Journal of Computer and System Sciences 77 (2011) 154–166

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

A bridging model for multi-core computing

Leslie G. Valiant [1]

School of Engineering and Applied Sciences, Harvard University, United States

ARTICLE INFO

Article history:
Received 14 May 2009
Received in revised form 23 December 2009
Available online 11 June 2010
Accepted 7 June 2010

Keywords:
Parallel algorithms
Multi-core
Bridging model
Bulk synchronous

ABSTRACT

Writing software for one parallel system is a feasible though arduous task. Reusing the substantial intellectual effort so expended for programming a second system has proved much more challenging. In sequential computing algorithms textbooks and portable software are resources that enable software systems to be written that are efficiently portable across changing hardware platforms. These resources are currently lacking in the area of multi-core architectures, where a programmer seeking high performance has no comparable opportunity to build on the intellectual efforts of others. In order to address this problem we propose a bridging model aimed at capturing the most basic resource parameters of multi-core architectures. We suggest that the considerable intellectual effort needed for designing efficient algorithms for such architectures may be most fruitfully expended in designing portable algorithms, once and for all, for such a bridging model. Portable algorithms would contain efficient designs for all reasonable combinations of the basic resource parameters and input sizes, and would form the basis for implementation or compilation for particular machines. Our Multi-BSP model is a multi-level model that has explicit parameters for processor numbers, memory/cache sizes, communication costs, and synchronization costs. The lowest level corresponds to shared memory or the PRAM, acknowledging the relevance of that model for whatever limitations on memory and processor numbers it may be efficacious to emulate it. We propose parameter-aware portable algorithms that run efficiently on all relevant architectures with any number of levels and any combination of parameters. For these algorithms we define a parameter-free notion of optimality. We show that for several fundamental problems, including standard matrix multiplication, the Fast Fourier Transform, and comparison sorting, there exist optimal portable algorithms in that sense, for *all* combinations of machine parameters. Thus some algorithmic generality and elegance can be found in this many parameter setting.

1. Introduction

Multi-core architectures, based on many processors and associated local caches or memories, are attractive devices given current technological possibilities, and known physical limitations. However, the designer of parallel algorithms for such machines has to face so many challenges that the road to their efficient exploitation is not clearly signposted. Among these challenges consider the following: First, the underlying computational substrate is much more intricate than it is for conventional sequential computing and hence the design effort is much more onerous. Second, the resulting algorithms have to compete with and outperform existing sequential algorithms that are often much better understood and highly

E-mail address: valiant@seas.harvard.edu.

[1] This work was supported in part by NSF-CCF-04-27129. A preliminary version appeared in Proc. 16th Annual European Symposium on Algorithms, September 15–17, 2008, Karlsruhe, Germany, Lecture Notes in Comput. Sci., vol. 5193, pp. 13–28, and had also been presented at SPAA, June 14–16, 2008.

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2010.06.012

optimized. Third, the ultimate reward of all this effort is limited, at best a speedup of a constant factor, essentially the number of processors. Fourth, machines differ, and therefore any speedups obtained for one machine may not translate to speedups on others, so that all the design effort may be substantially wasted. For all these reasons it is problematic how or whether efficient multi-core algorithms will be created and exploited in the foreseeable future. It is possible that even as these machines proliferate, their computational potential will be greatly underutilized.

We have argued previously that the general problem of parallel computing should be approached via two notions [37,18]. The first is the notion of, *portable parallel algorithms*, algorithms that are parameter-aware and run efficiently on machines with the widest range of values of these parameters. We suggest that the creation of such algorithms needs to be adopted as an important goal. The second notion is that such portable algorithms have to be expressed in terms of a *bridging model*, one that bridges in a performance-faithful manner what the hardware executes and what is in the mind of the software writer. In particular the algorithms need to be written in a language that can be compiled efficiently on to the bridging model. It is this bridging model that defines the necessary performance parameters for the parameter-aware software, and is a prerequisite for portable parallel algorithms to be possible.

The originally proposed bridging model for parallel computation was the BSP model [37]. Its main features are that: (i) it is a computational model in that it is unambiguous, (ii) it incorporates numerical parameters that are intended to reflect ultimately unevadable physical constraints, and (iii) it has barrier synchronization as its one nonlocal primitive, an operation that is powerful yet relatively easy to realize. We note that in such a parametrized model some alternative choices can also be made, and a variety of these have been explored in some detail [12,13,7,6,14].

In this paper we introduce the Multi-BSP model which extends BSP in two ways. First, it is a hierarchical model, with an arbitrary number of levels. It recognizes the physical realities of multiple memory and cache levels both within single chips as well as in multi-chip architectures. The aim is to model *all* levels of an architecture together, even possibly for whole datacenters. Second, at each level, Multi-BSP incorporates memory size as a further parameter. After all, it is the physical limitation on the amount of memory that can be accessed in a fixed amount of time from the physical location of a processor that creates the need for multiple levels.

The Multi-BSP model for *depth* d will be specified by $4d$ numerical parameters $(p_1, g_1, L_1, m_1)(p_2, g_2, L_2, m_2)$ $(p_3, g_3, L_3, m_3) \cdots (p_d, g_d, L_d, m_d)$. It is a depth d tree with memories/caches at the internal nodes and processors at the leaves. At each level the four parameters quantify, respectively, the number of subcomponents, the communication bandwidth, the synchronization cost, and the memory/cache size.

It may be thought that proliferating numerical parameters only further exponentiates the difficulty of designing parallel algorithms. The main observation of this paper is that this is not necessarily the case. In particular we show, by means mostly of well-known ideas, that for problems such as standard matrix multiplication, Fast Fourier Transform and comparison sorting, algorithms can be written that are optimal in a parameter-free sense, even in the presence of this plethora of parameters. Our purpose is to persuade that it is feasible and beneficial to write down the best algorithmic ideas we have in a standardized form that is compilable to run efficiently on arbitrary machines and guaranteed to be optimal in a specifiable sense.

In order to elucidate this striking phenomenon, we shall define a *parameter-free* notion of an *optimal Multi-BSP algorithm with respect to a given algorithm A* to mean the following: (i) It is optimal in parallel computation steps to multiplicative constant factors and in total computation steps to within additive lower order terms, (ii) it is optimal in parallel communication costs to constant multiplicative factors among Multi-BSP algorithms, and (iii) it is optimal in synchronization costs to within constant multiplicative factors among Multi-BSP algorithms. (We note that some of the algorithms we describe can be made optimal to within additive lower order terms also in parallel computation steps, but this is a complication that we do not pursue here. We also note that the actual lower bounds we shall derive for (ii) hold for more general distributed models also.)

Insisting on optimality to a factor of one in total computation time is a significant requirement and imposes a useful discipline, we believe. We tolerate multiplicative constant factors k_{comp}, k_{comm}, and k_{synch}, in the other three measures, but for each depth d we insist that these be *independent* of the p, g, L and m parameters. Identifying the best achievable combinations of these constant factors can be challenging if all combinations of parameters are to be anticipated, and is left for future research. The ultimate goal is $(k_{comp}, k_{comm}, k_{synch})$-optimal algorithms where the k's are close to one, but in this paper we will be satisfied with $(O(1), O(1), O(1))$-optimality.

