Non Linear Divisible Loads: There is No Free Lunch

Olivier Beaumont, Hubert Larchevêque
INRIA Bordeaux Sud-Ouest F-78000 and LaBRI - Univ. of Bordeaux F-33400 and LaBRI - CNRS F-33400
Loris Marchal
CNRS and Univ. of Lyon

***Abstract*—Divisible Load Theory (DLT) has received a lot of attention in the past decade. A divisible load is a perfect parallel task, that can be split arbitrarily and executed in parallel on a set of possibly heterogeneous resources. The success of DLT is strongly related to the existence of many optimal resource allocation and scheduling algorithms, what strongly differs from general scheduling theory. Moreover, recently, close relationships have been underlined between DLT, that provides a fruitful theoretical framework for scheduling jobs on heterogeneous platforms, and MapReduce, that provides a simple and efficient programming framework to deploy applications on large scale distributed platforms.**

The success of both have suggested to extend their framework to non-linear complexity tasks. In this paper, we show that both DLT and MapReduce are better suited to workloads with linear complexity. In particular, we prove that divisible load theory cannot directly be applied to quadratic workloads, such as it has been proposed recently. We precisely state the limits for classical DLT studies and we review and propose solutions based on a careful preparation of the dataset and clever data partitioning algorithms. In particular, through simulations, we show the possible impact of this approach on the volume of communications generated by MapReduce, in the context of Matrix Multiplication and Outer Product algorithms.

Keywords: Divisible Load Theory, MapReduce, Scheduling, Resource Allocation, Matrix Multiplication, Sorting

I. Introduction

A. Divisible Load Theory and MapReduce

Scheduling the tasks of a parallel application on the resources of a distributed computing platform is critical for achieving high performance, and thus it is the target of many research projects, both in theoretical studies and in software developments.

As far as theory is concerned, the scheduling problem has been studied for a variety of application models, depending on the nature of the dependencies between tasks. In the case of general dependencies, the well-known directed acyclic task graph model is used, and many scheduling heuristics have been developed [1]. A slightly less general model corresponds to the parallel execution of several (usually many, so that scheduling concentrates on the steady state phase) pipelined applications that execute concurrently on a given platform, in particular for applications whose structure is a linear chain of tasks. Such applications are ubiquitous in streaming environments, as for instance video and audio encoding and decoding, DSP applications, image processing, and so on [2], [3], [4], [5]. An extreme (in terms of dependencies) application model is that of independent tasks with no task synchronizations and no inter-task communications. Applications conforming to this admittedly simple model arise in most fields of science and engineering. The number of tasks and the task sizes (*i.e.* their computational costs) may or may not be set in advance. In this case, the scheduling problems are akin to off-line and on-line bin-packing and a number of heuristics have been proposed in the literature (see [6], [7], [8] for surveys in off-line and on-line contexts).

Another flavor of the independent tasks model is one in which the number of tasks and the task sizes can be chosen arbitrarily. This corresponds to the case when the application consists of an amount (a large amount in general) of computations, or load, that can be divided into any number of independent pieces. This corresponds to a perfectly parallel job, whose sub-tasks can themselves be processed in parallel on any number of resources. This divisible load model has been widely studied, once Divisible Load Theory (DLT) has been popularized by the landmark book [9]. DLT provides a practical framework for the mapping of independent tasks onto heterogeneous platforms, and has been applied to a large spectrum of scientific problems, including Kalman filtering [10], image processing [11], video and multimedia broadcasting [12], [13] database searching [14], [15], and the processing of large distributed files [16]. These applications are amenable to the simple master-worker programming model and thus can be easily implemented and deployed on computing platforms ranging from small commodity clusters to computational grids [17] and Clouds [18],

[19], [20]. From a theoretical standpoint, the success of the divisible load model is mostly due to its analytical tractability. Optimal algorithms and closed-form formulas exist for the simplest instances of the divisible load problem. This is in sharp contrast with the theory of task graph scheduling and streaming applications scheduling, which abounds in NP-completeness [21] and in inapproximability results [22].

On the side of software development for scheduling parallel applications, the MapReduce framework [23], [24] has recently received at lot of attention. Indeed, the MapReduce model, which has been popularized by Google, is particularly well-suited to parallel processing of arbitrary data. Just as in the case of divisible load, a large computation is broken into small tasks that run in parallel on multiple machines (the case of a sequence of Map and Reduce operations has been studied in [25], and scales easily to very large clusters of inexpensive commodity computers. Hadoop [26] is the most popular open-source implementation of the MapReduce framework, originally developed by Yahoo to manage jobs that produce hundreds of terabytes of data on thousands of cores. Examples of applications implemented with Hadoop can be found at http://wiki.apache.org/hadoop/PoweredBy. A crucial feature of MapReduce is its inherent capability of handling hardware failures and processing capabilities heterogeneity, thus hiding this complexity to the programmer, by relying on on-demand allocations and a detection of nodes that perform poorly (in order to re-assign tasks that slow down the process).

The MapReduce programming model has first been introduced to deal with linear complexity tasks such as standard text processing operations [23], but it has been later extended to many other types of operations, such as linear algebra operations [27]. In this case, the Map function, that is applied in parallel to every pair in the input, operates on a prepared dataset. For instance, in the case of a matrix product, one could imagine to have as input dataset all compatible pairs $(a_{i,k}, b_{k,j})$ for all n^3 possible values of i, j and k. In this case, the output of the Map operation would be a pair consisting of the value $a_{i,k} \times b_{k,j}$ and the key (i, j), so that all $a_{i,k} \times b_{k,j}$, for $0 \leq k \leq n-1$ would be associated to the same key (i, j) and therefore to the same reducer, that would in turn be responsible for computing their sum. While allowing complex computations to run over MapReduce, such solutions lead to a large redundancy in data communication: if a given processor is responsible for computing many $a_{i,k} \times b_{k,j}$ products, it will receives as many values of a and b, even if some of them overlap.

