth model (EWD) is a combination of the multi-thread model and the (degenerated) EM model, but without the constraint that each I/O has to read consecutive words on disk. Before giving the formal definition, we first give an example showing an EWD algorithm that enumerates the Cartesian product of two relations (Algorithm 1).

\begin{algorithm}
\caption{PCartesian(\( R_1, R_2 \))}
1 Divide \( R_1 \) into chunks of size \( M/2 \);
2 Divide \( R_2 \) into chunks of size \( M/2 \);
3 for each chuck of \( R_1 \) do in parallel
4 \hspace{1em} for each chuck of \( R_2 \) do in parallel
5 \hspace{3em} Read the chunk of \( R_1 \) and the chunk of \( R_2 \) into memory;
6 \hspace{3em} Emit all pairs between the two chunks;
\end{algorithm}

We see that this EWD algorithm is very similar to the classical blocked nested-loop join, with the addition of a do in parallel construct that can execute any number of subprograms in parallel. The whole do in parallel construct completes when all the subprograms are finished. The work \( W \) (the total number of I/Os) and the depth \( d \) of this algorithm are simply \( W = O((N/M)^2), d = O(1) \).

Formally, a program in the EWD model is defined as follows. The external memory space is a word-array \( A \), which is shared among all subprograms. Each program \( P \) is associated with two constant-size parameters, \( \text{IN}(P) \) and \( \text{OUT}(P) \); \( \text{OUT}(P) \) is allowed to take the special value of null, but \( \text{IN}(P) \) cannot. Initially, the input data of size \( N \) is stored at \( A[1, N] = A[1, 2, \cdots, N] \), while all other entries of \( A \) are initialized as zero. The EWD program \( P \) can be constructed recursively in one of the following three ways.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Previous results in EM</th>
<th>Previous results in BSP</th>
<th>BSP results from EWD simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartesian product</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Binary join</td>
<td>( IO = \tilde{O}\left(\frac{N}{M} + \frac{T}{M}\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M} + \frac{T}{M}\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Triangle join</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Loomis-Whitney join</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Berge-acyclic join</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\rho^*\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\rho^*\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Graph joins</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\rho^*\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\rho^*\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Arbitrary joins</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\rho^* + \frac{\max\rho^*}{M} + \frac{T}{M}\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\rho^* + \frac{\max\rho^*}{M} + \frac{T}{M}\right) )</td>
<td>same</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>( IO = \tilde{O}\left(\frac{N}{M}\log_4^t\right) )</td>
<td>( p = \tilde{O}\left(\frac{N}{M}\log_4^t\right) )</td>
<td>same</td>
</tr>
</tbody>
</table>

Table 2: Previous EM algorithms, BSP algorithms and new simulation results (in \( O(1) \) rounds). \( N \) is the input size and \( T \) is the output size. The \( \tilde{O} \) notation hides factors polylogarithmic in \( N \), \( \rho^* \) is any fractional edge cover of the join query. \( \psi^* \) is the edge quasi-packing number of the join query. For arbitrary joins, the \( t \) in \( \max_t \) ranges over all the nodes in the given GHD.

\(^{3}\)A BSP algorithm with \( p = \tilde{O}\left(\frac{N}{M}\frac{1}{d^*}\right) \) has been proposed [32], but the author has acknowledged that there are some errors in their algorithm.
(1) Base case: After examining $\text{IN}(P)$, $P$ reads at most $M$ words from arbitrary locations on external memory, performs some computation on these words using $M$ memory, and then writes at most $M$ words to arbitrary locations on external memory. The locations of the words read can only depend on $\text{IN}(P)$; $P$ is not allowed to do adaptive reads, i.e., it reads some words first and then decide where to read next based on these words. For the base case, we define the work $W(P)$ as the number of words read/written, divided by $M$. Note that $W(P)$ can be smaller than 1. The base case has depth $d(P) = 1$. $\text{OUT}(P)$ can be any word written to external memory by $P$. If $P$ does not write anything, set $\text{OUT}(P) = \text{null}$.

(2) Sequential execution: $P$ runs $P_1$ then $P_2$ sequentially, where $P_1$ and $P_2$ are both EWD programs. That is, $P_2$ starts only after all the I/O operations by $P_1$, which may contain parallel subprograms, have finished. In this case, define $W(P) = W(P_1) + W(P_2)$, $d(P) = d(P_1) + d(P_2)$, $\text{IN}(P) = \text{IN}(P_1)$, $\text{IN}(P_2) = \text{OUT}(P_1) \cup \text{OUT}(P_2)$, and $\text{OUT}(P) = \text{OUT}(P_2)$.

(3) Parallel execution: $P$ runs $P_1, P_2, \ldots, P_k$ in parallel, where each $P_i, i = 1, 2, \ldots, k$, is an EWD program. We allow $k$ to depend on $\text{IN}(P)$. To avoid race conditions, we disallow $P_i$ to read any external memory space that is written by $P_j$ if $i \neq j$. But they can still read the same external memory space. In this case, define $W(P) = \sum_{i=1}^{k} W(P_i)$, $d(P) = \max_{1 \leq i \leq k} (d(P_i) + \text{IN}(P_i) + \text{OUT}(P_i))$ and $\text{OUT}(P) = \text{OUT}(P_1)$.

In addition, each $P_i$ knows its own index $i$.

Finally, we impose the uniformity requirement, i.e., the whole EWD program has constant description size. This means that although there can be a large number of subprograms, all of them are just instances of a constant number of distinct programs, possibly with different input parameters $\text{IN}(P)$ and indices $i$.

In addition to the ability to spawn parallel subprograms, there are still some technical differences between the (degenerated) EM model and the EWD model. First, the external memory is no longer partitioned into blocks; instead, an EWD algorithm can read/write an $M$ words in one I/O step. Thus, the EWD is potentially more powerful than the EM model. Indeed, later we show that sorting can be done in the EWD model in $O(N/M)$ I/Os while it takes $\Theta(N/M \cdot \log N)$ I/Os in the EM model. This is actually intentional; otherwise, we would face a similar logarithmic-factor gap as in the BSP-to-PEM simulation.

