Extending the BSP model for multi-core and out-of-core computing: MBSP

Alexandros V. Gerbessiotis[a]

[a] *CS Department, New Jersey Institute of Technology, Newark, NJ 07102, USA.*

Abstract

We present an extension of the bulk-synchronous parallel (BSP) model to abstract and model parallelism in the presence of multiple memory hierarchies and multiple cores. We call the new model MBSP for multi-memory BSP. The BSP model has been used to model internal memory parallel computers; MBSP retains the properties of BSP and in addition can abstract not only traditional external memory-supported parallelism (eg. that uses another level of slower memory) but also multi-level cache-based memory hierarchies such as those present in multi-core systems. Present day multi-core systems are limited parallelism architectures with fast inter-core communication but limited fast memory availability. Abstracting the programming requirements of such architectures in a useful and usable manner is the objective of introducing MBSP. We propose multi-core program and algorithm design that measures resource utilization through a septuplet (p, l, g, m, L, G, M) in which (p, l, g) are the BSP parameters for modeling processor component size and interprocessor communication through latency-based and throughput-based cost mechanisms, and (m, L, G, M) are the new parameters that abstract additional memory hierarchies. Each processor component is attached to a memory of size M, and there are also m memory-units accessing a slower memory of unlimited size of latency-based and throughput-based cost (L, G). A deterministic sorting algorithm is described on this model that is potentially both usable and useful.

Keywords: parallel computing - multi-core - cost model - bulk-synchronous parallel - external memory

Email address: alexg@cs.njit.edu (Alexandros V. Gerbessiotis)

1. Overview

With multi-core systems more emphasis has been put to the design and realization of multiple memory hierarchies to support them and less attention has been drawn to modeling and abstracting their capabilities to a level that they can be understood by a practitioning software engineer. Hardware is useful and effectively utilized if its user is provided with a useful and usable performance prediction model, that facilitates the design, analysis, and implementation of programs and algorithms on it. Moreover, multi-core systems offer limited parallelism and it is worth exploring whether previous parallelism approaches can be exploited at all, or new approaches need to be developed to get satisfactory performance out of them. Multi-core algorithms and software are of paramount importance for the advances in computing envisaged for the 21st century.

The Parallel Random Access Machine (PRAM) [24] consisting of a collection of processors that can synchronously access a shared memory in unit time is a simple yet powerful model of a parallel computer supporting one level of memory interaction, and one could view it as a multi-core processor with all cores having access to its shared (main) memory. In multi-core systems synchronization and communication in one-clock cycle is untenable and thus the need for reliable modeling of these requirements of multi-core computing is apparent, and a PRAM approach is not the solution. The introduction of realistic internal-memory parallel computer models such as the Bulk-Synchronous Parallel (BSP) model [80] and the LogP model [17] and extensions ([55, 8]) address architecture independent parallel algorithm design and take into reasonable consideration synchronization, communication, and latency issues related to communication, bandwidth limitations and network conflicts during routing. They don't deal however with memory related issues, though augmentation of the BSP model to do so is encouraged [80, 16].

The existence of multi-memory hierarchies in multi-cores becomes an issue that needs to be effectively addressed through reliable, useful and usable modeling. In such systems memory becomes available in a multi-level hierarchy of varying and complex configurations and access speed, processor cores are tighter coupled than in traditional multiprocessors, and communication is through shared main memory or sometimes level-3 cache that is much faster than the main memory of traditional systems. This multi-level memory hierarchy in which cores operate poses some interesting challenges. The Multi-BSP model of [81, 82] offers a modeling of all the levels of a memory hier-

archy as long as the multi-core structure is a tree structure of nested components. The modeling approach that is being hereby undertaken does not make any assumptions about the multi-core structure, is less rigid and specific than that of [82], less complicated, possibly less accurate since it uses fewer parameters, but potentially more precise and thus useful and usable. One can still aim for *one-optimality* (ie asymptotical 100% efficiency) [45] which is not easy in [82]. This is because one accepts that any attempt to model the behavior of such complex, highly interacting and configuration-volatile memory hierarchies in as much detail as in [82] becomes a complex architecture dependent endeavor. We propose instead to model multi-core systems in a way that a practicing software designer could follow and use. Although multi-core systems is the intended target, the model is flexible enough to also incorporate external memory.