B. Model

In this paper, the target computing platform is a heterogeneous master/worker platform, with p computing resources labeled $P_1, P_2, \ldots, P_p$. The master P_0 sends out chunks to workers over a network: we can think of a star-shaped network, with the master in the center. In order to concentrate on the difficulty introduced by the non-linearity of the cost, we consider the simplest communication model, where all communications between the master and the computing resources can take place in parallel, the speed of the communication between P_0 and P_i being only limited by the incoming bandwidth of node P_i. Unless stated otherwise, we assume that the computing platform is fully heterogeneous. The incoming bandwidth of processor is denoted $1/c_i$ (so that c_i is the time needed to send one unit of data to P_i) and its processing speed $s_i = 1/w_i$ (so that w_i is the time spent by P_i to process a unit of computation).

In the literature, several models have been considered. The master processor can distribute the chunks to the computing resources in a single round, (also called single installment in [9]), or send the chunks to the computing resources in multiple rounds: the communications will be shorter (less latency) and pipelined, and the workers will be able to compute the current chunk while receiving data for the next one. In both cases, a slave processor can start processing tasks only once it has received the whole data.

Similarly, in order to concentrate on the influence of non-linearity, return communications [28], [29], [30] will not be taken into account and we will consider the case of a single round of communications.

C. Outline

Thereby, there exists on the one side powerful theoretical models like DLT for scheduling jobs onto heterogeneous platforms, and on the other side powerful tools like MapReduce that allow to easily deploy applications on large scale distributed platforms. However, both are best suited to workloads with linear complexity: in this paper, we prove that divisible load theory cannot directly be applied to quadratic workloads in Section II, such as proposed in [31], [32], [33], [34], [35]. We precisely state the limits for classical DLT studies in Section III. Then, we review and propose solutions based on a careful preparation of the dataset, and study how this approach could be applied in the context of MapReduce operations in Section IV.

II. Non-linear workloads are not amenable to Divisible Load Theory

Recently, several papers [31], [32], [33], [34], [35] have studied the case on non-linear divisible scheduling. For instance, let us consider that a non-linear (say $N^\alpha,\ \alpha > 1$) cost operation is to be performed on a list of N elements. In order to analyze the impact of non-linearity and to assess the limits of the approach followed in these papers, we will concentrate in this section on fully homogeneous platforms, where all communication and computing resources have the same capabilities.

Each computing resource $P_i,\ 1 \leq i \leq P$ has a (relative) computing power $\frac{1}{w}$ and it is associated to a bandwidth $\frac{1}{c}$. Thus, it will take $c \cdot X$ time units to transmit X data units to P_i and $w \cdot X^\alpha$ to execute these X units of load on P_i. In this model, the direct translation of classical linear DLT problem to the nonlinear case DLT that is proposed in [31], [32] consists in finding the amount of load that should be allocated to each processor.

In general, complex platforms are considered, with fully heterogeneous computing and communication platforms, and some flavor of one-port model, what makes the resolution of above problem difficult and requires sophisticated techniques (see for instance the solutions proposed in [33], [34], [35]) in order to determine in which order processing resources should be fed by the master node and how much data to process they should be given. In our very simplistic homogeneous model, with parallel communications and no return communication of the results, the above problem becomes trivial: the ordering is not important since all the processors are identical and therefore, in the optimal strategy, each P_i receives $\frac{N}{P}$ data elements in time $\frac{N}{P}c$ and starts processing them immediately until time

$$\frac{N}{P}c + \left(\frac{N}{P}\right)^\alpha w.$$

In general [33], [34], [35], for more heterogeneous platforms and more sophisticated communication models, obtaining the optimal allocation and a closed formula is impossible. Nevertheless, the importance of this issue is not crucial. In fact, the main problem with this approach is that when P is large, the part of the computations that is processed during this phase is negligible with respect to the overall work to be processed. Indeed, the overall work $\mathcal{W}$ is given by $\mathcal{W} = N^\alpha$ and the overall work $\mathcal{W}_{\mathrm{P}artial}$ performed by all p processors during this first phase is given by

$$\mathcal{W}_{\mathrm{P}artial} = P\left(\frac{N}{P}\right)^\alpha = \frac{N^\alpha}{P^{\alpha-1}},$$

so that

$$\frac{\mathcal{W} - \mathcal{W}_{\mathrm{P}artial}}{\mathcal{W}} = 1 - \frac{1}{P^{\alpha-1}}$$

that tends toward 0 when P becomes large !

Therefore, the difficult optimization problem solved in [33], [34], [35] has in practice no influence on the overall computation time, since asymptotically (when P becomes large) all the work remains to be done after this first phase. Above results shows the intrinsic linear complexity of the problems that are divisible. Indeed, a divisible load can be arbitrarily split between any number of processors (say N) in small pieces (of size 1 in this case) that can be performed independently (in time $c + w$), so that no dependencies should exist between data and the overall complexity is necessarily linear in the size of the data ($(c+w)N$ in this setting).

Nevertheless, this does not mean that only linear cost complexity tasks only can be processed as divisible load tasks or using MapReduce. Indeed, the possibility remains (i) either to modify the initial data, such as proposed in [36], [37], [27] for matrix multiplication (in this case the initial N^2 size data is transformed into a N^3 size data by replicating matrix blocks before applying a MapReduce operation) (ii) or to decompose the overall operation using a long sequence of MapReduce operations, such as proposed in [25]. In the remaining of this paper, we will concentrate on the first approach, that consists in expressing a problem that contains data dependencies into a (larger) problem, where data dependencies have been removed.

III. Divisible Load Theory for almost linear workloads

A. Sorting with Homogeneous Computing Resources

As we have seen in the previous section, DLT cannot in general be applied to workloads with super-linear complexity N^α, $\alpha > 1$ unless the size of the initial data is increased or if several DLT operations are applied. However, there are some intermediate cases where the complexity is close to N and thus where the workload can be seen as almost divisible.