There have existed several previous models that have substantial commonality with Multi-BSP. Using memory size as a fourth BSP parameter was proposed and investigated by Tiskin [36] and by McColl and Tiskin [28]. In a different direction a variety of hierarchical versions of BSP have been proposed such as the D-BSP of de la Torre and Kruskal [13], which has been further investigated by Bilardi et al. [6,8]. In [8] a network-oblivious result is proved in this communication hierarchical context that, like our analyses, allows for arbitrary parameters at each level. This is in analogy with cache-oblivious algorithms [15]. The D-BSP captures hierarchies in communication while Multi-BSP seeks additionally to capture hierarchies in the cache/memory system.

In a different direction, numerous models have been proposed for studying varieties of memory or cache hierarchies both in the sequential [1] and in the parallel [39] contexts. In Alpern et al. [4] a tree structure of memories akin to ours is defined. For such hierarchical models in both the sequential and parallel contexts authors have generally taken some uniform cost view of the various levels, rather than analyzing the effect of arbitrary parameters at each level. Savage [32,33]

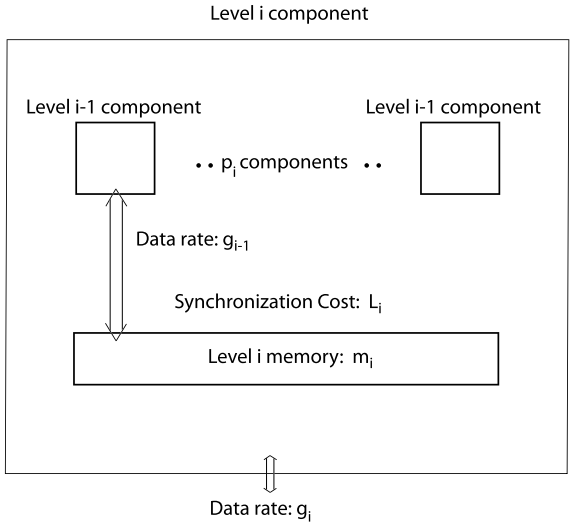

Fig. 1. Schematic diagram of a level i component of the Multi-BSP model in terms of level $i-1$ components.

has analyzed the communication requirements of a hierarchical memory model, with arbitrary parameters at each level, using a generalization of the Hong–Kung [23] pebble model.

Recently, also motivated by multi-core machines, a multi-level cache model has been proposed and analyzed by Blelloch et al. [9] and Chowdhury and Ramachandran [11]. In contrast with Multi-BSP, this model has a parameter for cache block size, but is less explicit on the issue of synchronization. The analyses published for this model have been for two levels with arbitrary parameters, but extensions are possible for arbitrary numbers of levels (Ramachandran [30]). A different model is described in [5]. Even more recently a multi-core model has been proposed by Savage and Zubair [34] that provides a precise model of the communication requirements in such architectures. There they independently give lower bounds for the communication requirements of FFT and matrix multiplication similar to these parts of our Theorem 1.

We emphasize that, in comparison with the previous literature, our goal here is that of finding a bridging model that isolates the most fundamental issues of multi-core computing and allows them to be usefully studied in some detail. The Multi-BSP model reflects the view that fundamentally there are just two unevadable sources of increasing cost that the physical world imposes at increasing distances: (i) a cost g for bandwidth, and (ii) a cost L related to latency that must be charged for synchronization and for messages that are too short. The model is a comprehensive model of computation in that it has mechanisms for synchronization as well as for computation and communication. The suggestion is that these unevadable costs already capture enough complications that we would be best advised to understand algorithmic issues in this *bridging* framework first.

The goal here is to identify a bridging model on which the community can agree, one which would influence the design of both software and hardware. It will always be possible to have performance models that reflect a particular architecture in greater detail than does any bridging model, but such models are not among our goals here.

There are many issues relevant to multi-core computing that we do not explore here. These include the use of multi-core for executing independent tasks, code automatically compiled from sequential code, and code compiled from languages in which parallelism is expressed but not scheduled. This paper is predicated on the idea that there will be a demand for exploiting multi-core architectures beyond what is possible by these means. A further issue not discussed here is the role of nonhomogeneous cores [22,27].

The main commonality between our algorithmic results and previous literature is the observation that certain recursive algorithms are well suited to computational models with multiple parameters. We push this observation further by allowing an arbitrary number of arbitrary parameters. Programming even simple recursive algorithms to run efficiently for all input sizes even on one machine can be an onerous task. Our finding is that for certain important problems, the use of a bridging model enables one to make one big effort, once and for all, to write a program that is efficient for all inputs and all machines.

2. The Multi-BSP model

An instance of a Multi-BSP is a tree structure of nested components where the lowest level or leaf components are processors and every other level contains some storage capacity. The model does not distinguish memory from cache as such, but does assume certain properties of it.

To define an instance of Multi-BSP we fix d, the *depth* or number of levels, and $4d$ further parameters (p_1, g_1, L_1, m_1) $(p_2, g_2, L_2, m_2)(p_3, g_3, L_3, m_3)\cdots(p_d, g_d, L_d, m_d)$. At the ith level there are a number of *components* specified by the parameters (p_i, g_i, L_i, m_i) each component containing a number of $i-1$st level components as illustrated in Fig. 1. In particular:

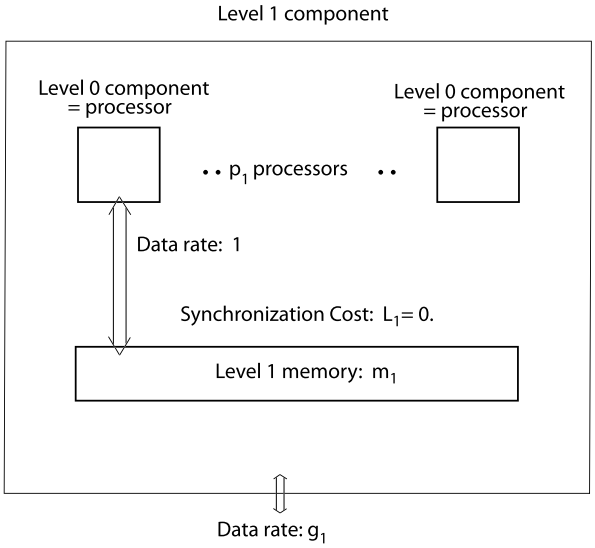

Fig. 2. Schematic diagram of a level 1 component of the Multi-BSP model.

(i) p_i is the number of $i-1$st level components inside an ith level component. For $i=1$ these 1st level components consist of p_1 raw processors, which can be regarded as the notional 0th level components. One computation step of such a processor on a word in level 1 memory is taken as one *basic unit* of time. (See Fig. 2.)

(ii) g_i, the communication bandwidth parameter, is the ratio of the number of operations that a processor can do in a second, to the number of words that can be transmitted in a second between the memories of a component at level i and its parent component at level $i+1$. A word here is the amount of data on which a processor operation is performed. Note that we shall assume here that the level 1 memories can keep up with the processors, and hence that the data rate (corresponding to the notional g_0) has value one.