The methodological model we propose is called MBSP for multi-memory BSP, since memory is pervasive in its interactions with multi-cores. Multi-core adaptability studies of algorithms on MBSP will further establish and verify its usefulness to predict performance of multi-core software. The problem of deterministic sorting will be used as a vehicle for this, and algorithm design and analysis will revolve around the satisfaction of the criterion of one-optimality. As [67] comments, "parallel sorting algorithms are very hard to parallelize .... This implies a large number of memory transactions between CPU cores and main memory through the cache hierarchy". The objective is to further the understanding of the effects tightly-coupled processors or cores have in accessing multi-level memory hierarchies, of the design principles behind multi-core algorithms, and their relationship with traditional parallel algorithms on higher-latency systems with alternative (e.g. external) memory and thus provide an abstraction that software engineers can use to write software whose performance characteristics can be reasoned with.

2. Past modeling efforts

We provide an overview of efforts to model internal memory parallel computing, external (hierarchical) memory parallel, and multi-core models.

2.1. Internal-memory parallel modeling: the BSP model

The BSP model is an internal memory model that has been proposed [80] as a unified framework for the design, analysis and programming of general purpose parallel computing systems. BSP

processors advance jointly through a program, with the required interprocessor communication occurring between supersteps, where a superstep is a segment of computation in which a processor can perform computations using locally available data only. A BSP computer [80] consists of three components: (a) a *collection of p processor/memory components*, (b) a *communication network* that delivers messages point to point among the components, and (c) *facilities* for synchronization of all or a subset of the processors. It is modeled by p, the number of components, l, the minimal time between successive synchronization operations, and g, the cost of communication per word (inverse of router throughput). (Here we use a lower case l for the traditional upper case L of [80].) The definition of g relates to the routing of an h-relation in continuous message usage, that is, the situation where each processor sends or receives at most h messages (or words of information); g is the cost of communication so that an h-relation is realized within gh steps [80], for any h such that $h \geq h_0$, where $h_0 = l/g$ is thus machine dependent. The time complexity of a superstep then becomes $\max\{l, w + gh\}$ basic time steps, where w is the maximum number of basic computational operations executed by any processor during the superstep, and h is the maximum amount of information/messages transmitted or received by any processor. Under the BSP programming paradigm, the objective in a parallel program is to minimize parallel computation and communication time, minimize the number of supersteps and maximize their size, and increase processor utilization.

The same BSP cost is assigned to a single large message transmission of size h or many (i.e. h) messages of small (i.e. unit) size. This abstraction [80] seems to work well in practice, as long as parameter l is modeled/measured properly [37]. Extensions of the BSP model are described in [55, 8]. The former [55] attempts to more accurately predict communication at the expense of architecture independence. The latter attempts to more accurately predict and model interprocessor communication and latency by using an extra parameter, message block size B. This latter approach makes algorithm design more complicated. Another extension is that of the BSPRAM [79]. Local processor memory is incorporated as an extra parameter. Beyond that, remote memory accesses are realized through a shared memory whose performance is modeled by l and g; this yields algorithms that are simulation based akin to the *automatic mode of BSP programming* [45], where PRAM algorithms are simulated on the BSP.

2.2. Multi-memory modeling: the PDM approach

BSP-like models are not fast-memory constrained as are multi-cores. Fast-memory is expensive, limited, and multi-level or hierarchical. The system cache serves the role of "main" or "internal memory" in multi-cores and main memory becomes the "external memory" to the "internal memory" cache. If one decides to extend BSP, one can do it in a "two-step" "simultaneous" process. First extend the BSP model to deal with memory in a limited two-level hierarchy, and then (and concurrently) adapt it to deal with multi-core architectures. Thus a consideration of approaches to model external memory in the presence or not of multiple processors or cores is worthwhile.