In this section, we consider the problem of sorting N numbers and we propose a parallel algorithm based on DLT to distribute this computation onto an homogeneous platform. Indeed, since the work required by sorting is $\mathcal{W} = N \log N$ and if the initial dataset is split into p lists of size $\frac{N}{p}$, the work produced when

sorting the p lists in parallel in the DLT phase is given by

$$\mathcal{W}_{\mathrm{P}artial} = p\left(\frac{N}{p}\right)\log\left(\frac{N}{p}\right) = N\log N - N\log p,$$

so that

$$\frac{\mathcal{W} - \mathcal{W}_{\mathrm{P}artial}}{\mathcal{W}} = \frac{\log p}{\log N},$$

that is arbitrarily close to 0 for large values of N. Therefore, contrarily to what happens is the case of tasks whose complexity is N^α, in the case of sorting, almost all the work can be expressed as a divisible load task and sorting is likely to be amenable to DLT.

Nevertheless, applying directly partial sorts to the p lists would not lead to a fully sorted result, and a preprocessing of the initial list is needed. More precisely, we rely on the sample sort introduced and analyzed in [38], [39]. The sample sort is a randomized sort, that relies on the use of a random number generator. The parallel running time is almost independent of the input distribution of keys and all complexity results will be given with high probability. The algorithm proceeds in three phases, depicted on Figure 1 (where red lists are unsorted and blue lists are sorted).

- **Step 1**: A set of $p-1$ splitter keys are picked and then sorted to partition the linear order of key values into p buckets.
- **Step 2**: Based on their values, the keys are sent to the appropriate bucket, where the ith bucket is stored in the ith processor.
- **Step 3**: The keys are sorted within each bucket (using 1 processor per bucket).

Clearly, due to randomization, the buckets do not typically have exactly equal size and oversampling is used to reduce the ratio between the size of the largest bucket and its expected size $\frac{N}{p}$. Using an oversampling ratio of s, a sample of sp keys are chosen at random, this sample is sorted, and then the $p-1$ splitters are selected by taking those keys in the sample that have ranks $s, 2s, 3s, ..., (p-1)s$.

Therefore, the time taken by **Step 1** (using oversampling) is $sp\log(sp)$, *i.e.* the necessary time to sort (on the master processor) the sample of size sp. The cost of **Step 2** (on the master processor) is given by $N\log p$, since it involves, for each element to perform a binary search in order to determine its bucket.

At last, let us determine the time taken by **Step 3**. First, let us apply Theorem B.4 proved in [40], with $\alpha = 1 + (1/\log N)^{\frac{1}{3}}$ and $s = \log^2 N$. In this case, if $MaxSize$ denotes the size of the largest bucket, then

$$\Pr\left(MaxSize \geq \frac{N}{p}\left(1 + (1/\log N)^{\frac{1}{3}}\right)\right) \leq N^{-\frac{1}{3}},$$

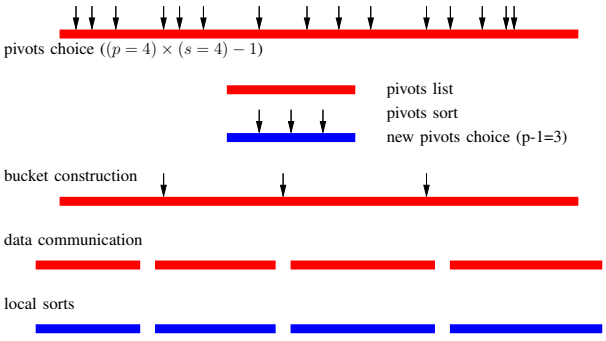


Figure 1. Sample sort with $p = 4$ processors and $s = 4$ oversampling ratio

i.e. the size of the largest bucket is of order $\frac{N}{p} + o(N)$ with high probability $1 - o\left(\frac{1}{N}\right)$.

The time taken by **Step 3** is given by $(MaxSize \log MaxSize)$ so that, with high probability, it is bounded by

$$\frac{N}{p}\log N + o\left(N\log N\right).$$

Therefore, if we choose $s = (\log N)^2$ as oversampling ratio, the time required by the preprocessing of **Step 1** and **Step 2** is dominated by $N\log p$ and the overall execution time is given by the time taken by **Step 3**, *i.e.* $\frac{N}{p}\log N + o(N\log N)$ and is therefore optimal on p processors with high probability.

Therefore, in the case of sorting, it is possible, by introducing a preprocessing phase on the initial data (but keeping the same data size), to reduce the high cost operation to a fully Divisible Load Task and therefore, in the case of sorting, optimizing the data distribution phase to slave processors under more complicated communication models that the one considered in this paper, is meaningful.

B. Generalization to Heterogeneous Processing Resources

Extending above result to heterogeneous computing resources is not difficult. Indeed, let us denote as previously by $\frac{1}{w_i}$ the processing power of computing resource P_i, $1 \leq i \leq P$, so that sorting N_i elements sequentially on P_i takes $w_i N_i \log N_i$. Then, after oversampling with ratio $s = (\log N)^2$, we assign to the ith bucket elements whose value is in the interval

$$\left[\left\lfloor \frac{\sum_0^{i-1} 1/w_k}{\sum_0^P 1/w_k}(P-1)\right\rfloor s, \left\lfloor \frac{\sum_0^{i} 1/w_k}{\sum_0^P 1/w_k}(P-1)\right\rfloor s - 1\right].$$

Therefore, processor P_i receives (see Theorem B.4 in [40], taking again $\alpha = 1 + (1/\log N)^{\frac{1}{3}}$ and $s =$

$\log^2 N$)) a list of at most $\frac{1/w_i}{\sum_0^P 1/w_k}$ elements (asymptotically when N becomes large) with high probability, and the load is well balanced with high probability between the processors (asymptotically when N becomes large).