(iii) A level i superstep is a construct within a level i component that allows each of its p_i level $i-1$ components to execute independently until they reach a barrier. When all p_i of these level $i-1$ components have reached the barrier, they can all exchange information with the m_i memory of the level i component with communication cost as determined by g_{i-1}. The communication cost charged will be mg_{i-1}, where m is the maximum number of words communicated between the memory of this ith level component and any one of its level $i-1$ subcomponents. This charge will be at most $2m_{i-1}g_{i-1}$, which we call the *gross* charge. The next level i superstep can then start. L_i is the cost charged for this barrier synchronization for a level i superstep. Note that the definition requires barrier synchronization of the subcomponents of a component, but no synchronization across branches that are more separated in the component hierarchy. (Note also that in this paper we use $L_1=0$, since the subcomponents of a level 1 component have no memories and directly read from and write to the level 1 memory.)

(iv) m_i is the number of words of memory inside an ith level component that is not inside any $i-1$st level component.

Finally we have to specify the nature of the communication between a level i component and the level $i+1$ component of which it is a part. The question is whether concurrent reading or writing (or some other combining operation) is allowed in either direction. The algorithms in this paper are all exclusive read and exclusive write (EREW), while the lower bounds hold for the strongest concurrent (CRCW) version.

We note that the parameters of the model imply values for certain other useful measures. The number of processors in a level i component will be $P_i = p_1 \cdots p_i$. The number of level i components in a level j component will be $Q_{i,j} = p_{i+1} \cdots p_j$, and the number in the whole system will be $Q_{i,d} = Q_i = p_{i+1} \cdots p_d$. The total memory available within a level i component will be $M_i = m_i + p_i m_{i-1} + p_{i-1}p_i m_{i-2} + \cdots + p_2 \cdots p_{i-1}p_i m_1$. The gap or bandwidth parameter that characterizes the cost of communication from level 1 to outside level i is $G_i = g_i + g_{i-1} + g_{i-2} + \cdots + g_1$.

Since the intention is to model the entire system, defining as many levels as necessary, we assume by convention that $Q_d = 1$ and that g_d is infinite. In other words the total machine is regarded as a single component, and includes any "external storage" to which communication is to be analyzed. For the same reason it is assumed that for any problem instance of size n and an algorithm for it, the level d memory is in fact sufficient to support the computation, and certainly $m_d \geqslant n$. In the applications in this paper $m_d = O(n)$ is sufficient.

We make the assumption that for all i

$$m_i \geqslant m_{i-1} \tag{1}$$

in order to simplify certain analyses. Also, we sometimes invoke the assumption that for all i

$$m_i \geqslant M_i/i. \tag{2}$$

Both of these hold for conventional cache hierarchies where any word at one level has distinct copies at every higher level. (We note that in the treatment here we do not otherwise distinguish between memory and cache.) Finally, we note that a major complication in the analysis of our algorithms arises from allowing arbitrary variations in the successive m_i. For our sorting algorithm it will be convenient to impose the mild constraint that for some constant k for all i

$$\log_2 m_i \leqslant (\log_2 m_{i-1})^k. \tag{3}$$

As far as relationships to other models, the $d = 1$ case with $(p_1 = 1,\ g_1 = \infty,\ L_1 = 0,\ m_1)$ gives the von Neumann model, and with $(p_1 \geqslant 1,\ g_1 = \infty,\ L_1 = 0,\ m_1)$ gives the more general PRAM [16,26,38] model. In both instances m_1 is the size of the memory.

The BSP model [37] with parameters (p, g, L) where the basic unit has memory m would be modeled with $d = 2$ and $(p_1 = 1,\ g_1 = g,\ L_1 = 0,\ m_1 = m)(p_2 = p,\ g_2 = \infty,\ L_2 = L,\ m_2)$. The difference is that in the basic BSP model direct communication is allowed horizontally between units at the same level, while in Multi-BSP such communication would need to be simulated via memory at a higher level. This case $(p_1 = 1,\ g_1 = g,\ L_1 = 0,\ m_1 = m)(p_2 = p,\ g_2 = \infty,\ L_2 = L,\ m_2)$ corresponds to the BSPRAM model of Tiskin [36].

In general, we regard $m = \min\{m_i \mid 1 \leqslant i \leqslant d\}$ and the input size n to be large numbers. In relating resource bounds F_1, F_2 expressed in terms of the parameters $\{p_i, g_i, L_i, m_i \mid 1 \leqslant i \leqslant d\}$ and the input size n, we shall define the relation $F_1 \precsim F_2$ to mean that for all $\varepsilon > 0$, $F_1 < (1+\varepsilon)F_2$ for all sufficiently large values of m and n. This enables expressions such as $(1 + 1/m_i)$, $(1 + 1/m_i^{1/2})$ or $(1 + 1/\log m_i)$ to be approximately upper bounded by 1.

Also, we define $F_1 \precsim_d F_2$ to mean that for some constant c_d depending possibly on d but not on n or on any of the parameters $\{p_i, g_i, L_i, m_i \mid 1 \leqslant i \leqslant d\}$ it is the case that $F_1 < c_d F_2$ for all sufficiently large values of n and m.

Because we can suppress constant multipliers with these notations, we shall sometimes identify a parameter such as m_j, for example, with a fixed multiple of itself.

For a Multi-BSP algorithm A^* we shall define $\mathrm{Comp}(A^*)$, $\mathrm{Comm}(A^*)$, and $\mathrm{Synch}(A^*)$ to be the *parallel* costs of computation, communication, and synchronization respectively on a Multi-BSP machine H in the sense that for any computation of A^* on H and along any single critical path in it, at most $\mathrm{Comp}(A^*)$ processor operations have been executed and at most $\mathrm{Comm}(A^*)$ communication charge and at most $\mathrm{Synch}(A^*)$ synchronization charge has been incurred. (For randomized algorithms the same claim holds with high probability.) Note that all three charges are expressed in terms of the basic unit of time taken by a processor to perform one operation.

To quantify the efficiency of A^* we specify a *baseline* algorithm A of which A^* is the Multi-BSP implementation and for that:

(i) $\mathrm{Comp}_{seq}(A)$ is the total number of computational operations of A. $\mathrm{Comp}(A)$ is $\mathrm{Comp}_{seq}(A)/P_d$ where P_d is the total number of processors in H.
(ii) $\mathrm{Comm}(A)$ is the minimal communication cost of any Multi-BSP implementation of A on H.
(iii) $\mathrm{Synch}(A)$ is the minimal synchronization cost of any Multi-BSP implementation of A on H.

A Multi-BSP algorithm A^* is *optimal with respect to algorithm A* if

(i) $\mathrm{Comp}(A^*) \precsim_d \mathrm{Comp}(A)$, and $\mathrm{Comp}_{seq}(A^*) \precsim \mathrm{Comp}_{seq}(A)$,
(ii) $\mathrm{Comm}(A^*) \precsim_d \mathrm{Comm}(A)$, and
(iii) $\mathrm{Synch}(A^*) \precsim_d \mathrm{Synch}(A)$.

The philosophy of the above definitions is the following: First, we are not specific about the class of algorithms to which A belongs, as long as the specified costs can be defined. For example, we variously allow A to be any implementation of the standard matrix multiplication algorithm, or any comparison algorithm for sorting. Second, allowing at each level some efficiency loss in communication and synchronization is tolerable for problems for which computational costs dominate asymptotically. It frees the analysis of several concerns, such as whether the input size is an exact multiple of the memory sizes. Analogous to the role of the polynomial time criterion in sequential computing, we believe that freeing the algorithm designer from the tedium of certain well-chosen optimality criteria will encourage the development of practical algorithms. In this instance we permit constant factor inefficiencies in the parallel computation, communication and synchronization costs. However, for all machines of the same depth d these constant factors are bounded by *the same constants independent of all the other machine parameters.* In all three measures additive lower order terms that have a vanishing relative contribution as the input size n and $m = \min\{m_i \mid 1 \leqslant i \leqslant d\}$ grow, are also allowed.