Computing models that incorporate external memory as alternative memory, and algorithms and data structures that work in its presence are reviewed in [85, 86]. The Parallel Disk Model (PDM) [84] models external memory primarily in sequential computing ($P = 1$), by using a quadruplet (M, B, D, P), where M is the processor's internal memory size, B is the block transfer size to the external memory, D is the number of drives, and P is the number of processors available (if $P > 1$). Under the PDM, in one I/O operation D blocks of size B can be accessed, one block per disk unit. For the analysis of an algorithm under the PDM, the problem size N is introduced. All I/O activity is expressed as multiples of blocks of size B and the primary measure of performance is (i) number of I/O operations performed, (ii) amount of space used, and (iii) internal computation time. Parallelism in the PDM utilizes a PRAM abstraction [65] and D/P drives are then assigned to each available processor. If one attempts to predict the performance of a parallel (or sometimes, even a sequential) algorithm that uses external memory, the PDM model may be of limited practical use: for many algorithms that work on the PDM model, the internal computation time is ignored and interprocessor communication time is not considered at all for a PRAM parallel algorithm [16]. Ignoring computation time might make sense for applications thought to be I/O-bound. It is noted, however, in [16] that "What is apparent may not be the case, however. Early experiences with algorithms implemented in the PDM indicate that although wall-clock time for a given algorithm follows the prediction of the model, the algorithms themselves are not I/O bound. ..., the time spent waiting for I/O is typically less than 50% of the total wall-clock time". Therefore, if performance prediction is the intention, then, as noted in [16], "... the PDM's predictive power is helpful (for analyzing I/O time) but limited (by omitting computation and communication)".

Computation under the PDM model, with alternating rounds of internal memory processing and external I/O is similar to BSP's bulk-processing of local computation followed by interprocessor communication. One could then extend BSP using features of the PDM that would allow the new model to be both *useful* and *usable*. For a model to be *usable* the number of parameters that abstract the performance of hardware must be kept small. As noted in [16] "The challenge, therefore, is to synthesize a coherent model that combines the best aspects of the PDM and BSP models to develop and analyze algorithms that use parallel I/O, computation, and communication". A model will be *useful* if, by using the words of [16] "Along with such a model, we need programming primitives to enable the algorithm implementation under the model. These primitives must be portable and have performance matching the model's requirements". The incorporation of alternative memory usage into a realistic model of computation (BSP or LogP) could be achieved by using a minimal number of two parameters to express the performance of alternative I/O. Performance would then be expressed in terms of three contributing factors: (i) internal computation time, (ii) interprocessor communication time, and (iii) alternative memory access time.

2.3. Parallel multi-memory modeling: prior PDM and BSP augmentations.

Augmentations of the BSP model to incorporate external memory appear in [19, 23, 20, 54, 22]. The use of BSP to derive external memory algorithms was also independently studied in [76]. In [19, 20] an external memory model, the EM-CGM, is introduced extending CGM under the PDM. The CGM [19] is a restricted version of the BSP model where in each superstep an $O(n/p)$ relation is realized, where n is the problem size. In EM-CGM each processor has its own disk units and I/O is performed only on a processor's local units. EM-BSP and EM-BSP* are introduced in [54, 22, 23] and [54] respectively as the PDM augmentations of BSP and BSP* [8]. In all of these models, each processor has its own D disks and local memory M and modeled under the PDM with I/O performed only to a processor's local disks. The BSP, BSP*/CGM cost models are augmented to include an extra parameter G that models the cost of I/O to a processor's local disks, in addition to the PDM parameters D, B, M.