Therefore, sorting is amenable to a divisible task at the price a preprocessing phase, even in presence of heterogeneous processors.

IV. Scheduling non-linear workloads on heterogeneous platforms

As shown in Section II, Divisible Load Theory cannot be used to schedule non-linear workloads, *i.e.* workloads with a complexity of order $O(N^\alpha)$, with $\alpha > 1$ for a size N of the data. For these workloads, dividing the computation into small chunks (whose size sums up to N) does not enable to perform enough work, and the same chunk of data is needed for several chunks of computations. For example, when multiplying two $N \times N$ matrices A and B, the same element $a_{i,j}$ needs to be multiplied to N elements of B, thus contributing to several chunks of computation.

One solution to overcome this problem is to introduce data redundancy, *i.e.* to replicate the data for each chunk of computation. For matrix multiplication, the $a_{i,j}$ element will be replicated for each computational chunk where $a_{i,j}$ is involved. Several distributions of the data are possible and have been implemented using MapReduce, such as row/column distribution or block distribution [27], [36]. Whatever the distribution, in MapReduce, the load-balancing is achieved by splitting the workloads in many tasks, which are then scattered across the platform. The fastest processor (or the one with smallest external load) gets more chunks than the others, so that all processors finish their share approximately at the same time (in Hadoop [26], some tasks are themselves replicated at the end of the computations to minimize execution discrepancy). Assuming large and fast communication capacities, this method produces an almost perfect load-balancing. However, because of data replication, it may lead to a large communication overhead. Thus, communication links may become bottleneck resources if the replication ratio is large.

In this section, we study how to reduce the communication overhead for non-linear workloads on heterogeneous platforms. Our method relies on understanding the underlying data structure and the structure of dependencies between tasks, and then proposing a well-suited distribution of the data. Our objective is to achieve a perfect load-balancing of the workload and simultaneously to minimize the amount of communications.

We first present in Section IV-A a data distribution scheme for a basic operation, the outer product of two vectors (whose complexity is of order N^2 and data size is of order N), before extending it in Section IV-B to the matrix multiplication (whose complexity is of order N^3 and data size is of order N^2). For each problem, we compare our method to classical block distributions used in MapReduce implementations, both theoretically and through simulations.

A. Outer-product: 2D data distribution

We consider the problem of computing the outer-product $a^T \times b$ of two (large) vectors a and b (of size N), what requires the computation of all $a_i \times b_j$, $\forall\ 1 \leq i, j \leq N$ (see Figure 2(a)). As stated above, we target an heterogeneous computing platforms and we denote by $s_i = 1/w_i$ the processing speed of P_i. Let us also introduce the normalized processing speed of P_i $x_i = s_i / \sum_k s_k$ so that $\sum_i x_i = 1$. At last, let us assume without loss of generality that processors are sorted by non-decreasing processing speed: $s_1 \leq s_2 \leq \cdots \leq s_p$.

Our objective is to minimize the overall amount of communications, *i.e.* the total amount of data send by the master initially holding the data (or equivalently by the set of devices holding it since we are interested in the overall volume only), under the constraint that a perfect load-balancing should be achieved among resources allocated to the outer product computation. Indeed, due to data dependencies, if we were to minimize communications without this load-balancing constraint, the optimal (and inefficient) solution would consist in making use of a single computing resource.

In what follows, we compare two approaches, **Homogeneous Blocks** and **Heterogeneous Blocks**.

1) **Homogeneous Blocks**: the first approach is based on a classical block distribution of the data, where the computational domain is first split into a (large) number of homogeneous chunks. Then, a demand driven model is used, where processors ask for new tasks as soon as they end processing one. With this approach, load is well balanced and faster processors get more blocks than slower one. This typically corresponds to the data distribution schemes used by MapReduce implementations.
2) **Heterogeneous Blocks**: The second approach consists in taking into account the heterogeneity of the processors when splitting the computational domain. Among the partitions where processors receive a fraction of the domain proportional to their speed, we search for the one that minimizes the amount of exchanged data.

1) Homogeneous Blocks approach: Let us first consider the **Homogeneous Blocks** approach, where the

computation domain is split into squares of size $D \times D$. Each chunk of computation consists in the outer-product of two chunks of data of size D from vectors a and b: $(a_i, \ldots, a_{i+D-1})^T \times (b_j, \ldots, b_{j+D})$. We choose to partition the computation domain into square blocks in order to minimize the communication costs: for a given computation size (D^2), the square is the shape that minimize the data size ($2D$). Heterogeneity is handled using dynamic load balancing: faster processors will get more blocks than slower ones. In order to minimize the communication cost, we choose the size of blocks so as to send a single chunk to the slowest processor. Since its relative processing speed is x_1, the size of the atomic block will be $D^2 = x_1 N^2$, so that $D = \sqrt{x_1}N$ (let us assume that N is large so that we can assume that this value is an integer). The total number of blocks is thus $\left(N/\sqrt{x_1}N\right)^2 = 1/x_1$. Using this atomic block size, if the demand driven scheme achieves perfect load balancing, processor P_i receives a number of blocks n_i proportional to its computing speed

$$n_i = \frac{x_i}{x_1} = \frac{s_i}{s_1}.$$

Again, let us assume for now that all these quantities are integer. The number of chunks distributed among processors is therefore $\sum_i x_i/x_1 = 1/x_1$, what ensures that all blocks are processed.