3. Relationships with architecture

While we advocate that any proposed bridging model be faithful to physical realities in terms of numerical parameters, we also believe that there is room for architects to design systems that are faithful to any agreed bridging model. In the latter spirit, the BSP model has been emulated by several software systems, including BSPlib [21], the Paderborn University BSP library [10], and the bulk synchronous graphics processor system BSGP [24], and it has been shown also (e.g. [19,20])

that such systems running on certain parallel machines do actually emulate the model faithfully in the desired quantitative sense.

In relating Multi-BSP to existing multi-core designs we can start with an attempt to relate the Multi-BSP parameters to those reported in the technical specifications of existing designs. For example, consider a parallel machine consisting of p Sun Niagara UltraSparc T1 multi-core chips connected to an external storage device that is large enough to store the input to the problem at hand. The parameters of the chip according to one interpretation of the specifications and modulo the serious qualifications listed below, would be the following:

Level 1: 1 core has 1 processor with 4 threads plus L1 cache: ($p_1 = 4,\ g_1 = 1,\ L_1^* = 3,\ m_1 = 8$ KB).

Level 2: 1 chip has 8 cores plus L2 cache: ($p_2 = 8,\ g_2 = 3,\ L_2^* = 23,\ m_2 = 3$ MB).

Level 3: p multi-core chips with external memory m_3 accessible via a network at rate g_2: ($p_3 = p,\ g_3 = \infty,\ L_3^* = 108,\ m_3 \leqslant 128$ GB).

Now the qualifications include the following: First, the L^*-parameters listed are certain latency parameters given in the chip specifications, rather than the actual synchronization costs. Second, the caches on the chip are controlled by implicit protocols, rather than explicitly by the programs as is customary for memories. Third, in the actual chip each physical processor runs four threads, and groups of processors share a common arithmetic unit. For all these reasons, while the relative magnitudes of the various g and L^* values shown may be meaningful, their absolute values for this instance are harder to pin down.

We have three main observations about the requirements on architectures to support Multi-BSP.

The first is that barrier synchronization needs to be supported efficiently. There are reports in the literature suggesting that this can be done already with current architectures [31].

The second is that the model controls the storage explicitly. It is not clear how the various cache protocols currently used are supportive of the model.

The third is that we only need that machines support the features of this model – no constraint is implied about what else they might support. Thus there is no objection to machines being able to switch off the barrier synchronization mechanism, or having as additional mechanisms some of those that have been advocated as alternatives in other related models. Our proposal here concerns the bridging model – a minimal description on which architects and algorithm designers can come together.

4. Work-limited algorithms

Our proofs of optimality for communication and synchronization given in this section and the one to follow all derive from lower bounds on the number of communication steps required in distributed algorithms and are direct applications of previous work, particularly of Hong and Kung [23], Aggarwal and Vitter [3], Savage [32,33] and Irony, Toledo and Tiskin [25].

Definition. A straight-line program A is $w(S)$-*limited* if every subset of its operations that uses at most S inputs (i.e. excluding data items generated by the subset) and produces at most S outputs (i.e. items of data that are used by operations of the program outside the subset, or are the outputs of the program) consists of no more than $w(S)$ operations.

Note that this limitation to S items imposes the twin constraints that at most S items can be used as data by the subset of the operations, and at most S items can be used to pass values computed by that subset to later computation steps. (As a minor comment we note also that our definition is slightly less onerous than those of red/blue pebbling and span [23,32], which, in addition, restrict to S the space that can be used internally by the algorithm fragment.)

We first consider the task of *associative composition* AC(n): Given a linear array A of n elements from a set X, an associative binary operation $\otimes$ on X, and a specific set of disjoint contiguous subarrays of A, the object is to compute the composition of each subarray under $\otimes$ in some order, where the only operation allowed on elements of X is $\otimes$.

Proposition 4.1. *For any n and S, any algorithm for associative composition* AC(n) *is* $(S-1)$*-limited.*

Proof. On S elements from X at most $S-1$ operations are to be performed. □

Next we consider the problem MM($n \times n$) of multiplying two $n \times n$ matrices by the standard algorithm, where the additions can be performed in any order.

Proposition 4.2. *For any n and S, the standard matrix multiplication algorithm* MM($n \times n$) *is* $S^{3/2}$*-limited.*

Proof. This is observed by Irony, Toledo and Tiskin [25, Lemma 2.2] by applying the Loomis–Whitney inequality to matrix multiplication as originally suggested by M. Paterson, and is also implicit in [23,32]. □

Next we consider FFT(n) the standard binary recursive algorithm for computing the one-dimensional Fast Fourier Transform on n points where n is a power of two.

Proposition 4.3. *For any n and S, the standard Fast Fourier Transform algorithm* FFT(n) *is* $2S\log_2 S$*-limited.*

Proof. This has been shown by Aggarwal and Vitter [3, Lemma 6.1]. □

Finally we shall consider Sort(n) the problem of comparison sorting. This is defined as one in which a given set X of n elements from an ordered set is to be sorted, such that the only operations allowed on members of X are pairwise comparisons. As the computation time we shall count only these comparisons as operations. Note, however, that comparison algorithms are not restricted to be straight-line programs.

5. Lower bounds

Our lower bound results for straight-line programs we derive using the approach of Irony, Toledo and Tiskin [25] (and also of [23,32]), while the result for sorting uses an adversarial argument of Aggarwal and Vitter [3]. The bounds will be stated for Multi-BSP but the lower bound arguments for communication hold more generally, for all distributed algorithms with the same hierarchy of memory sizes and costs of communication, even if there is no bulk synchronization.

Lemma 5.1. *Suppose W computation steps are executed of a $w(S)$-limited straight-line program on a Multi-BSP machine. Then for any j the total number of words transmitted between level j components and the level $j+1$ components to which they belong is at least*

$$M_j\big(W/w(2M_j) - Q_j\big), \tag{4}$$

and the total number of level j component supersteps at least

$$W/w(M_j). \tag{5}$$

Proof. For each level j component divide the computation into phases, where each phase ends when the total number of messages sent to or received from level $j+1$ reaches M_j. In each phase therefore at most $2M_j$ words are available, including those residing in memory before the start of the phase. Then at most $w(2M_j)$ operations can be performed by any execution sequence in one phase. It follows that the total number of such component phases is at least $W/w(2M_j)$. Further, each of these component phases, except possibly the last one for each component, must complete, and involve a movement of M_j data for each of the Q_j such components. Hence the total amount of data movement between level j and level $j+1$ is at least as claimed in (4).