This modeling approach however could be considered limited. Parallel computing allows a processor to use more memory than locally available by distributing a problem among the local

memories of more than one processors. Therefore *allowing only local disk access may seem limiting as the model becomes not general enough.* In [19, 20, 54, 23, 22] various simulation results are obtained by simulating an internal memory algorithm designed to run on v "virtual processors" on an external memory parallel computer with p physical processors. The approach of using simulations to simulate CGM or traditional BSP algorithms on a target platform that uses external memory is akin to the automatic mode of BSP programming. The *direct mode* of BSP programming [45], however, is more preferable with the programmer retaining absolute control of memory distribution that leads to higher efficiencies, and outperforms the automatic mode [39]. In [76] external memory in a sequential computer is modeled through the BSP model. The underlying idea is one proposed in [14], namely, that external-memory sequential computation can be realized by simulating parallel algorithms on realistic parallel machines that take communication issues into consideration. The method behind the approach in [76] is the following. Slow interprocessor communication and high latency correspond to I/O transfer rate and unit-induced access time (sum of seek and latency time) delays. The execution of an external memory algorithm for a problem of size N in a machine of internal memory M and I/O transfer rate G can be viewed as a parallel internal-memory algorithm on $P = O(N/M)$ BSP processors of internal memory M, with the input fitting into the internal memory of all processors. BSP parameters l and g are then dependent on external memory performance: g is set to G and l to MG to accommodate the in/out paging of a processor's internal memory. This way communication into the external memory becomes interprocessor communication and the cost of access is modeled in terms of the BSP parameters that model interprocessor communication. Moreover, [76] does not utilize block size B. Likewise, the proposed MBSP models I/O performance without using a B, and focuses on the direct mode of programming on MBSP thus deviating from the simulation-based (ie automatic mode) work of [76, 19, 20, 54, 23, 22, 79].

2.4. Multi-BSP

In [81, 82] an alternative memory hierarchy that is a tree structure of nested components is modeled under Multi-BSP. The MBSP modeling approach however, does not make any assumptions about the underlying multi-core structure, is less rigid and specific than that of [81, 82] and possibly less accurate, but we claim more precise and thus more useful and usable: the two models offer

a potential tradeoff between accuracy and usability and usefulness. We accept that detailed and accurate abstraction of memory behavior in the form of a bridging model for the benefit of the average programmer is a complex and possibly unrealistic task for a variety of reasons. Not many programmers are currently using such detailed abstractions of memory in sequential computing. Why would one expect them to use similarly accurate but more complex multi-core models ? Moreover, multi-core systems and their memories are idiosyncratic and architecture and vendor dependent. In several cases level-1 cache is private to a given core (e.g. in Sun Ultra Sparc architectures) but in other designs shared by a small number of cores (e.g. in Sun's Rock architecture several cores share the same level-1 data cache). In Intel's Dunnington architecture the level-2 cache is shared by two cores, and all the cores share a larger level-3 cache. Not many multi-core systems might be modeled as tree structures of nested components adequately enough. Thus any attempt to model the behavior of such varying, complex and highly interacting memory hierarchies becomes a rather architecture dependent endeavor which might make Multi-BSP not useful or usable enough because algorithmic performance would not be precise and thus repeatable and reproducible. In Multi-BSP algorithm design satisfying the criterion of one-optimality is not possible or easy any more [82].

The usefulness and usability of traditional BSP lies in its ability to capture parallelism in an architecture independent manner through parameters l, g. These capture not only synchronization and communication throughput costs, but also communication latency, including software latency. They also capture block-based communication to such a useful and usable level that the introduction of an explicit block size B such as that used in BSP* becomes redundant. Moreover, in the Unified Model for Multicores [74] some complexity results (eg. lower bounds) for some very regular-patterned computations are derived. The complete memory hierarchy is not used to capture these results because of its modeling complexity; instead a level of simplification of the hierarchy is undertaken that is more in line with our proposal and less with that of [82]. We thus propose a simpler approach to use and we feel that with it we can abstract away changes in the interactions and structure of several levels of the memory hierarchy that occur when we move from one generation of multi-core processors to the next. On one hand, one can design and analyze algorithms and potentially evaluate multi-core program performance on our simpler abstraction. If on the other hand,

one encounters problems in such a design or performance evaluation using the two-level hierarchy of ours, it will be unlikely that software designers would undertake modeling under a more complex model such as the Multi-BSP. If software designers are satisfied with such modeling under MBSP, then they might undertake a more detailed modeling under Multi-BSP. Moreover, our approach maintains the simplicity of the traditional BSP model in capturing memory behavior.