Second, a technical difficulty in designing EM algorithms is that reading/writing less than a full block of data is still counted as one I/O step. This often creates problem in both the design (one must try to fully utilize the data inside a block) and analysis (number of I/Os have to be rounded up, and all the ceilings often create technical inconveniences in the analysis). In the EWD model, the work as defined is just the actual amount of data read/written, divided by $M$. This often makes the analysis much easier, as illustrated in some examples shown later.

Thirdly, in the EM model, the contents in the internal memory can persist over multiple I/O steps. In the EWD model, we disallow this. More precisely, when a parent program spawns subprograms, each subprogram will start afresh, with nothing in its local memory. If the parent program wants to pass something to a subprogram, it will need to write the data to external memory for the subprogram to read later.

The following theorem, whose proof is deferred to Section 6, shows that we can simulate any EWD program on the CREW BSP in $O(d + W/p)$ rounds, which is optimal, provided that the EWD program uses $O(pM)$ space. This can be seen as the analogy of Brent’s Theorem [11] for the BSP.

**Theorem 3.** Any depth-$d$, work-$W$ program in the EWD model can be simulated on a CREW BSP with $p$ processors each with memory $M$ in $O(W/p + d)$ rounds, provided that the EWD program uses at most $pM$ words of external memory.

If a constant-round BSP algorithm is desired, then Theorem 3 implies that all we have to do is design an EWD algorithm with depth $d = O(1)$. Then $W = p$ will be the number of number of processors needed. Note that if the EWD program has work $W$, then it uses at most $O(W/M) = O(pW)$ external memory, so the pre-condition of Theorem 3 is always satisfied.

**Corollary 4.** If there is an EWD program with depth $d = O(1)$ and work $W$, then it can be simulated on a CREW BSP with $p = O(W)$ processors each with memory $M$ in $O(1)$ rounds.

For example, the EWD algorithm (Algorithm 1) has $d = O(1)$ and $W = O((N/M)^2)$, so it leads to a BSP algorithm with $p = O(N/M^2)$ in $O(1)$ rounds, matching the previous results [2, 25].

In fact, Theorem 3 leads to more general tradeoffs than previous BSP algorithms. Still using the Cartesian product example, previous results only give the tradeoff $p = O((N/M)^2)$ with $r = O(1)$. On the other hand, Theorem 3 gives the more general tradeoff $r = O(N^2/pM^2)$. Since Algorithm 1 used only $N$ space (just to store the input data), this tradeoff holds as long as $pM \geq N$. For example, if there are only $p = O(N/M)$ processors, then previous algorithms will just fail, while the algorithm implied by Theorem 3 can still solve the problem, although with more rounds $r = O(N/M)$, which is nevertheless optimal, since $p$ processors can emit at most $rpM^2$ join results in $r$ rounds.

## 5 ALGORITHM DESIGN IN THE EWD

In this section, we give a series of examples on how algorithms can be designed in the EWD. Because the EWD is a combination of the EM model and a PRAM work-depth model, the EWD algorithm is often either a parallel version of the corresponding EM algorithm or an external version of the corresponding PRAM algorithm. This allows us to build on the vast literature in these two areas.

We will be focusing on join algorithms (and the primitives they need), but should point out that the EWD is a general computational model for algorithm design. As a further example, in Appendix A we show how Strassen’s algorithm can be written in the EWD, thus recovering previous BSP algorithms [35, 40] by Corollary 4.

The formal definition of the EWD model only has a global external memory space $A$, but when designing EWD algorithms, for
convenience we often allocate some local memory space for each subprogram. By using some consistent allocation scheme (e.g., using the position of a subprogram in the recursive structure of the whole program), these local memory spaces can always be mapped to the global memory space.

5.1 Prefix-sums

The prefix-sums problem is defined as follows. Given \( N \) elements \( x_1, x_2, \ldots, x_N \), compute \( y_i = x_1 + x_2 + \cdots + x_i \) for each \( i = 1, 2, \ldots, N \), where \( \oplus \) is an associative operator. The EW algorithm presented below is a straightforward "externalization" of the PRAM algorithm except that the fanout increases from 2 to \( M \).

Assume the input data is stored at \( A[1, N] \). Our EW algorithm \( \mathcal{P} \) will write the output also at \( A[1, N] \) when it finishes. Set \( \text{IN}(\mathcal{P}) = (A, N) \), i.e., the starting address of \( A \) and its size. The program \( \mathcal{P} \) is defined recursively as follows. If \( N \leq M \), \( \mathcal{P} \) is just a base-case program solving the problem trivially. Otherwise, \( \mathcal{P} \) creates a local array \( B \) of size \( N/M \), and runs the following three EW programs \( \mathcal{P}^*, \mathcal{P}^**, \mathcal{P}^{***} \) sequentially (we omit how the IN and OUT parameters should be set, which is straightforward):

1. \( \mathcal{P}^* \) first gets the starting address of \( A \) and \( N \) from \( \text{IN}(\mathcal{P}^*) = \text{IN}(\mathcal{P}) \). Then it spawns \( N/M \) base-case programs. Each \( \mathcal{P}_i \) reads \( A[(i-1)M+1, iM] \) and writes the sum \( \sum_{j=(i-1)M+1}^{iM} A[j] \) to \( B[i] \).
2. \( \mathcal{P}^{**} \) runs \( \mathcal{P} \) recursively on \( (B, N/M) \), after which \( B \) has been replaced by its prefix-sums.
3. \( \mathcal{P}^{***} \) constructs \( N/M \) base-case programs in parallel. Each \( \mathcal{P}_i \) reads \( A[(i-1)M+1, iM] \) as well as \( B[i] \). Then it computes the prefix-sums of \( A[(i-1)M+1, iM] \) sequentially, starting from \( B[i] \), and writes the result back to \( A[(i-1)M+1, iM] \).