The total amount of communications $Comm_{hom}$ generated by the **Homogeneous Blocks** approach is the number of blocks times the input data for a block ($2D$).

$$Comm_{hom} = 1/x_1 \times 2N\sqrt{x_1} = 2N\sqrt{\frac{\sum_i s_i}{s_1}}.$$

2) Heterogeneous Blocks approach: The main drawback of the **Homogeneous Blocks** approach is clear in the context of strongly heterogeneous resources. When the ratio between the smallest and the largest computing speeds is large, the fastest processor will get a large number of (small) square chunks. For such a processor, the ratio between communications and computations is far from being optimal. Indeed, if these small square chunks could be grouped into a larger square chunk, data reuse would be better and the required volume of communications would be smaller.

With the **Heterogeneous Blocks** approach, a unique chunk is sent to each processor in order to avoid unnecessary data redundancy. We build upon a partitioning algorithm presented in [41]. In this paper, the problem of partitioning a 1×1 square into p rectangles of prescribed area $a_1, a_2, \ldots, a_p$ is addressed. Two different objectives related to the perimeter of the rectangles in the partition are considered: minimizing the maximum half-perimeter (PERI-MAX) or the sum of the half-perimeters (PERI-SUM).

Our objective is to partition the square computational domain of the $a_i \times b_j$ into p rectangles, whose area are proportional to their relative computation speed (so as to enforce an optimal load balancing), and to minimize the overall volume of communications. A processor will be given a chunk of k consecutive values of a $(a_i, \ldots, a_{i+k})$ and l values of b $(b_j, \ldots, a_{j+l})$. The amount of communication for this processor is given by $k+l$, *i.e.* the half-perimeter of its rectangular chunk. Moreover, chunks must partition the whole domain without overlap. Using scaled computational speeds $a_i = x_i$, our problem is equivalent to minimizing the sum of the half-perimeters when partitioning the unit square into rectangles of prescribed area, *i.e.* the PERI-SUM problem.

In [41], several column-based algorithms are proposed: the square domain is first split into columns that are then divided into rectangles. In particular, a column-based partitioning algorithm for the PERI-SUM problem is proven to have a guaranteed performance. The sum of the half-perimeters $\hat{C}$ given by the algorithm is such that

$$\hat{C} \leq 1 + \frac{5}{4} LBComm \quad \text{where} \quad LBComm = 2\sum_{i=1}^{p} \sqrt{a_i},$$

where $LBComm$ is a straightforward lower bound on the sum of the half-perimeters, which is larger than 2. Thus,

$$\hat{C} \leq \frac{7}{4} LBComm.$$

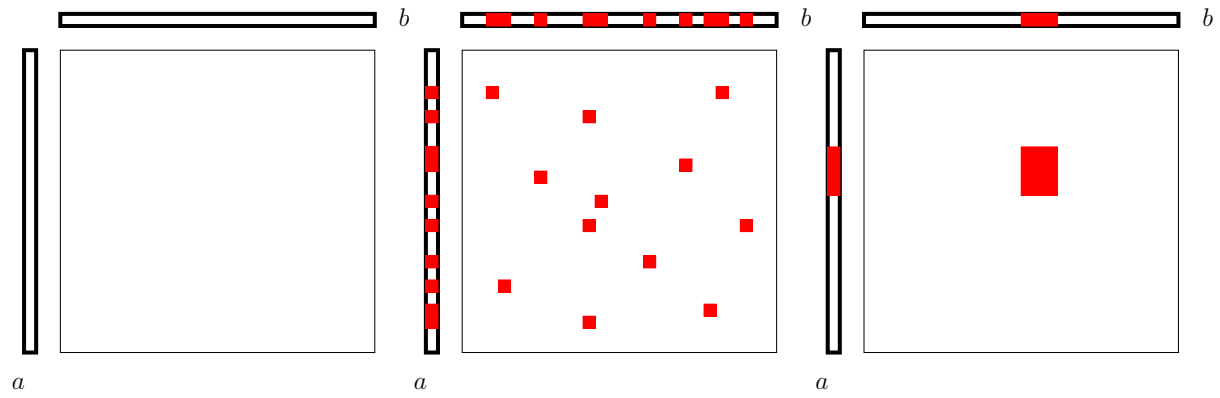
Note that as soon as there are enough processors, $LBComm \gg 2$ so that the approximation ratio gets asymptotically close to $5/4$.

Using this partitioning, scaled to the $N \times N$ computational domain, the total amount of communications $Comm_{het}$ can be bounded as

$$Comm_{het} \leq \frac{7N}{2} \sum_{i=1}^{p} \sqrt{x_i} = \frac{7N}{2} \frac{\sum_{i=1}^{p} \sqrt{s_i}}{\sqrt{\sum_{i=1}^{p} s_i}}.$$

3) Comparison of Block Homogeneous and Block Heterogeneous Approaches: Using previous analysis, we can bound the ratio ρ between the amounts of communication generated by both **Block Homogeneous** and **Block Heterogeneous** approaches.

$$\rho = \frac{Comm_{hom}}{Comm_{het}} \geq \frac{2N\sqrt{\frac{\sum_i s_i}{s_1}}}{\frac{7N}{2}\frac{\sum_{i=1}^{p}\sqrt{s_i}}{\sqrt{\sum_{i=1}^{p} s_i}}} = \frac{4}{7} \times \frac{\sum_i s_i}{\sqrt{s_1}\sum_i \sqrt{s_i}}.$$



(a) Initial data. (b) Using **Homogeneous Block** approach. (c) Using **Heterogeneous Block** approach.

Figure 2. Data sent to a given processor for both implementation of outer-product.

When the platform is fully homogeneous, *i.e.* when all processors have the same computation speed $s_i = s_1$, both approaches provide the same solution, and our analysis simply states that $\rho \geq 4/7$. However, the **Block Heterogeneous** approach is interesting as the platform goes heterogeneous.

A typical situation is depicted on Figure 2(b) and Figure 2(c), where a basic task corresponds to a small red matrix block. In the case of the **Homogeneous Block** approach, tasks are requested on demand and therefore, the memory footprint (data needed by a processor of relative speed 12) on a and b vectors will be high, whereas with the **Heterogeneous Block** approach, the volume of necessary data is highly reduced.