3. MBSP: multi-memory BSP

The *Multi-memory Bulk-Synchronous Parallel (MBSP)* model of computation is an augmentation of the BSP model. It is parameterized by the septuplet (p, l, g, m, L, G, M) to abstract, computation and memory interactions among multi-cores. In addition to the modeling offered by the BSP model and abstracted by the triplet (p, l, g), the collection of core/processor components has m alternative memory units distributed in an arbitrary way, and the size of the "fast memory" is M words of information. The model makes no assumption whether an alternative memory unit resides on every processor/core (on-chip) or not. The model does not make any assumptions whether each unit is controlled by one of the p processors or not. Whatever the case is, the cost of memory unit-related I/O is modeled by the pair (L, G). L and G are similar to the BSP parameters l and g respectively. Parameter G expresses the unit transfer time per word of information and thus reflects the memory-unit throughput cost of writing a word of information into a memory unit (i.e. number of local computational operations as units of time per word read/written). Parameter L abstracts the memory-unit access latency time (expressed also in local computational operations as units of time) that reflects access delays contributed mainly but not exclusively by two factors: (a) unit access-related delays that can not be hidden or amortized by G because of the smallness of the amount of data involved, and (b) possible communication-related costs involved in accessing a non-local unit as this could require intraprocessor or interprocessor communication or access to a switching or bus mechanism that facilitates this (cf. the structure of level-3 cache memory in Intel's Dunnington architecture). One may thus require that $L \geq l$ and $G \geq g$.

Under MBSP, the cost of a superstep is $\max\{l, L, w + gh + Gb\}$, where w is the maximum computational work of any processor, h is the maximum number or size messages sent or received by any processor in a superstep and b is the maximum amount of information read or written into

an alternative memory-unit. In MBSP algorithm design satisfying the criterion of one-optimality would imply minimizing not only communication time but also alternative memory access time. One could view MBSP as a two-level version of [81, 82]. However such an abstraction claim would not be accurate; this two-level structure can still model the general behavior and performance characteristics of a multi-level hierarchy abstracted under [82] without the need of the latter to be a tree structure of nested components. Moreover in Multi-BSP the notion of one-optimality is difficult to maintain due to the complexities involved [82]. We claim that in MBSP it is still a useful and potentially usable concept. In addition, if the number of units m is such that $m = Dp$, then each individual processor could still be modeled under the PDM and abstracted by MBSP or EM-BSP. Then memory units would correspond to disks, and G would express the cost of I/O into the local or non-local disks that then become the alternative memory units. The absence of a block size B goes against the tradition of PDM and the work of [21, 54, 22]. We got rid of B for the following reasons.

(a) **Block size B is unnecessary.** In external memory models that use a block size B, the cost of I/O is essentially $B \cdot G$ (though G is not explicitly defined). The choice of B was supposed to be tied to a hard disk drive's track size [54]; yet its choice seems arbitrary and not much useful. Remarks in [54] seem to suggest that peak performance is reached for sizes smaller than the application chosen B. This might suggest that the choice of a single and artificially high B may be arbitrary and not much useful. Let alone the fact that the effective application block size is in fact DB or PDB [22, 23] and thus its relationship to physical disk characteristics becomes even more non-sensical. It is noted in [59] that abstracting memory latency and data transfer through a fixed parameter, B, may not be a good idea, as it limits the flexibility of having a variable block size.