By the same argument, since in a component superstep at most M_j memory is available, at most $w(M_j)$ operations can be performed, and hence at least $W/w(M_j)$ component supersteps are needed, which gives (5). □

Theorem 5.1. *Suppose $W(n)$ operations are to be performed of a $w(m)$-limited straight-line program A on input size n on a depth d Multi-BSP machine. Then the communication cost over the whole machine is at least*

$$\mathrm{Comm}(n,d) \gtrsim_d \sum_{i=1\ldots d-1} \big(W(n)/\big(Q_i w(2M_i)\big) - 1\big)M_i g_i, \tag{6}$$

and the synchronization cost at least

$$\mathrm{Synch}(n,d) \gtrsim_d \sum_{i=1\ldots d-1} W(n)L_{i+1}/\big(Q_i w(M_i)\big). \tag{7}$$

Proof. This follows from Lemma 5.1 by adding the costs over all the levels. Consider the Q_1 paths from the level 1 components to the level d component in the tree hierarchy as potential critical paths of the executions. The average load on these, and hence the worst case also, is as claimed in (6) and (7). □

Corollary 5.1.

$$\text{AC-Comm}(n,d) \gtrsim_d \sum_{i=1\ldots d-1} \big(n/(M_i Q_i) - 1\big)M_i g_i, \tag{8}$$

$$\text{AC-Synch}(n,d) \gtrsim_d \sum_{i=1\ldots d-1} nL_{i+1}/(Q_i M_i), \tag{9}$$

$$\text{MM-Comm}(n\times n,d) \gtrsim_d \sum_{i=1\ldots d-1} \big(n^3/\big(Q_i M_i^{3/2}\big) - 1\big)M_i g_i, \tag{10}$$

$$\text{MM-Synch}(n \times n, d) \succsim_d \sum_{i=1\ldots d-1} n^3 L_{i+1}/\big(Q_i M_i^{3/2}\big), \tag{11}$$

$$\text{FFT-Comm}(n, d) \succsim_d \sum_{i=1\ldots d-1} \big(n \log n/(Q_i M_i \log M_i) - 1\big) M_i g_i, \tag{12}$$

$$\text{FFT-Synch}(n, d) \succsim_d \sum_{i=1\ldots d-1} n \log n L_{i+1}/(Q_i M_i \log M_i), \tag{13}$$

$$\text{Sort-Comm}(n, d) \succsim_d \sum_{i=1\ldots d-1} \big(n \log n/(Q_i M_i \log M_i) - 1\big) M_i g_i, \tag{14}$$

$$\text{Sort-Synch}(n, d) \succsim_d \sum_{i=1\ldots d-1} n \log n L_{i+1}/(Q_i M_i \log M_i). \tag{15}$$

Proof. Applying Theorem 5.1 directly gives the first six inequalities.

The bounds for sorting follow from the following adversarial argument adapted from Aggarwal and Vitter [3]: Let $S = M_i$. Consider any total ordering of all the level i component supersteps that respects all the time dependencies. As we go through these component supersteps in that order we shall adversarially define a sequence of successively stricter partial orders on the input set X as may be discovered by the comparisons made in the successive component supersteps. In each such component superstep that inputs a set Y of r elements of X we shall identify a total ordering on $Y \cup Z$, where Z is the subset of X that was present in the component at the start of that component superstep. The total order identified for $Y \cup Z$ will be one that maximizes the number of partial orders on X still consistent with both this total order as well as the total orders found on subsets in previous stages of this adversarial process. Since the total order on Z was found in an earlier component superstep, the number of choices available is no more than S^r, which upper bounds the number of ways Y can be inserted into a sorted Z. Hence the logarithm to the base S of the number of partial orders on X still consistent after this component superstep is reduced by at most r, at the cost of r inputs, if we pick the total order on $Y \cup Z$ that is consistent with the most partial orders on X. Since this logarithm initially is $n \log_S n$, the total number of inputs must be at least this number, minus the total number of elements inside the level i components at the start, which is SQ_i. This establishes the lower bound on communication. For the final synchronization bound we observe that any component superstep can depend on at most S input elements. □

6. Optimal algorithms

We shall describe algorithms that meet the lower bounds on communication and synchronization given in the previous section but with M_j replaced by m_j. Under the assumption that (2) holds, M_j can be replaced by m_j in lower bounds and hence our upper and lower bounds then match. For each of the algorithms described it is easy to verify that the conditions $\text{Comp}(A^*) \precsim_d \text{Comp}(A)$ and $\text{Comp}_{seq}(A^*) \precsim \text{Comp}_{seq}(A)$ on computation steps are satisfied, and we shall not comment on them further.

A *level j component superstep* will refer to what a single level j component performs in a level j superstep. For simplicity of description we assume throughout that every fraction referenced, such as m_j/m_{j-1}, has integral value.

6.1. Associative composition

For AC(n) consider the recursive process where each level j component contains a set Z of contiguous subsequences of total length m_j, distributes the task of performing the required compositions of subsequences of length m_{j-1} of those sequences to its p_j subcomponents, and when it receives the m_j/m_{j-1} results back, it performs up to m_j/m_{j-1} further pairwise $\otimes$ operations recursively so as to derive the composition of the subsequences Z assigned to it.

The costs of the recursion at *one* level j component can be divided into (i) the data movement steps between the level j component and its level $j-1$ subcomponents, and (ii) the recursive computation of the m_j/m_{j-1} further pairwise $\otimes$ operations. For (i) since in the overall computation a level $j-1$ memory has to be filled with information from level j (and one word returned) at most m_j/m_{j-1} times, the parallel cost of communication at this level is at most

$$\big(m_j/(p_j m_{j-1})\big)(m_{j-1} + 1) g_{j-1} \precsim m_j g_{j-1}/p_j$$

and the total cost of synchronization at level j is

$$\precsim m_j L_j/(p_j m_{j-1}).$$

In addition $m_j/(p_j m_{j-1})$ component supersteps of AC($m_{j-1}, j-1$) will need to be executed.

For (ii) we observe that the cost corresponds to the original problem for a level j component superstep, but for input length m_j/m_{j-1} rather than m_j. In other words its costs are AC-Comp($m_j/m_{j-1}, j$), AC-Comm($m_j/m_{j-1}, j$) and AC-Synch($m_j/m_{j-1}, j$).

Hence, for the overall algorithm

$$\text{AC-Comm}(m_j, j) \precsim m_j g_{j-1}/p_j + \big(m_j/(p_j m_{j-1})\big)\,\text{AC-Comm}(m_{j-1}, j-1) + \text{AC-Comm}(m_j/m_{j-1}, j)$$

and

$$\text{AC-Synch}(m_j, j) \precsim m_j L_j/(p_j m_{j-1}) + \big(m_j/(p_j m_{j-1})\big)\,\text{AC-Synch}(m_{j-1}, j-1) + \text{AC-Synch}(m_j/m_{j-1}, j).$$

Consider the first of these equations. Note that this expression for AC-Comm(m_j, j) could have been derived and holds for any $m \leqslant m_j$. We can therefore expand this expression recursively by substituting it repeatedly for the last term AC-Comm$(m_j/m_{j-1}, j)$, first with $m = m_j/m_{j-1}$, then with $m = m_j/(m_{j-1})^2$, etc., to obtain

$$\begin{aligned}\text{AC-Comm}(m_j, j) &\precsim \big(m_j g_{j-1}/p_j + \big(m_j/(p_j m_{j-1})\big)\,\text{AC-Comm}(m_{j-1}, j-1)\big)\big(1 + 1/m_{j-1} + 1/m_{j-1}^2 \cdots\big)\\ &\precsim_d \big(m_j g_{j-1}/p_j + \big(m_j/(p_j m_{j-1})\big)\,\text{AC-Comm}(m_{j-1}, j-1)\big).\end{aligned}$$

Since we can equate the input size n with m_d, it follows by induction on j that

$$\text{AC-Comm}(n, d) \precsim_d \sum_{i=1\ldots d-1} n g_i/Q_i. \tag{16}$$

Expanding the second recurrence in exactly the same way gives

$$\text{AC-Synch}(n, d) \precsim_d \sum_{i=1\ldots d-1} n L_{i+1}/(Q_i m_i). \tag{17}$$

6.2. Matrix multiplication

For matrix multiplication $w(m) = O(m^{3/2})$ by Proposition 4.2. In a level j component superstep it is optimal to within constant factors therefore to do $w(3m)$ operations per component having total memory $3m$. This can be realized by inputting an $m^{1/2} \times m^{1/2}$ submatrix of each of A and B, computing the product of these submatrices, and outputting the m results as additive contributions to each of m entries in the corresponding $m^{1/2} \times m^{1/2}$ submatrix of $C = AB$. This follows the algorithms described in [2,28].