Consider for example the case where the first half of the platform is built from slow nodes (speed s_1) while the second one is built from nodes that are k times faster (speed ks_1). Then, our analysis shows that $\rho \geq \frac{1+k}{1+\sqrt{k}} \geq \sqrt{k} - 1$.

B. 3D data distribution: matrix multiplication

Let us start by briefly recalling the classical parallel matrix multiplication implementations. The whole set of computations can be seen as a 3D cube where element (i, k, j) corresponds to the basic operation $a_{i,k}b_{k,j}$. At the notable exception of recently introduced 2.5D schemes [42], all implementations (see [43] for a recent survey), including those implemented with MapReduce [36], [27] or designed for GPUs [44] are based on the ScaLAPACK algorithm [45], that uses the outer product described in Section IV-A as building block. For the sake of simplicity, we will concentrate on the case of square matrices only. In that case, all 3 matrices (A, B and $C = A \times B$) share the same layout, *i.e.* for all i, j, the same processor is responsible for storing $A_{i,j}, B_{i,j}$ and $C_{i,j}$. Then, at each step k, any processor that holds some part of the kth row of A broadcasts it to all the processors of its column and any processor that holds some part of the kth column of B broadcasts it to all the processors of its row (see Figure 3).

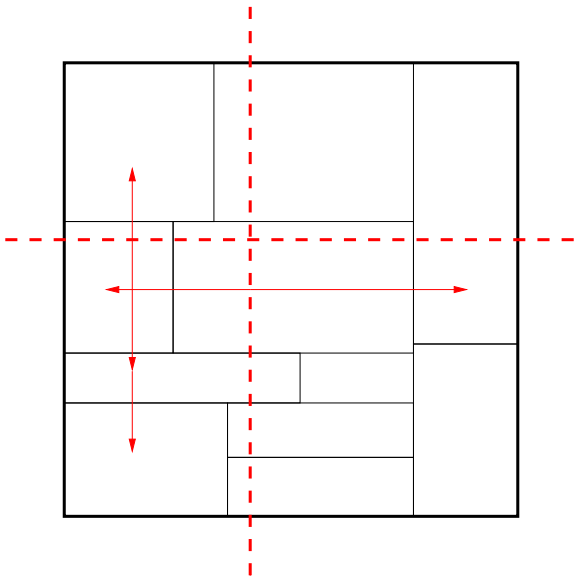


Figure 3. Matrix Multiplication algorithm based on the Outer Product

Of course, actual implementations of this algorithm use a blocked version of this scheme, and a level of virtualization is added. Indeed, since the number of blocks is much larger than the number of processors, blocks are scattered in a cyclic fashion along both grid dimensions, so that each processor is responsible for updating several blocks at each step of the algorithm.

Thus, the volume of communication induced by the matrix multiplication algorithm is exactly proportional to the sum of the perimeters of the rectangles assigned to the processors, and therefore, the ratio proved in Section IV-A is valid between heterogeneity aware implementations based on **Heterogeneous Block** distributions and MapReduce implementations [36], [27]. In Section IV-C, we experimentally analyze this ratio through extensive simulations corresponding to several heterogeneity profiles.

C. Experimental evaluation

To assess the quality of the two data distribution policies presented above, we present here an experimental evaluation based on simulations. Both policies are compared to the lower bound on the communication $LBComm$ introduced above: this bound corresponds to allocating to each processor a square whose area corresponds exactly to its relative processing speed, *i.e.* each processor P_i is given a $N\sqrt{x_i} \times N\sqrt{x_i}$ square. Considering that all obtained quantities are integer, the following lower bound holds for the total number of necessary communications $LBComm = 2N\sum_i \sqrt{x_i} = 2N\frac{\sum_i \sqrt{s_i}}{\sqrt{\sum_i s_i}}$.

In the **Block Homogeneous** strategy, we make a strong assumption: the number of chunks to give to processor P_i is given by s_i/s_1, that is supposed to be an integer. In fact, these numbers have to be rounded to integers, thus leading to a possibly prohibitive load imbalance. Therefore, we propose a more realistic strategy, that splits the chunks into smaller blocks in order to avoid a large load imbalance. Let us first define the load imbalance of a given load distribution as

$$e = \frac{t_{\max} - t_{\min}}{t_{\min}},$$

where $t_{\max}$ (respectively $t_{\min}$) is the largest (resp. smallest) computation time in the platform. In the $Comm_{hom}$ strategy, the chunk size is chosen as large as possible to avoid unnecessary data redundancy, what may lead to a large load imbalance. To avoid this, we introduce the $Comm_{hom/k}$ strategy, that divides the block-size by k for increasing values of k until an acceptable load-balance is reached. In our simulations, the stopping criterion for this process is when $e \leq 1\%$. This strategy is expected to lead to a larger amount communications, at the benefit of a better load balancing.

We perform simulations for a number of processors varying from 10 to 100. For each simulation, we generate the processing speeds using three different policies: the processing speeds either (i) are homogeneous, (ii) follow a uniform distribution in the range [1,100] or (iii) follow a log-normal distribution with parameters $\mu = 0$ and $\sigma = 1$. Figures 4(a), 4(b) and 4(c) present the results of these simulations. We compute the amount of communication induced by each strategy for a large matrix and we plot the ratio with the lower bound on communication. Each point represent the average ratio of a given strategy for 100 simulations with random parameters and error bars illustrate the standard deviation of this ratio.