(b) **The lack of a block size B makes algorithm design simpler.** Models such as BSP* [8] that use message block size B make algorithm design more complicated and not more usable than the traditional BSP which implicitly models B by the ratio l/g. As noted in [59] previously, depending on the circumstances there may be cases where one wants the block size to flexibly grow or shrink locally in the algorithm in order to optimize its global performance. So a fixed B is not useful from an algorithmic point of view as well.

4. Evaluating MBSP: modeling its usability and usefulness

Internal memory sequential or parallel sorting of n keys is well studied [58]. First approaches on parallel sorting include Batcher's sorting networks [7] with optimal results achieved by the AKS network [3]. Since then, several results have been obtained [57, 58, 69, 15]. Reif and Valiant [69] use randomized sorting on a fixed-connection network to achieve optimality and smaller constant multipliers. Cole [15] presents the first optimal $O(\lg n)$ PRAM algorithm. These methods fail to deliver optimal performance on realistic BSP-like models. Previous results on BSP sorting include deterministic [1, 11, 30, 34, 47, 77] and randomized [45, 32, 36] algorithms. These algorithms exhibit in many instances optimal performance, are sometimes quite practical, and algorithm implementations yield performance with optimal or close to optimal speedup even for moderate problem sizes [4, 35]. The observed performance of the implemented algorithms is also within the predictions of the model [41]. It is shown in [1] that for all values of n and p such that $n \geq p$, a BSP version of column-sort [57] requires $O((\lg n/\lg(n/p))^{3.42}((n/p)\lg(n/p) + g(n/p) + l))$ time. In [11], it is shown that an adaptation on the BSP model of the cube-sort algorithm [18] requires time $O(25^{\lg^* n - \lg^*(n/p)}(\lg n/\lg(n/p))^2((n/p)\lg(n/p) + g(n/p) + l))$. Moreover, as noted in [47], a modification of cube-sort shown in [68] eliminates term $25^{\lg^* n - \lg^*(n/p)}$. An adaptation of [3] on the BSP model is available in [11]. The constant factors hidden in these algorithms, however, are large.

One important technique in sorting is that of random oversampling [53, 25, 69, 70] that is quick-sort inspired [52]. The median of three or $2t + 1$ keys is an early instance of the use of the technique of oversampling in splitter selection: draw a larger sample from the input to pick a better (or more reliable) candidate from it as splitter. The power of oversampling is fully developed in the context of parallel sorting in [69] where it is combined with the idea of using more than one splitters [25] employed for external memory sorting. Using the technique of oversampling [53, 69], one can draw a sample of $ps - 1$ keys with s being the *oversampling factor*, sort the sample, and identify $p - 1$ splitters as equidistant keys in the sorted sample. In [69] it is shown that the p sequences of the n keys induced by the splitters will retain with high probability $O(n/p)$ keys.

In [45], a randomized BSP sorting algorithm that uses oversampling is introduced that under realistic assumptions requires computation and communication time $(1 + o(1))\ (n \lg n/p)$, with high-probability, with $p = O(n/\lg^{1+\alpha} n)$, for any constant $\alpha > 0$. A similar (but not on the BSP

model) algorithm for $p^2 < n$ but with a less tight analysis is discussed in [61]. The *fine-tuning of the oversampling factor* s in [45], results in p sorted output sequences that retain with high probability $(1+\epsilon)n/p$ keys, are more finely balanced, and $0 < \epsilon < 1$ is controlled by s. The bounds on processor imbalance during sorting are tighter than those of any other random sampling/oversampling algorithm [53, 25, 69, 70]. This randomized algorithm has been implemented on realistic systems and exhibited performance well within the bounds indicated by the theoretical analysis ([4, 35, 41]). Under the same realistic assumptions of [45], a new deterministic sorting algorithm is introduced in [30], and an optimal bound of $(1+(\lfloor(\beta)^{-1}(1-\beta)\rfloor((1-\beta)/2)+o(1))(n \lg n/p)$ and $O(g\beta^{-1}(n/p))+O(l\ \beta^{-1})$ is shown for computation and communication respectively, where $\beta = \lg(n/p)/\lg n$, thus improving upon the upper bounds of [1, 3, 11, 18, 57]. The bound on computation is subsequently improved by the deterministic sorting algorithm of [34, 40]. For $p = n/\lg^{2+\alpha} n$, $\alpha = \Omega(1)$, the latter requires computation and communication time $(1 + 2/\alpha + o(1))\ n \lg n/p$ and $O(g\ \beta^{-1}\ (n/p)) + O(l\ \beta^{-1})$ respectively. [30, 34, 40] extends deterministic sorting by *regular sampling* [75] to perform deterministic sorting by *regular oversampling*.