Hence at level 1 a component superstep consists of an $m_1^{1/2} \times m_1^{1/2}$ matrix multiplication. (Here and in the remainder of this section we shall suppress constant multipliers where it does not affect the conclusion.) Overall one will need $n^3/m_1^{3/2}$ such component supersteps, and hence $n^3/(Q_1 m_1^{3/2})$ parallel stages of such level 1 component supersteps, where Q_1 is the total number of level 1 components.

In general, a level j component superstep consists (within multiplicative constant factors) of an $m_j^{1/2} \times m_j^{1/2}$ matrix multiplication. Overall one will need $n^3/m_j^{3/2}$ such component supersteps, and hence $n^3/(Q_j m_j^{3/2})$ parallel stages of such supersteps.

In a level j component superstep there will be $m_j^{3/2}/(m_{j-1}^{3/2})$ level $j-1$ component supersteps of $m_{j-1}^{1/2} \times m_{j-1}^{1/2}$ matrix multiplications. In addition we will charge to this level the further $m_j(m_j^{1/2}/m_{j-1}^{1/2}) = m_j^{3/2}/m_{j-1}^{1/2}$ additions needed to combine the results of the $m_{j-1}^{1/2} \times m_{j-1}^{1/2}$ multiplications of the level $j-1$ local supersteps. For the latter operations we will use $m_j^{1/2}/m_{j-1}^{1/2}$ successive Associative Composition operations AC(m_j, j) we analyzed earlier, each such operation performing compositions on sets of $m_j^{1/2} m_{j-1}^{1/2}$ subarrays each of size $m_j^{1/2}/m_{j-1}^{1/2}$. Hence using (16) and $Q_j \geqslant 1$, the total communication cost we charge that derives from this jth level of recursion is

$$\begin{aligned}&\big(n^3/\big(m_j^{3/2} Q_j\big)\big)\big(g_j m_j + \big(m_j^{1/2}/m_{j-1}^{1/2}\big)\,\text{AC-Comm}(m_j, j)\big)\\ &\quad\precsim_d \big(n^3/\big(m_j^{3/2} Q_j\big)\big)\Big(g_j m_j + \big(m_j^{3/2}/m_{j-1}^{1/2}\big)\sum_{i=1\ldots j-1} g_i/Q_i\Big)\\ &\quad\precsim_d n^3 g_j/\big(m_j^{1/2} Q_j\big) + \big(n^3/m_{j-1}^{1/2}\big)\sum_{i=1\ldots j-1} g_i/Q_i.\end{aligned}$$

Hence adding over all levels $j = 1, \ldots, d-1$ gives

$$\begin{aligned}\text{MM-Comm}(n \times n, d) &\precsim_d \sum_{i=1\cdots d-1} \big(n^3 g_i/Q_i\big) \sum_{k=i\ldots d-1} \big(1/m_k^{1/2}\big)\\ &\precsim_d n^3 \sum_{j=1\ldots d-1} g_j m_j^{-1/2}/Q_j\end{aligned} \tag{18}$$

since by (1) the m_j are nondecreasing in j. Assuming (2) this meets the lower bound (10).

Similarly, the total charge for synchronization at level j is

$$\begin{aligned}
&\left(n^3/\left(Q_j m_j^{3/2}\right)\right)\left(L_{j+1} + \left(m_j^{1/2}/m_{j-1}^{1/2}\right)\text{AC-Synch}(m_j, j)\right)\\
&\precsim_d \left(n^3/\left(m_j^{3/2} Q_j\right)\right)\left(L_{j+1} + \left(m_j^{3/2}/m_{j-1}^{1/2}\right)\sum_{i=1\ldots j-1} L_{i+1}/(Q_i m_i)\right)\\
&\precsim_d n^3 L_{j+1}/\left(m_j^{3/2} Q_j\right) + n^3 \sum_{i=1\ldots j-1} L_{i+1}/\left(Q_i m_i^{3/2}\right).
\end{aligned}$$

Using (17), (1) and $Q_j \geqslant 1$, and adding over all levels $j = 1, \ldots, d-1$ gives

$$\text{MM-Synch}(n \times n, d) \precsim_d n^3 \sum_{j=1\ldots d-1} L_{j+1} m_j^{-3/2}/Q_j \tag{19}$$

since by (1) the m_j are nondecreasing in j. Assuming (2) this meets the lower bound (11).

6.3. Fast Fourier Transform

We consider the FFT problem for input size $N = 2^u$ as a straight-line program where each operation corresponds to a node at a layer $k \in 0, 1, \ldots, u$, and an operation at layer k is a linear combination of the values produced at its two antecedent nodes at layer $k-1$. The operation sequence can be represented as a directed acyclic graph with nodes $(i_1 i_2 \cdots i_u, k)$ where $i_j \in \{0, 1\}$ and $k \in \{0, 1, \ldots, u\}$, and edges $((i_1 i_2 \cdots i_u, k), (i_1 i_2 \cdots i_u, k+1)$ and $((i_1 i_2 \cdots i_u, k), (i_1 i_2 \cdots i^*_{k+1} \cdots i_u, k+1))$ where for $i \in \{0, 1\}$, $i^* \in \{0, 1\}$ is the complement, namely $i + i^* = 1 \bmod 2$. The feature of this network that we use is that once the first $\log_2 x$ layers of operations have been computed, the computation can be completed by doing x independent subproblems on FFT graphs of size N/x.

Our basic algorithm FFT(m_j, x, j) will compute x instances of FFTs on disjoint sets of m_j/x points all initially held in the memory of one level j component with the output to be held also in that memory. By FFT we shall mean here a computation on the FFT graph possibly with different constant multipliers than in the actual FFT. Also, for simplicity of description we are assuming below that the memory in a level j component is a fixed constant multiple of m_j, and as throughout, that all fractions referenced have integral values. The algorithm consists of doing the following:

(i) compute m_j/m_{j-1} disjoint problems of type FFT$(m_{j-1}, 1, j-1)$, and
(ii) on the m_j values so obtained compute FFT$(m_j, x m_{j-1}, j)$.