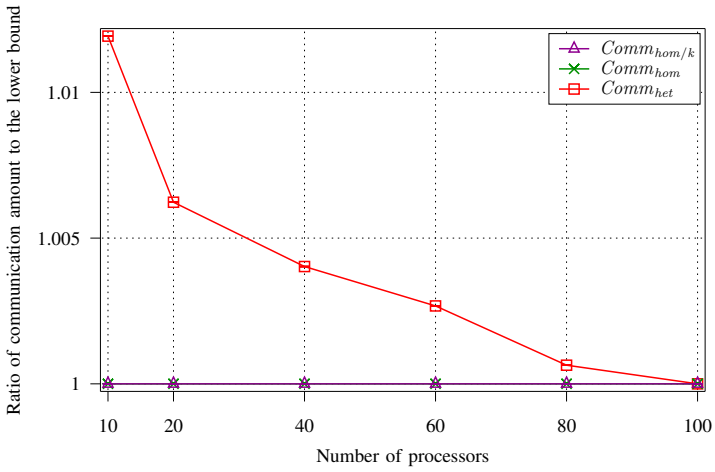
As expected, the Block Homogeneous $Comm_{hom}$ strategy performs very well in an homogeneous setting: each processor gets a square corresponding to its share, so that $Comm_{hom/k}$ does not increase the number of chunks. $Comm_{het}$ requires more communications, but the increase is usually as small as 1% of the lower bound. However, as soon as computing speeds get heterogeneous, $Comm_{hom}$ and $Comm_{hom/k}$ experience a large increase in the volume of communications: with 100 processors, the more realistic $Comm_{hom/k}$ strategy leads to a communication amount which is 15 to 30 times the lower bound (depending on the random distribution of computing speeds). On the contrary, the $Comm_{het}$ strategy never requires more than 2% more than the lower bound.

Interestingly, we can notice that $Comm_{het}$ is much better than its theoretical guarantee: in the previous analysis, we proved that it is a 7/4-approximation compared to the lower bound, but in all our experiments, it is always within 2% of this bound.

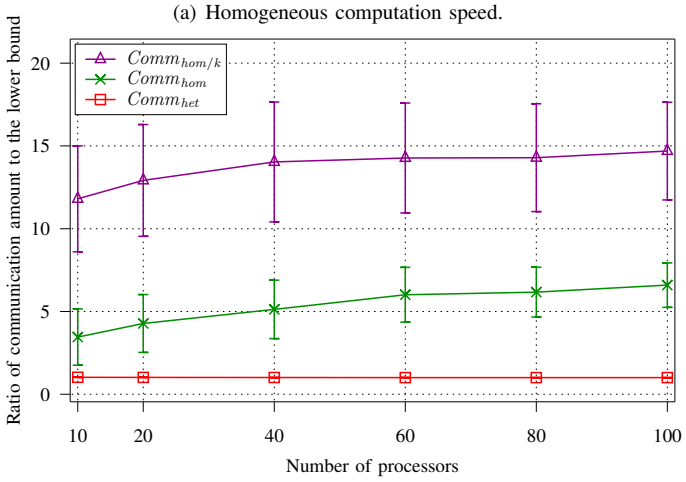
V. Conclusion

Recently, several papers have considered the case on non-linear divisible scheduling. We prove in this paper that tasks whose complexity is of order N^α, for $\alpha > 1$ cannot be considered as divisible tasks, except if data is first replicated. The same applies to MapReduce jobs for non linear complexity tasks, that require a specific preparation of the data set. We have considered and analyzed the case of matrix multiplication (or equivalently outer product). On the one hand, we prove that if all computing resources have the same capacities, then classical implementations achieve very good results, even for non linear complexity tasks.

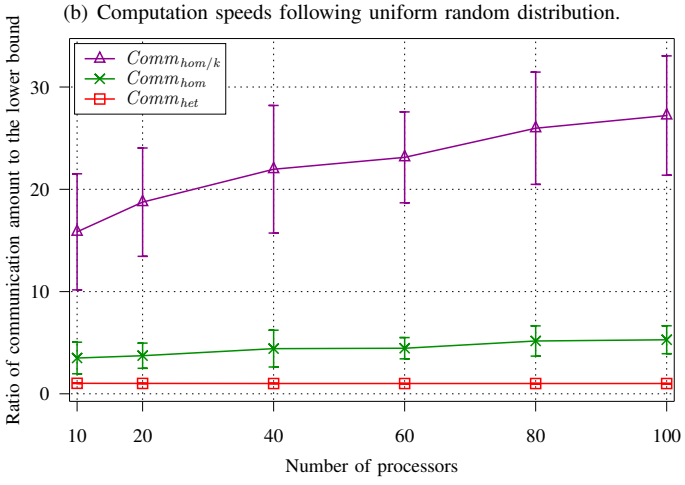
On the other hand, these experiments prove that in the context of heterogeneous computing resources, and when the complexity of the underlying task is not linear, taking explicitly heterogeneity into account when partitioning data and building tasks is crucial in order to minimize the overall communication volume, that can be easily reduced by a factor of 15 to 30. Nevertheless, at present time, by construction, MapReduce implementations are not aware of the execution platform and



(a) Homogeneous computation speed.



(b) Computation speeds following uniform random distribution.



(c) Computation speeds following log-normal random distribution.

cannot therefore use an adapted data layout. Without changing the programming framework, whose simplicity is essential, adding directives in order to declare affinities between tasks and data could be partially solve this problem. For instance, in the context of the outer product or the matrix multiplication, favoring among all available tasks on the master those that share blocks with data already stored on a slave processor in the demand-driven process would improve the results.

Therefore, we believe that this paper provides a sound theoretical background for studying data distribution and partitioning algorithms in the context of the execution of non-linear complexity tasks on heterogeneous platforms and opens many practical perspectives, by proving the interest of proposing new mechanisms in MapReduce to take into account affinities between tasks and data.

REFERENCES

[1] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[2] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "Datacutter: Middleware for filtering very large scientific datasets on archival storage systems," in *NASA conference publication*. NASA; 1998, 2000, pp. 119–134.

[3] S. Hary and F. Ozguner, "Precedence-constrained task allocation onto point-to-point networks for pipelined execution," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 8, pp. 838–851, 1999.