Results that use regular sampling are available for cases with $p^2 < n$ [75]. The BSP algorithms in [30, 34, 40] further extend the processor range and achieve asymptotically optimal efficiency very close to that of the randomized BSP algorithms in [45, 36]. The insistence, under an architecture independent algorithm design, of satisfying the criterion of one-optimality led to these improvements. In [47] a BSP adaptation of parallel merge-sort [15] is presented that for all $p \leq n$, requires computation and communication/synchronization time $O(n \lg n/p)$ and $O(g\beta^{-1}\ (n/p)) + O(l\beta^{-1})$ respectively. The BSP adaptation of parallel merge-sort is at least as involved as the original algorithm [15], the constant factors involved are large and the algorithm is of little practical use as opposed to the algorithms in [45, 30, 40]. Other studies of practical parallel algorithms with optimal observed performance [49, 50] contain an analysis of their performance that is architecture dependent and thus cannot be used in a systematic way for the study of the performance of other algorithms or to predict the performance of the given algorithm on other platforms reliably.

In the context of external memory sorting various results have been available; [85] discusses such results. Perhaps one interesting example of a parallel external memory algorithm of benchmark quality is that of AlphaSort [66]. The parallel model employed in that work is a master-slave model

with an inherent sequential part, involving the work of the master process/processor that does all the input/output, with the slaves taking care of internal sorting of various runs. This leads to the conjecture that AlphaSort may not scale well in non-shared memory systems with a larger number of processors. The benefit of using quicksort to take advantage of locality over the more popular in such settings replacement-selection sort is also discussed. Alphasort utilizes parallel I/O on multiple disks, i.e. the output file is divided among multiple disks. A previous merging-based external algorithm, FastSort [71], used a single input and a single output file. Lower and upper bounds for external memory sorting in the event of simultaneous access of D multiple blocks from a single disk are discussed in detail in the work of [2]. The unit of measure for algorithm performance is the number of disk block accesses by an algorithm. The more interesting case where each one of the D blocks originates from a different disk is examined in [84], where the PDM model is introduced. In that work a randomized algorithm is proposed that allows efficient external memory sorting. The algorithm is a sample based distribution sort algorithm.

In [83] sequential external memory sorting is discussed under the PDM model. Various randomized distribution sort methods are introduced and examined. The results of [72] are also utilized; in that work, N arbitrary blocks of D disks can be accessed in $N/D + 1$ simultaneous steps by performing data (block) replication (the replication factor is at most one). The work of [83] was mainly motivated by the work of [6] that introduced a randomized version of simple merge-sort under the PDM model to deal with optimizing block placement in external memory. Among all external sorting algorithms the one in [6] is considered most promising and of practical value, also requiring $D = O(B)$, a not unrealistic assumption. Oversampling can help in quantifying possible imbalances in external memory sorting algorithms which is a desired requirement for any distribution sort algorithm as noted in [85]. The algorithm in [65] is a complicated distribution based sort. [85, page 15] conjectures that distribution sort algorithms may be preferable to merge-sort.

More recent work of sorting on multi-cores includes [60], [67], [73], and [26]. As [67] concludes, "parallel sorting algorithms are very hard to parallelize .... This implies a large number of memory transactions between CPU cores and main memory through the cache hierarchy". And [73] points out that among those sorting algorithms "whose efficient parallelization on manycore GPUs we believe should be explored, foremost among them being sample sort".