The depth of \( \mathcal{P} \) follows the recurrence \( d(N) = d(N/M) + O(1) \), so \( d(N) = O(\log_M \frac{N}{M}) = O(1) \). The total number of words read or written by \( \mathcal{P} \) is at most \( N + N/M + N/M^2 + \cdots + M = O(N) \), so \( W = O(N/M) \). When the work is \( O(N/M) \), we often say that it is linear.

5.2 Sorting

Our EW sorting algorithm is a parallel version of the EM distribution sort [4]. The basic idea of distribution sort is to find \( \sqrt{N} - 1 \) approximate splitters to partition the array into \( \sqrt{N} \) buckets, each of size \( O\left( \frac{N}{\sqrt{N}} \right) \), and then sort each bucket recursively. More precisely, we first divide \( A \) into chunks of size \( M \), use one subprogram to sort each chunk to find its \( j \sqrt{M} \)th element for \( j = 1, 2, \ldots, \sqrt{M} \) as representatives. Then, we sort all the \( \frac{N}{\sqrt{M}} \) representatives recursively to find \( \sqrt{N} - 1 \) splitters, denoted as \( b_1, b_2, \ldots, b_{\sqrt{N}-1} \) (define \( b_0 = -\infty \) and \( b_{\sqrt{N}} = +\infty \)) that evenly partition the representatives. We claim that the number of elements falling between every two consecutive splitters is \( O\left( \frac{N}{\sqrt{M}} \right) \). Consider the \( i \)-th chuck of \( A \). There are \( \sqrt{M} - 1 \) elements between every two consecutive representatives from this chunk. Consider any two consecutive splitters \( b_j, b_{j+1} \). Suppose \( x_i \) representatives from the \( i \)-th chunk fall inside \( (b_j, b_{j+1}) \). Then at most \( (x_i + 1)\sqrt{M} \) elements from this chunk fall inside \( (b_j, b_{j+1}) \). Summing over all \( \frac{N}{\sqrt{M}} \) chunks, the number of elements falling inside \( (b_j, b_{j+1}) \) is at most \( \sum_{i=1}^{\sqrt{M}} (x_i + 1)\sqrt{M} \leq \frac{2N}{\sqrt{M}} \), noticing that \( \sum_i x_i \leq N \) since the \( \sqrt{M} - 1 \) splitters evenly split the \( \frac{N}{\sqrt{M}} \) representatives.

Finally, we need to partition all the elements using the splitters. We will use one subprogram to partition one chunk of data, and write the elements to the correct bucket. In order to do so in parallel without conflicts, we need to find, for each chunk \( i \) and each bucket \( j \), the starting location in the array \( A \). To do so, we first find \( c_{i,j} \), the number of elements in chunk \( i \) going to bucket \( j \) for all \( i = 1, \ldots, \sqrt{M}, j = 1, \ldots, \sqrt{M} \). We create a temporary array \( B \) and set \( B[(j-1)\frac{N}{\sqrt{M}} + i] = c_{i,j} \). Then running prefix-sums on \( B \) will give us all the desired starting locations.

Next, we analyze the work and depth of this algorithm. It has \( O(\log_M \sqrt{N}) = O(1) \) levels of recursion. On each level, running prefix-sums has \( O(1) \) depth. Thus the overall depth is still \( O(1) \). The work follows the recurrence

\[
W(N) \leq W\left( N\sqrt{M} \right) + \sum_j W(N_j) + O\left( \frac{N}{M} \right),
\]

where \( N_j = O\left( \frac{N}{\sqrt{M}} \right), \sum_j N_j = N \). Although this recurrence grows with a rate of \( 1 + \frac{1}{\sqrt{M}} \), but because of its constant depth, it still solves to \( W(N) = O(N/M) \).

5.3 Binary joins

As a direct application of the primitives developed so far, we show how the sort-merge join algorithm can be written in the EW, which computes the join \( R_1(A, B) \bowtie R_2(B, C) \). The algorithm (Algorithm 2), simply replaces all its steps with their EW counterparts.

In fact, this algorithm is even simpler than the sort-merge join algorithm in the EM model, which cannot afford to call PCartesian if \( |\sigma_{B=b}R_1| \) or \( |\sigma_{B=b}R_2| \) is smaller than one block, which would waste one I/O. Thanks to the way how work is defined in the EW, this is no longer an issue.

Let \( T = \sum_{b \in \text{dom}(R) \cap \sigma_{B=b}R_1 \cap \sigma_{B=b}R_2} |\sigma_{B=b}R_1| \cdot |\sigma_{B=b}R_2| \) be the join size. Plugging in the results on sorting, prefix-sums, and Cartesian product, we get that the work of this algorithm is \( O\left( \frac{T}{M^2} + \frac{N}{\sqrt{M}} \right) \). The depth is obviously \( O(1) \). Then applying Corollary 4 allows us to match the previous results (see Table 2).

\begin{algorithm}
1. Sort \( R_1 \) and \( R_2 \) by join attribute \( B \);
2. Find the starting locations of tuples with \( B = b \) in \( R_1 \) and \( R_2 \) for each \( b \in \text{dom}(B) \) using prefix-sums;
3. for each \( b \in \text{dom}(B) \) do in parallel
4. \( \text{PCartesian}(\sigma_{B=b}R_1, \sigma_{B=b}R_2) \);
\end{algorithm}

5.4 Simulating BSP algorithms

Next, we show that any BSP algorithm can be simulated optimally in the EW. This allows us to reuse all known BSP algorithms in the EW.
An External-Memory Work-Depth Model and Its Applications to Massively Parallel Join Algorithms

Theorem 5. Any algorithm that runs in the EREW BSP model in \( r \) rounds using \( p \) processors with local memory size \( M \) can be simulated by an EWD program with depth \( O(r) \) and work \( O(pr) \).