In other words we shall compute the FFT of x disjoint sets of m_j/x points each, by first doing $m_j/(m_{j-1})$ FFT's on m_{j-1} points each. This will have the effect of increasing the number, and hence also reducing the size, of the resulting disjoint FFT problems remaining to be done, by a factor of m_{j-1}. Hence there will remain $x m_{j-1}$ instances to do each of size $m_j/x m_{j-1}$, and this will be accomplished by (ii).

We can realize (ii) by calling recursively the whole procedure ((i) and (ii) together) with the middle parameter increased by a factor of m_{j-1} each time. Then after $r = \log m_j/\log m_{j-1}$ such calls, FFT(m_j, y, j) will be called with $y = m_j$, which requires no operations. Hence if we denote by FFT-Comm(m_j, x, j) and FFT-Synch(m_j, x, j) the communication and synchronization costs of FFT(m_j, x, j) then

$$\text{FFT-Comm}(m_j, x, j) = \left(m_j/(m_{j-1} p_j)\right)\left[m_{j-1} g_{j-1} + \text{FFT-Comm}(m_{j-1}, 1, j-1)\right] + \text{FFT-Comm}(m_j, x m_{j-1}, j)$$

and

$$\text{FFT-Synch}(m_j, x, j) = \left(m_j/(m_{j-1} p_j)\right)\left[L_j + \text{FFT-Synch}(m_{j-1}, 1, j-1)\right] + \text{FFT-Synch}(m_j, x m_{j-1}, j).$$

Expanding the first through the $r = \log m_j/\log m_{j-1}$ levels of recursion gives that

$$\begin{aligned}
&\text{FFT-Comm}(m_j, x, j)\\
&\quad = \left(m_j/(m_{j-1} p_j)\right)\left[m_{j-1} g_{j-1} + \text{FFT-Comm}(m_{j-1}, 1, j-1)\right]\\
&\quad\quad \vdots\\
&\quad\quad + \left(m_j/(m_{j-1} p_j)\right)\left[m_{j-1} g_{j-1} + \text{FFT-Comm}(m_{j-1}, 1, j-1)\right]\\
&\quad\quad + \text{FFT-Comm}\left(m_j, x(m_{j-1})^r, j\right)\\
&\quad = r\left(m_j/(m_{j-1} p_j)\right)\left[m_{j-1} g_{j-1} + \text{FFT-Comm}(m_{j-1}, 1, j-1)\right]\\
&\quad = (\log m_j/\log m_{j-1}) g_{j-1} m_j/p_j + \left(m_j \log m_j/(p_j m_{j-1} \log m_{j-1})\right)\text{FFT-Comm}(m_{j-1}, 1, j-1).
\end{aligned}$$

Now assuming by induction that

$$\text{FFT-Comm}(m_{j-1}, 1, j-1) \leqslant m_{j-1} \log m_{j-1} \sum_{i=1\ldots j-2} (1/\log m_i) g_i / Q_{i,j-1}$$

and substituting in the above using $Q_{i,j-1} p_j = Q_{i,j}$ gives the following as an upper bound for FFT-Comm(m_j, x, j):

$$\begin{aligned}
&(\log m_j / \log m_{j-1}) g_{j-1} m_j / p_j \\
&\quad + \big(m_j \log m_j / (p_j m_{j-1} \log m_{j-1})\big) \sum_{i=1\ldots j-2} (\log m_{j-1} / \log m_i) g_i m_{j-1} / Q_{i,j-1} \\
&= (\log m_j / \log m_{j-1}) g_{j-1} m_j / p_j + m_j \log m_j \sum_{i=1\ldots j-2} (1/\log m_i) g_i / Q_{i,j} \\
&= m_j \log m_j \sum_{i=1\ldots j-1} (1/\log m_i) g_i / Q_{i,j}.
\end{aligned}$$

Then for $n \leqslant m_d$

$$\text{FFT-Comm}(n, 1, d) \precsim_d \sum_{i=1\ldots d-1} (n \log n) g_i / (Q_i \log m_i). \tag{20}$$

For synchronization the second recurrence gives by an identical argument

$$\text{FFT-Synch}(n, 1, d) \precsim_d \sum_{i=1\ldots d-1} (n \log n) L_{i+1} / (Q_i m_i \log m_i). \tag{21}$$

More simply, (21) follows from (20) by observing that for the parallel cost of a level i superstep, while we charge $m_i g_i$ in (20), the charge is L_{i+1} in (21).

6.4. Sorting

Our purpose is to show that, within constant multipliers, asymptotically optimal parallel computation, communication and synchronization costs can be achieved by a comparison sorting algorithm that makes $(1 + o(1))n \log_2 n$ comparisons altogether. Thus for the full range of parameters, optimality in our sense, that is for large enough m_i, can be achieved. No doubt smaller multipliers for the various costs can be obtained by algorithms that specialize in different ranges of the parameters, but a full analysis remains a challenging problem.

Sorting by deterministic oversampling and splitting into smaller subsets of about equal size is known to be achievable using the following idea [17,29,35]:

Lemma 6.1. *For numbers N, S, G and t one can find a set of S splitters in any ordered set X of N elements such that in the ordering on X the number of elements between two successive splitters is $N/S \pm 2tG$ by using the following procedure: Partition X into G sets of N/G elements each, sort each such set, pick out every tth element from each such sorted list, sort the resulting N/t elements, and finally pick every $N/(tS)$th element of that.*

Let Sort(n, x, j) be a procedure for sorting a set Y of size $n \leqslant m_j$, residing in the memory of a level j component, that includes an identified set of x splitters that already split Y into x sublists of *about* equal size. Our recursive step will divide the set Y into $x m_{j-1}/t_j^2$ sublists of about equal size at the next stage for $t_j = e^{\sqrt{(\log_2 m_{j-1})}}$. This is achieved by applying the method of Lemma 6.1 to each of the x sublists of Y with $N = (m_j/x)$, $S = m_{j-1}/t_j^2$, and $G = N/m_{j-1}$.

Assuming first that the sublists are of exactly the same length we get the recurrence

$$\begin{aligned}
\text{Sort-Comm}(m_j, x, j) \precsim \big(m_j/(m_{j-1} p_j)\big)\big(m_{j-1} g_{j-1} + \text{Sort-Comm}(m_{j-1}, 0, j-1)\big) \\
+ \text{Sort-Comm}\big(m_j, x m_{j-1}/t_j^2, j\big) + \text{Sort-Comm}(m_j/t_j, 0, j).
\end{aligned}$$

First we observe that it follows from the definition of t_j and condition (3) that for any $l > 0$, $t_j = \omega((\log_2 m_j)^l)$. It follows that the last term in the above recurrence can be ignored since it contributes a lower order term even if implemented by a sorting algorithm in which the communication cost is proportional to the computation cost, rather than a logarithmic factor cheaper as is the overall goal. (Note also that the constant k in (3) will only influence the threshold at which the m's can be considered large.)