[4] Q. Wu and Y. Gu, "Supporting distributed application workflows in heterogeneous computing environments," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 3–10.

[5] Q. Wu, J. Gao, M. Zhu, N. Rao, J. Huang, and S. Iyengar, "On optimal resource utilization for distributed remote visualization," *IEEE Trans. Computers*, vol. 57, no. 1, pp. 55–68, 2008.

[6] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, Eds., *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[7] D. Hochbaum, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[8] L. Epstein and R. van Stee, "Online bin packing with resource augmentation." *Discrete Optimization*, vol. 4, no. 3-4, pp. 322–333, 2007.

[9] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[10] J. Sohn, T. Robertazzi, and S. Luryi, "Optimizing computing costs using divisible load analysis," *IEEE Transactions on parallel and distributed systems*, vol. 9, no. 3, pp. 225–234, Mar. 1998.

[11] M. Hamdi and C. Lee, "Dynamic load balancing of data parallel applications on a distributed network," in *9th International Conference on Supercomputing ICS'95*. ACM Press, 1995, pp. 170–179.

[12] D. Altilar and Y. Paker, "An optimal scheduling algorithm for parallel video processing," in *IEEE Int. Conference on Multimedia Computing and Systems*. IEEE Computer Society Press, 1998.

[13] ——, "Optimal scheduling algorithms for communication constrained parallel processing," in *Euro-Par 2002*, ser. LNCS 2400. Springer Verlag, 2002, pp. 197–206.

[14] M. Drozdowski, "Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems," Ph.D. dissertation, Poznan University of Technology, Poznan, Poland, 1998.

[15] J. Blazewicz, M. Drozdowski, and M. Markiewicz, "Divisible task scheduling - concept and verification," *Parallel Computing*, vol. 25, pp. 87–98, 1999.

[16] R. Wang, A. Krishnamurthy, R. Martin, T. Anderson, and D. Culler, "Modeling communication pipeline latency," in *Measurement and Modeling of Computer Systems (SIGMETRICS'98)*. ACM Press, 1998, pp. 22–32.

[17] I. Foster and C. Kesselman, Eds., *Computational Grids: Blueprint for a New Computing Infrastructure*, 2nd ed. M. Kaufman Publishers, Inc., 2003.

[18] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[19] W. Shih, S. Tseng, and C. Yang, "Performance study of parallel programming on cloud computing environments using mapreduce," in *Information Science and Applications (ICISA), 2010 International Conference on*. IEEE, 2010, pp. 1–8.

[20] G. Iyer, B. Veeravalli, and S. Krishnamoorthy, "On handling large-scale polynomial multiplications in compute cloud environments using divisible load paradigm," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 48, no. 1, pp. 820 –831, jan. 2012.

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

[22] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation*. Springer Verlag, 1999.

[23] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[24] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 29–42.

[25] J. Berlińska and M. Drozdowski, "Scheduling divisible mapreduce computations," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 450 – 459, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510002698

[26] T. White, *Hadoop: The definitive guide*. Yahoo Press, 2010.

[27] S. Seo, E. Yoon, J. Kim, S. Jin, J. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 721–726.

[28] O. Beaumont, L. Marchal, and Y. Robert, "Scheduling divisible loads with return messages on heterogeneous master-worker platforms," *High Performance Computing–HiPC 2005*, pp. 498–507, 2005.

[29] O. Beaumont, L. Marchal, V. Rehn, and Y. Robert, "Fifo scheduling of divisible loads with return messages under the one-port model," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 14–pp.

[30] O. Beaumont and A. Rosenberg, "Link-heterogeneity vs. node-heterogeneity in clusters," in *High Performance Computing (HiPC), 2010 International Conference on*. IEEE, 2010, pp. 1–8.

[31] J. T. Hung and T. Robertazzi, "Scheduling nonlinear computational loads," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 44, no. 3, pp. 1169 –1182, july 2008.

[32] ——, "Scheduling nonlinear computational loads," in *Conference on Information Sciences and Systems*, july 2003.

[33] S. Suresh, A. Khayat, H. Kim, and T. Robertazzi, "An analytical solution for scheduling nonlinear divisible loads for a single level tree network with a collective communication model," *IEEE Transactions on Systems, Man, and Cybernetics, vol. resubmitted after revision*, 2008.

[34] S. Suresh, H. Kim, C. Run, and T. Robertazzi, "Scheduling nonlinear divisible loads in a single level tree network," *The Journal of Supercomputing*, pp. 1–21, 2011.

[35] S. Suresh, C. Run, H. J. Kim, T. Robertazzi, and Y.-I. Kim, "Scheduling second-order computational load in master-slave paradigm," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 48, no. 1, pp. 780 –793, jan. 2012.

[36] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[37] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally *et al.*, "Sequoia: programming the memory hierarchy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 4–4.

[38] W. Frazer and A. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM)*, vol. 17, no. 3, pp. 496–507, 1970.

[39] H. Li and K. Sevcik, "Parallel sorting by over partitioning," in *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1994, pp. 46–56.

[40] G. Blelloch, C. Leiserson, B. Maggs, C. Plaxton, S. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine cm-2," in *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*. ACM, 1991, pp. 3–16.

[41] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Partitioning a square into rectangles: Np-completeness and approximation algorithms," *Algorithmica*, vol. 34, no. 3, pp. 217–239, 2002.

[42] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin / Heidelberg, 2011, vol. 6853, pp. 90–109.

[43] K. Goto and R. Van De Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Soft*, vol. 34, no. 3, p. 12, 2008.

[44] K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, and S. G. Sedukhin, "Multi-level optimization of matrix multiplication for GPU-equipped systems," *Procedia Computer Science*, vol. 4, no. 0, pp. 342 – 351, 2011.

[45] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK user's guide*. Siam Philadelphia, 1997, vol. 3.