Proof. We create an array \( A \) of size \( pM \) that stores the memory contents of the \( p \) BSP processors consecutively. We also create an auxiliary array \( B \) of size \( p + 1 \) that stores boundaries of the memory contents as they are stored in \( A \). Initially, \( A \) stores the input, and \( B[i] = (i - 1)M + 1 \). Below we show how each BSP round can be simulated by an EWD program with work \( O(p) \) and depth \( O(1) \). We spawn \( p \) subprograms \( \mathcal{P}_i, i = 1, \ldots, p \) in parallel to simulate the \( p \) processors, as follows.

1. Receiving messages: Each \( \mathcal{P}_i \) first reads \( B[i] \) and \( B[i + 1] \), and then reads the data at \( A[B[i], B[i + 1] - 1] \).
2. Local computation: \( \mathcal{P}_i \) performs the same computation as the corresponding BSP processor does.
3. Sending message: If the corresponding BSP processor sends a word \( w \) to processor \( j \), \( \mathcal{P}_i \) writes a pair \( (j, w) \) in \( A[(i - 1)M + 1, iM] \). If the BSP processors send out less than \( M \) words, we pack dummy pairs to \( A[(i - 1)M + 1, iM] \). Then we sort \( A[1, pM] \) by \( j \), after which we use prefix-sums to find the boundaries of messages destined to the processors, and store them in \( B \).

Theorem 5 allows us to deploy any BSP algorithm in the EWD (but we cannot apply it to Goodrich’s BSP sorting algorithm to get an EWD sorting algorithm, since the proof of this theorem uses EWD sorting as a primitive). In particular, we mention the following primitives that can all be solved on the BSP with \( p = O(N/M) \) processors in \( O(1) \) rounds. Therefore, they immediately lead to EWD programs with linear work and constant depth.

ReduceByKey [25]: Given \( N \) (key, value) pairs, compute the sum of values for each key, where the sum is defined by any associative operator.

MultiNumbering [25]: Given \( N \) (key, value) pairs, for each key assigns consecutive numbers \( 1, 2, 3, \ldots \) to the pairs with the same key.

MultiSearch [21, 25]: Given \( N \) elements \( x_1, x_2, \ldots, x_N \) as set \( X \) and \( N \) elements \( y_1, y_2, \ldots, y_N \) as set \( Y \), for each \( x_i \) find its predecessor in \( Y \).

Semijoin: Given two relations \( R_1 \) and \( R_2 \) with common attribute \( A \), the semijoin \( R_1 \times R_2 \) returns all the tuples in \( R_1 \) whose value on \( A \) matches that of at least one tuple in \( R_2 \). This can be reduced to MultiSearch: For each \( t \in R_1 \), if its predecessor on \( A \) is the same as that of \( t \), then it is in the semijoin.

ParallelPacking: Given \( N \) numbers \( x_1, x_2, \ldots, x_N \) where \( 0 < x_1 \leq 1 \) for \( i = 1, 2, \ldots, N \), group them into \( m \) sets \( Y_1, Y_2, \ldots, Y_m \) such that \( \sum_{j \in Y_i} x_j \leq 1 \) for all \( j \), and \( \sum_{j \in Y_i} x_j \geq \frac{1}{2} \) for all but one \( j \). Initially, the \( N \) numbers are stored on \( \frac{N}{M} \) processors evenly, and the algorithm should produce all pairs \( (i, j) \) if \( i \in Y_j \) when done.

We are not aware of an explicit reference for this primitive, but it can be solved quite easily, as follows. We ask each processor \( i \) to first perform grouping on its local data. It is obvious that the condition above can be satisfied. The processor then reports two numbers: \( g_i \), the number of groups with sum between 1/2 and 1, and \( h_i \), the sum of remaining group with sum smaller than 1/2. Note that \( g_i \) and \( h_i \) can be 0. Next, we run the BSP algorithm for prefix-sums [21] on the \( g_i \)'s. After that, we can assign consecutive group id's to each of the \( g_i \) groups on each processor \( i \). For the remaining \( p \) partial groups whose sums are \( h_i \) with \( 0 < h_i < 1/2 \), we recursively run the algorithm, using group id’s starting from \( \sum_i g_i + 1 \). After the recursion ends, for each partial group \( h_i \) that has been assigned to group \( j \), we assign every element in \( h_i \) to group \( j \). The problem size reduces by a factor of \( M \) after each round, so the number of rounds is \( O(\log M, N) = O(1) \).

Other operations can be built on top of the primitives above. For example, a full-reducer for removing dangling tuples in an acyclic-join [38, 45] can be implemented with a constant number of semi-join operations.

5.5 Triangle Join

Now consider the triangle join \( R(A, B) \bowtie S(B, C) \bowtie T(A, C) \). We present an EWD algorithm below based on the EM algorithm in [23]. For simplicity, our algorithm may emit the same triangle twice; some tie breaking rule can be used so that each triangle is emitted exactly once.

For each value \( a \in \text{dom}(A) \), let its degree in \( R \) be \( |\sigma_{A=a} R| \). Note that the degrees of all values can be computed using ReduceByKey. We call a value \( a \in \text{dom}(A) \) heavy if \( |\sigma_{A=a} R| \geq \sqrt{M^2 + \frac{M - 1}{|T|}} \), otherwise light, and a value \( b \in \text{dom}(B) \) heavy if \( |\sigma_{B=b} R| \geq \sqrt{M^2 + \frac{M - 1}{|S|}} \), otherwise light. The basic idea is to classify the triangles into two cases: (1) at least \( A \) or \( B \) is heavy; and (2) both \( A \) and \( B \) are light.

Algorithm PTriangleJoin obviously has depth \( O(1) \). The work analysis is almost the same as the I/O analysis for the EM algorithm [23]. Since all the primitives take linear work, while they incur an extra \( O(\log N) \) factor in the EM model, the work in the EWD model does not have any logarithmic factor as in [23], and is simply \( O\left(\sqrt{|R||S||T|}M^{-1.5}\right) \). Again applying Corollary 4 gives us the result in Table 2.