Omitting that last term leaves the same recurrence as for FFT-Comm but with the multiplier m_{j-1}/t_j^2 rather than m_{j-1} in the second term. Since $\log_2(m_{j-1}/t_j^2) \geqslant (\log_2 m_{j-1})/2$ for large enough m_{j-1} it will have the following solutions analogous to (20) and (21):

$$\text{Sort-Comm}(n,d) \precsim_d \sum_{i=1\ldots d-1} (n\log n)g_i/(Q_i \log m_i), \quad \text{and} \tag{22}$$

$$\text{Sort-Synch}(n,d) \precsim_d \sum_{i=1\ldots d-1} (n\log n)L_{i+1}/(Q_i m_i \log m_i). \tag{23}$$

In fact, the sublists into which the splitters split at any stage, are not of exactly equal length as the analysis above assumes, but are of lengths within multiplicative factors of $1 \pm 2/t_j$ of each other, or $1 \pm o((\log m_j)^{-l})$ for any constant l. This is because the mean of their lengths is $N/S = Nt_{j-1}^2/m_{j-1}$ while the maximum deviation from this length is $2t_{j-1}G = 2Nt_j/m_{j-1}$, which is a fraction $2/t_j$ of that mean. It can be verified that accommodating these variations in the subset lengths does not change the conclusion.

Acknowledgments

I am grateful to Alex Gerbessiotis and Alex Tiskin for their comments on this paper and to Guy Blelloch, Phil Gibbons, Mark Hill, Vijaya Ramachandran and John Savage for conversations.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, M. Snir, A model for hierarchical memory, in: Proc. of the 19th ACM Symp. on Theory of Computing, 1987, pp. 305–314.
[2] A. Aggarwal, A.K. Chandra, M. Snir, Communication complexity of PRAMs, Theoret. Comput. Sci. 71 (1) (1990) 3–28.
[3] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM 31 (9) (1988) 1116–1127.
[4] B. Alpern, L. Carter, E. Feig, T. Selker, A uniform memory hierarchy model of computation, Algorithmica 12 (1994) 72–109.
[5] L. Arge, M. Goodrich, M. Nelson, N. Sitchinava, Fundamental parallel algorithms for private-cache chip multiprocessor, in: Proc. 20th Symp. on Parallelism in Algorithms and Architectures, 2008, pp. 197–206.
[6] G. Bilardi, C. Fantozzi, A. Pietracaprina, G. Pucci, On the effectiveness of D-BSP as a bridging model of parallel computation, in: International Conference on Computational Science, 2001, pp. 579–588.
[7] G. Bilardi, A. Pietracaprina, G. Pucci, K.T. Herley, P. Spirakis, BSP versus LogP, Algorithmica 24 (1999) 405–422.
[8] G. Bilardi, A. Pietracaprina, G. Pucci, F. Silvestri, Network-oblivious algorithms, in: Proc. 21st International Parallel and Distributed Processing Symposium, IPDPS, 2007, pp. 1–10.
[9] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, M. Kozuch, Provably good multicore cache performance for divide and conquer algorithms, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 2008, pp. 501–510.
[10] O. Bonorden, et al., The Paderborn University BSP (PUB) library, Parallel Comput. 29 (2) (2003) 187–207.
[11] R. Chowdhury, V. Ramachandran, Cache-efficient dynamic programming algorithms for multicores, in: Proc. 20th Symp. on Parallelism in Algorithms and Architectures, 2008, pp. 207–216.
[12] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, T. von Eicken, LogP: A practical model of parallel computation, Comm. ACM 39 (11) (1996) 78–85.
[13] P. de la Torre, C.P. Kruskal, Submachine locality in the bulk synchronous setting (extended abstract), in: Euro-Par, vol. II, 1996, pp. 352–358.
[14] F. Dehne, A. Ferreira, E. Caceres, S.W. Song, A. Roncato, Efficient parallel graph algorithms for coarse-grained multicomputers and BSP, Algorithmica 33 (2002) 183–200.
[15] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proc. 40th IEEE Symp. on Foundations of Computer Science, 1999, pp. 285–298.
[16] S. Fortune, C. Wyllie, Parallelism in random access machines, in: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, 1978, pp. 114–118.
[17] A.V. Gerbessiotis, C.J. Siniolakis, Efficient deterministic sorting on the BSP model, Parallel Process. Lett. 9 (1) (1999) 69–79.
[18] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, J. Parallel Distributed Comput. 22 (1994) 251–267.
[19] M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, Towards efficiency and portability: Programming with the BSP model, in: Proc. 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA '96), 1996, pp. 1–12.
[20] M.W. Goudreau, K. Lang, G. Narlikar, S.B. Rao, BOS is boss: A case for bulk-synchronous object systems, in: Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99), 1999, pp. 115–125.
[21] J.M.D. Hill, et al., BSPLib: The BSP programming library, Parallel Comput. 24 (14) (1998) 1947–1980.
[22] M.D. Hill, M.R. Marty, Amdahl's law in the multicore era, IEEE Comput. 41 (July 2008) 33–38.
[23] J. Hong, H. Kung, I/O-complexity: The red–blue pebble game, in: Proceedings of ACM Symposium on Theory of Computing, 1981, pp. 326–333.
[24] Q. Hou, K. Zhou, B. Guo, BSGP: Bulk-synchronous GPU programming, ACM Trans. Graphics 27 (3) (2008) 19:1–19:3.
[25] D. Irony, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, J. Parallel Distributed Comput. 64 (9) (2004) 1017–1026.
[26] R.M. Karp, V. Ramachandran, Parallel algorithms for shared memory machines, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V., 1990, pp. 869–941.
[27] R. Kumar, D. Tullsen, N. Jouppi, P. Ranganathan, Heterogeneous chip multiprocessors, IEEE Comput. (November 2005) 32–38.
[28] W.F. McColl, A. Tiskin, Memory-efficient matrix computations in the BSP model, Algorithmica 24 (3–4) (1999) 287–297.
[29] M.H. Nodine, J.S. Vitter, Optimal deterministic sorting on parallel processors and parallel memory hierarchies, manuscript, 1993. Also: Deterministic distribution sort in shared and distributed memory multiprocessors, in: Proc. ACM Symp. on Parallel Algorithms and Architectures, 1993, pp. 120–129.
[30] V. Ramachandran, Note: Dynamic programming and hierarchical divide and conquer for hierarchical multi-level memories, personal communication, July 2008.
[31] J. Sampson, R. Gonzalez, J.-F. Collard, N.P. Jouppi, M.S. Schlansker, Fast synchronization for chip multiprocessors, SIGARCH Comput. Architecture News 33 (4) (2005) 64–69.
[32] J.E. Savage, Extending the Hong–Kung model to memory hierarchies, in: D.-Z. Du, M. Li (Eds.), Computing and Combinatorics, Springer-Verlag, 1995, pp. 270–281.
[33] J.E. Savage, Models of Computation, Addison–Wesley, Reading, MA, 1998.

[34] J.E. Savage, M. Zubair, A unified model of multicore architectures, in: Proc. First Int. Forum on Next-Generation Multicore/Manycore Technologies, Cairo, Egypt, November 24–25, 2008.
[35] H. Shi, J. Schaeffer, Parallel sorting by regular sampling, J. Parallel Distributed Comput. 14 (1992) 362–372.
[36] A. Tiskin, The bulk-synchronous parallel random access machine, Theoret. Comput. Sci. 196 (1–2) (1998) 109–130.
[37] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.
[38] U. Vishkin, S. Dascal, E. Berkovich, J. Nuzman, Explicit Multi-Threading (XMT) bridging models for instruction parallelism, in: Proc. 1998 ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1998, pp. 140–151.
[39] J.S. Vitter, E.A.M. Shriver, Algorithms for parallel memory II: Hierarchical multilevel memories, Algorithmica 12 (2/3) (1994) 148–169.