5.6 Loomis-Whitney Join

The Loomis-Whitney (LW) join query is defined by the hypergraph \( Q = (V, E) \) where \( V = \{v_1, v_2, \ldots, v_n\} \) and \( e_i = \{v_i\} \) for each \( e_i \in E \). The optimal EM algorithm [23] has I/O complexity \( O\left(\frac{N}{M} \left\lceil \frac{n}{M} \right\rceil \right) \). It is based on two base-case joins, small join when one relation fits in main memory, and point join when there is only one value in the domain of an attribute. Both joins can be done in linear I/O cost. The main idea is to divide the original join into sub-joins by one attribute each time and solve each subjoin recursively. Both the small join and the point join can be converted to the EWD model directly by replacing the operations by their EWD counterparts, using a generalized PSortMerge defined on \( m > 2 \) relations. The algorithm is the same as Algorithm 2 except that we use \( m \) nested loops to compute the Cartesian product of \( m \) relations for PCartesian.
13 for each heavy $a \in \text{dom}(A)$ do in parallel
14 $S_a = S \times \sigma_{A=a,R} \times \sigma_{A=a,T}$ (SemiJoin);
15 for each memory chunk $M$ of $S_a$ do in parallel
16 Emit($t$, $a$) for each $t \in M$;
17 end for
18 end for

Algorithm 3: PTriangleJoin($R, S, T$)

19 SmallJoin: Assume $N_1 \leq M$. First we compute $R_1 \times R_1$ on attributes $V = \{v_1, v_1\}$ for $i = 2, 3, \ldots, n$ in parallel, and then apply PSortMerge to $m = 2, 3, \ldots, n$ ($R_1 \times R_1$) on attribute $v_1$.

20 PointJoin: Assume $|\text{dom}(v_1)| = 1$. First we compute $R_1 \times R_i$ on attributes $V = \{v_1, v_1\}$ for $i = 2, 3, \ldots, n$ in parallel, and then the intersection $|\cap_i = 2, 3, \ldots, n (R_1 \times R_i)$ by PSortMerge.

21 Both joins have depth $O(1)$. The work analysis is almost the same as the I/O analysis for the EM algorithm [23]. Thus both joins lead to EWD programs with linear work and constant depth.

22 The complete EWD program for LW join is presented as PLWJoin, with parameters $\tau_l = (N^{m-2}M^{-1})^\frac{1}{M}$ for $i = 1, 2, \ldots, n$. It has depth $O(1)$. Since all the primitives take linear work, following the same I/O analysis in [23], we get that the work of the EWD algorithm is $O\left(\frac{N}{M}\tau^2\right)$, as claimed in Table 2.

5.7 Simulating PRAM Work-Depth Algorithms

The EWD model allows each base-case program to read/write up to $M$ words in external memory, but nothing prevents it from just reading $O(1)$ words. When the algorithm does so, it is just a PRAM work-depth algorithm. This implies that any PRAM work-depth algorithm $A$ with work $W(A)$ and depth $d(A)$ (measured in the PRAM model) will have work $W(A)/M$ and depth $d(A)$ in the EWD model. It turns out that this simple observation leads to some new bounds for BSP joins algorithms.

The worst-case optimal RAM algorithms for arbitrary joins have a running time of $O(N^p)$ [36, 37, 43]. Although it has not been explicitly stated, the Generic-Join algorithm [37] can be easily written in the PRAM work-depth model with work $O(N^p)$ and depth $O(1)$. Thus Corollary 4 implies a BSP algorithm with $p = O\left(\frac{N^p}{M}\right)$. Currently, the best BSP algorithm for arbitrary joins has $p = O\left(\frac{N^{p^*}}{M}\right)$, where $\psi^*$ is the edge quasi-packing number [32] of the join query. Since $p^* \leq \psi^*$, these two bounds are incomparable. But one advantage of the $O\left(\frac{N^p}{M}\right)$ algorithm is that it actually writes all join results to external memory, while the $O\left(\frac{N^{p^*}}{M}\right)$ algorithm only emits the join results. This advantage is useful when combined with the parallel Yannakakis algorithm [1] in the generalized hypertree decomposition (GHD) framework [22], which needs to write down the parallel Yannakakis algorithm [1] in the generalized hypertree decomposition (GHD) framework [22], which needs to write down the intermediate join results. Indeed, the GYM algorithm [1, 33] has $p = \hat{O}\left(\frac{N^{max}_{\psi^*}}{M}\right)$, because it uses the $O\left(\frac{N^{\psi^*}}{M}\right)$ algorithm to compute the join in each $t$. Replacing it with the $O\left(\frac{N^{p^*}}{M}\right)$ algorithm removes the $O\left(\frac{N^{max}_{\psi^*}(t)}{M}\right)$ term, simplifying and improving the result (Table 2).

If one is allowed to express the bound using degree information, then the best RAM algorithm has running time $p = O(N^{MO})$, where $MO$ is a degree-based join size upper bound introduced in [28]. We observe that their RAM algorithm [28] can be easily written in the PRAM work-depth model (detailed provided in Appendix B) with work $O(N^{MO})$ and depth $O(1)$. Applying Corollary 4 yields a BSP algorithm with $p = \hat{O}\left(\frac{N^{MO}}{M}\right)$. This improves upon the DBP based BSP algorithm [28], which has $p = \hat{O}\left(\frac{N^{DBP}(\mathcal{R})}{M} + N^{DBP}(\mathcal{M})\right)$. DBP

\footnote{In [28], the claimed bound is $p = \hat{O}\left(\frac{N^{DBP}(\mathcal{R})}{M} + N^{DBP}(\mathcal{M})\right)$, but the correct bound should be $\hat{O}\left(\frac{N^{DBP}(\mathcal{R})}{M} + N^{DBP}(\mathcal{M})\right)$.}

\newpage

Algorithm 4: PLWJoin($R_1, R_2, \ldots, R_n, l$)

1 if $N_1 \leq M$ then
2 SmallJoin($R_1, R_2, \ldots, R_n$);
3 else if there exists $j$ such that $|\text{dom}(v_j)| = 1$ then
4 PointJoin($R_1, R_{j+1}, \ldots, R_{j-1}$);
5 else
6 $h = \min\{h : l + 1 \leq h \leq d, \tau_k < \frac{T}{M}\}$;
7 Identify heavy and light values of $v_h$ (ReduceByKey);
8 Sort $R_i$ by attribute $v_h$ for $i = 2, 3, \ldots, n$;
9 for each heavy $a \in \text{dom}(v_h)$ do in parallel
10 PointJoin($R_h, \sigma_{v_h=a}R_{h+1}, \ldots, \sigma_{v_h=a}R_{h-1}$);
11 Put light values of dom(v) into $m = O\left(\frac{N}{M}\right)$ groups $I_1, I_2, \ldots, I_m$ such that the total degree of $R_i$ on each group is $O(\tau_h)$ (ParallelPartitioning);
12 for $i = 1, 2, \ldots, n$ (– $h$) do in parallel
13 Tag each $t \in R_i$ with $j \in \text{dom}(v_h)$ (MultiSearch);
14 Sort $R_i$ by the attached integer;
15 for $j = 1, 2, \ldots, m$ do in parallel
16 PLWJoin($\sigma_{v_h\in I_j}R_1, \ldots, R_h, \ldots, \sigma_{v_h\in I_j}R_n, l$);
An External-Memory Work-Depth Model and Its Applications to Massively Parallel Join Algorithms

is another degree-based join size upper bound defined in [28], but
looser than MO, as it has been shown that MO \leq DBP(2) + c \log^2 n \leq 2
for some constant c [28]. But we should point out that the DBP
based algorithm runs in 3 rounds for any query, while the MO based
algorithm runs in O(n) rounds where n is the number in attributes of
the query. Furthermore, using the \( \tilde{O}\left(\frac{N^{MO}}{pM}\right) \) algorithm in the GHD
framework yields a BSP algorithm with \( p = \tilde{O}\left(\frac{N^{MO}(pM)}{M} + \frac{\ell}{M}\right) \),
further improving the result above, since MO(t) \leq p^*(t) for any t.

6 FROM EWD TO BSP

In this section, we prove Theorem 3, which is the analogy of Brent’s
Theorem for the BSP. The proof of Brent’s Theorem is straightforward
if we assume a centralized scheduler. However, to make the
simulation rigorous, the scheduler itself also has to run in a distributed
fashion, which is nontrivial [12, 13]. Here, we face a similar
situation, that we need to run our scheduler on the BSP as well.

We will actually make use of a total of \( O(p) \) processors, which
do not affect the asymptotic bound by scaling down the given p.
We use \( p \) processors for the actual simulation, \( p \) processors to serve
as the external memory, and another \( O(p) \) processors to serve as
schedulers. The \( p \) processors serving as external memory will keep
a list of \((i, A[i])\) pairs where \( A[i] \) is nonzero.

Unfolding the recursive definition, an EWD program can be
modeled as a directed acyclic graph (DAG), where each node is a
base-case program (below we will not distinguish a node and its
corresponding base-case program). Note that this DAG is generated
in an online fashion, namely, we can only see a node/program \( P \)
after all its predecessors have been executed. Therefore, we will
simulate the DAG level by level. We will achieve the desired bound
if we can simulate each level in \( O(W/\ell/p) \) rounds, where \( W_\ell \) is the
total work on this level.

One technical difficulty is that a level in this DAG may have up
to \( W/\ell \) nodes, which can be larger than the total memory space
on the BSP. To resolve this difficulty, we introduce the notion of
footprint and closure. For a node \( P \) in the DAG, if \( OUT(P) \neq null \),
its footprint includes (1) \( OUT(P) \), (2) \( k \), the number of nodes \( P' \)
in the next level with \( IN(P') = OUT(P) \), and (3) the description
of the algorithm associated with each such \( P' \). If \( OUT(P) = null \),
then it has no footprint. Note that although \( P \) may be followed by
any number of nodes \( P' \) in the next level, but their description can
only have constant size. In fact, the whole EWD program can be
described in constant size by the uniformity requirement. For any
node \( P \), its closure\(^7\) includes (1) the description of the algorithm
associated with \( P \), (2) its input parameter \( IN(P) \), (3) its index \( i \) if it is
one of the \( k \) subprograms spawned in parallel sharing the same
\( IN(P) \), and (4) \( R(P) \), the number of words \( P \) will read (which can be
computed from (1), (2), and (3) — recall that \( P \) cannot make
adaptive reads). The closure also has constant size, and it contains
all the information needed to simulate \( P \). The key observation is
that, although there can be too many closures, there can be at most
\( pM \) footprints on each level. This is because the EWD program uses
at most \( pM \) external memory, and we require \( OUT(P) \) to be a word
written by \( P \). We can then generate the closures in batches of size
\( O(pM) \) as the next level unfolds.

To simulate a level, we proceed as follows. We inductively assume
that the footprints of all nodes on the last level have been stored
on the schedulers. Initially, there is only one footprint associated
with the root node.

1. The examiners check their stored footprints, and from
each footprint, find \( k \), the number of nodes in the next level
whose closure will be generated from it. Using parallel pack-
ing, we put the footprints into groups so that each group
will generate \( \Theta(pM) \) closures except for one. For each group
of footprints, perform the following steps.

2. The schedulers generate the closures from the footprints
in this group. Some schedulers may produce more than \( M \)
closures, so we may need to redistribute the load. This can
be achieved by first running prefix-sums on the number of closures
each scheduler produces.

3. The schedulers examine the \( R(P) \) in each closure, and put
them (using parallel packing) into groups, so that in each
group except for one, the sum of \( R(P)'s \) is \( \Theta(pM) \). For each
group of closures, perform the following steps.

4. Further divide the closures (using parallel packing) into
groups, so that in each group except for one, the sum of the
\( R(P)'s \) is \( \Theta(M) \). Each group of closures can now be executed
by one processor. The execution consists of the following
steps.

5. Reading phase: For each closure assigned to a processor \( j \),
the processor extracts the description of \( P, IN(P) \), index
\( i \) (if available), and produces a list of \((i, j)\) pairs, where \( i \) is a
memory address \( P \) intends to read from. Using multi-
search [25] on the p processors storing the current external
memory contents, we fetch \( A[i] \) to processor \( j \) for all \( i, j \).

6. Local computation: For each node \( P \) assigned to processor \( j \),
the processor performs the same computation as \( P \) does.

7. Writing phase: For each node \( P \) assigned to processor \( j \),
the processor produces a list of \((i, w)\) pairs where \( w \) is a
word \( P \) intends to write to \( A[i] \). Recall that by definition,
nodes on the same level are not allowed to write to the same
external memory locations, so all the \( i's \) are distinct. Also,
by the assumption that the EWD program uses at most \( pM \)
words of memory, there are at most \( pM \) such \((i, w)\) pairs. We
then sort these \((i, w)\) pairs with the existing \((i, A[i])\) pairs to
update the external memory contents.

8. Finally, for each node \( P \) with \( OUT(P) \neq null \), we store its
footprint on the schedulers. As argued earlier, there are at
most \( pM \) footprints over the entire level. Similar to step (2),
to avoid overloading any scheduler with storing more than
\( O(M) \) footprints, we may need to redistribute load.

To see that each level can be simulated in \( O(W/\ell/p) \) rounds, we
make the following observation: Inside each full group of footprints
(i.e., having \( \Theta(pM) \) closures), we can create at least one full group
of closures (i.e., the sum of the \( R(P)'s \) is \( \Theta(pM) \)). Thus, the cost
for dealing with each partial group of closures can be charged to
another full group of closures, while each group of closures (partial
or full) can be handled in \( O(1) \) rounds.

\(^7\)Terminology borrowed from Spark.
7 FUTURE DIRECTIONS

Although this paper is only about theory, it would very interesting to study its implications to the programming models for today’s massively parallel systems. In fact, the recent transition from MapReduce and Hadoop to newer systems like Spark and Flink can also be seen as a step taken from a low-level parallel programming model to a work-depth-like model. Both Spark and Flink define their programs as a DAG, similar to the EWD model. However, a key difference between the EWD and Spark’s DAG model is that Spark’s DAG is statically defined at compile time, while EWD’s DAG is dynamically generated at runtime. A static DAG can have the same depth as the dynamic one but can be polynomially larger in size (i.e., work). Essentially, this is equivalent to simulating the PRAM by circuits [41]. Implementing a dynamic DAG model would be highly rewarding, as it gives the programmer much more flexibility and potentially huge (polynomial) improvements in resource consumption. It could involve some challenging system work, but not impossible. Indeed, most of the complications in our EWD-to-BSP simulation is to take care of the case $M \ll p$. In practice, $M$ is large enough so that we can run a centralized scheduler at the master node.

REFERENCES


A STRASSEN’S ALGORITHM IN EWD

Let $A$ and $B$ be two $\sqrt{N} \times \sqrt{N}$ matrices and we wish to compute $C = AB$. Unfolding Strassen’s recursion for $h$ levels, we divide $A$ and $B$ into $2^h \times 2^h$ sub-matrices of dimension $\sqrt{N/2^h} \times \sqrt{N/2^h}$. We number these sub-matrices as $A_j$ and $B_j$ for $i, j = 1, 2, \ldots, 4^h$. We similarly divide $C$ into $2^h \times 2^h$ sub-matrices $C_j$, $l = 1, 2, \ldots, 4^h$. Strassen’s recursion yields a series of coefficients $\alpha(i, k), \beta(j, k), \gamma(l, k), \chi$. 

X. Hu et al.
can be purely computed from \( h \), such that there are \( 7^h \) matrices \( P_k \) such that

\[
P_k = \sum_i a(i, k)A_i \cdot \sum_j \beta(j, k)B_j,
\]

\[
C_l = \sum_k \gamma(l, k)P_k.
\]

Set \( h = \log_2 M \). In the EWD program, each level of recursion will correspond to \( h \) levels of Strassen’s recursion. For simplicity we assume that \( \log_2 M \) is a multiple of \( h \), so that the EWD program will have \( O(1) \) recursion depth and each base case deals with sub-matrices of size \( \sqrt{M} \times \sqrt{M} \). For other values of \( \log_2 M \), we can use a smaller \( h \) in the first level of recursion. The EWD algorithm can be quite straightforwardly written; please see Algorithm 5.

**Algorithm 5: PStrassen\((A, B)\)**

1. If \( A \) and \( B \) have size smaller than \( M \) then
2. Read \( A \) and \( B \) into memory;
3. Write \( A \times B \) back to external memory;
4. Create \( 3M = 3 \cdot 7^h \) sub-matrices \( P_{k, A_i B_j} \) in local space;
5. For each \((x, y) \in [\sqrt{N}/2^h] \times [\sqrt{N}/2^h]\) do in parallel
6. For \( k = 1, \ldots, M \) do
7. \[ A_k(x, y) \leftarrow \sum_i a(i, k)A_i(x, y); \]
8. \[ B_k(x, y) \leftarrow \sum_j \beta(j, k)B_j(x, y); \]
9. For \( k = 1, 2, \ldots, M \) do in parallel
10. \[ P_k = PStrassen(A_k, B_k); \]
11. For each \((x, y) \in [\sqrt{N}/2^h] \times [\sqrt{N}/2^h]\) do in parallel
12. \[ C_i(x, y) \leftarrow \sum_k \gamma(l, k)P_k(x, y); \]

The algorithm has \( O(1) \) depth as argued earlier. To analyze the work, notice that line 6–8 writes a total of \( M \) words and reads less than \( M \) words, so it is a base-case program with \( O(1) \) work. In line 12, we read \( M \) words and write less than \( M \) words, so it also has work \( O(1) \). Thus, the total work has the recurrence

\[
W(N) = 7^h \cdot W\left(\frac{N}{4^h}\right) + O\left(\frac{N}{2^h}\right),
\]

which solves to \( W(N) = O\left(\frac{N}{3^h} \cdot \log_7 N\right) \).

**B A DEGREE-BASED JOIN ALGORITHM**

Degree-based join processing was introduced in [28]. First, they observe that the degree information can be computed in the BSP with \( p = O(N/M) \) processors and \( O(1) \) rounds. Then, they decompose the join into a polylogarithmic sub-joins, such that for each sub-join, for any subset of attributes, all values have the same degree (up to a constant factor) within each relation. Below, we will focus on computing one such sub-join.

The MO bound for a sub-join is defined as the objective value of the following linear program:

\[
\max \ s_V,
\]

\[ s.t. \ (1) s_X = 0; (2) s_Y \leq s_Y \text{ if } X \subseteq Y \subseteq \mathcal{V}; \]

\[ (3) s_{Y \cup Z} \leq s_{X \cup Z} + \deg(X, Y, e) \forall X \subseteq Y \subseteq e, e \in \mathcal{E}, Z \subseteq \mathcal{V}; \]

where \( \deg(A, B, e) \) is the maximum degree of values over attributes \( A \) in subrelation \( \pi_B R(e) \), taking logarithm with base \( N \), i.e.,

\[
\deg(A, B, e) = \log N \cdot \max_{t \in \pi_A R(e)} \left[ \{ t' \in \pi_B R(e) : \pi_A t' = t \} \right].
\]

Then, their observation is that when the objective \( s_V = \text{MO} \) is achieved, there must be a chain \( V_0, V_1, \ldots, V_k \), where \( V_0 = \emptyset \), \( V_k = \mathcal{V}, V_i \subseteq V_j \text{ for } i < j \), such that for each \( i = 0, 1, \ldots, n - 1 \), there is a tight constraint of \( 3 \) with \( V_{i+1} \) on the LHS and \( V_i \) on the RHS, i.e., \( s_{V_{i+1}} = s_{V_i} + \deg(X, Y, e) \) where \( X \subseteq Y \subseteq e_i, V_i = X \cup Z \) and \( V_{i+1} = Y \cup Z \) for some \( Z \subseteq \mathcal{V} \). The algorithm PMOJoin then follows this chain to compute the join, as described in Algorithm 6.

**Algorithm 6: PMOJoin\((V_0, V_1, \ldots, V_n, R)\)**

1. \( L_{n-1} \leftarrow \text{PMOJoin}(V_0, V_1, \ldots, V_{n-1}, R); \)
2. \( L_n \leftarrow \pi_{V_n} R(e_n); \)
3. For each \( e \in \mathcal{E} \) do
4. \( L_n \leftarrow \pi_{V_n} R(e); \)
5. Return \( L_n \).

The algorithm has been shown to have running time \( O(N^{\text{MO}}) \) in the RAM model [28]. Replacing the binary join in line 2 and the semi-join in line 4 by their EWD counterparts, this algorithm can be executed in the EWD model with work \( W = O(N^{\text{MO}}/M) \) and depth \( d = O(n) = O(1) \).

**C BERGE-ACYCLIC JOIN**

A join query is Berge-acyclic if its hypergraph \((\mathcal{V}, \mathcal{E})\) is Berge-acyclic [10]. The optimal EM algorithm [26] is known with I/O complexity \( \tilde{O}\left(\left(\frac{N}{M}\right)^{\rho^*}\right) \) where \( \rho^* \). Their EM algorithm first classifies some special edges as island, bud and leaf, and then recursively peels off a bud relation, an island relation or a leaf relation from the join query each time. The EWD program is essentially the same of the EM version by replacing some operations by their counterparts, as shown in Algorithm 7.

The algorithm has depth \( O(1) \). The analysis of work is essentially the same as that for the EM algorithm. Since all the primitives take linear work, the work in the EWD model is simply \( O\left(\left(\frac{N}{M}\right)^{\rho^*}\right) \).
Algorithm 7: PacyclicJoin($V, E, R$)

```plaintext
1 if $E = \{e\}$ then
2     for each memory chunk $M$ of $R(e)$ do in parallel
3         Emit $t \in M$ if $t$ participating in a join result;
4 else if there is a bud $e \in E$ then
5     $R(e') \leftarrow R(e') \times R(e)$ for each $e' \in \Gamma(e)$ (SemiJoin);
6     PacyclicJoin($V, E - \{e\}, R$);
7 else if there is an island $e \in E$ then
8     for each memory chunk $M$ of $R(e)$ do in parallel
9         $L \leftarrow$ Pacyclic($V - e, E - \{e\}, R$);
10        Emit matched $(t, t')$ for every $t' \in M$ and $t \in L$;
11 else
12    Put light values of $\text{dom}(v)$ into $m = O(N(e)/M)$ groups $I_1, I_2, \ldots, I_m$ such that the total degree of $R(e)$ on each group is $\Theta(M)$ (ParallelPacking);
13    for $j \in \{1, 2, \ldots, m\}$ do in parallel
14        Read $\sigma_{v=a} R(e)$ into memory as $M$;
15        $R(e') \leftarrow R(e') \times M$ for each $e' \in \Gamma(e)$ (SemiJoin);
16        $L \leftarrow$ PacyclicJoin($V - U, E - \{e\}, R$);
17        Emit matched $(t, t')$ for every $t' \in M$ and $t \in L$;
```

X. Hu et al.