5. Preparing for MBSP sorting

Multi-core computing raises some questions about the effects of alternative memory. From an algorithmic point of view, it would be interesting to abstract with MBSP a system with alternative memory as reliably as this has been the case for a parallel computer without alternative memory. A measure of success would be the design and analysis of an algorithm that uses alternative memory and behaves as specified by the proposed model. This may involve multi-core oriented sorting.
For base reference we use a BSP internal memory sorting algorithm referred to as DSort shown in Algorithm 1, that uses regular oversampling [30, 34, 40], an extension of regular sampling [75]. (Moreover, it can also handle duplicate keys without requiring doubling of communication and/or computation time that other approaches seem to require [49, 50, 51].) It is the single iteration version of [30, 40, 35] (an iteration variable m is set to 1 there). For sample sorting [30, 40] use a parallel inefficient sorting method (eg. Batcher's bitonic sort) and not a sequential algorithm as used in [75]. The concept of regular sampling as introduced in [75] works as follows. Split regularly and evenly the n input keys into p sequences of equal size, sort each sequence, pick from each sequence $p-1$ equidistant sample keys for a sample of size $p(p-1)$, and after sorting or multi-way merging these keys, $p-1$ equidistant splitters can be retrieved. The p resulting sequences split by the n keys will be of size no more than $2n/p$ [75]. Although oversampling has been used in randomized sorting [53, 69, 45], the extension of regular sampling into regular oversampling was only used in [30, 40, 35]: it allows for a larger sample $p(p-1)s$, with the deterministic regular oversampling factor s controlled by the algorithm and the p sequences split by the $p-1$ splitters of size no more than $(1+\delta)n/p$, with δ controlled by s.

DSort in Algorithm 1 sorts input X of n keys with P processors, where s is the regular oversampling factor. The are three main phases in DSort: (1) *local sorting*, where each processor locally sorts a sequence of size n/P using a generic sorting algorithm called Is, (2) *partitioning*, where processors cooperate to evenly split the sorted sequences, and (3) *merging*, where processors merge a small number of subsequences of total size $(1+o(1))(n/P)$. Parameter s controls δ through $r = \lceil \omega_n \rceil$ and $s = rP$. Thus $\delta = 1/r = 1/\lceil \omega_n \rceil$ relates to the maximum possible imbalance of the output sequences of DSort. If $P = p$ then Dsort is the algorithm of [30, 40, 35]. Note that [30, 40] performs $\lg n / \lg (n/p)$ iterations of phases 2 (steps 2-6) and 3 (step 7) matching the lower bound

of [47]. In each such iteration of [30, 40] the key set is partitioned into sequences of approximately equal size and then in the following iteration each such sequence is subpartitioned similarly, so that no processor holds more than $(1+o(1))n/P$ keys and sorting is completed using multi-way merging [56]. Local sorting uses a generic merge-sort or heap-sort like function of log-linear performance $n \lg n$ named Is. As a side-note, the choice of P in the MBSP algorithm MCDSort that follows depends on n and M and is $P = n/M$. Proposition 1 simplifies and slightly improves the results shown in a more general context in [30, 34, 40] with Lemma 1 establishing that at the completion of step 7 the input keys are partitioned into (almost) evenly sized subsequences. We first show Lemma 1.

Algorithm 1 DSort $(X, n, P,$ Is $)$ {sort n keys of X; use Is for sorting }

1: InitialSorting. The n input keys are regularly and evenly split into P sequences each one of size approximately n/P. Each sequence is then sorted by Is. Let X_k, $0 \leq k \leq P-1$, be the k-th sequence after sorting.

2: Partitioning: Sample Selection. Let $r = \lceil \omega_n \rceil$ and $s = rP$. Form locally a sample T_k from the sorted X_k. The sample consists of $rP-1$ evenly spaced keys of X_k that partition it into rP evenly sized segments; append the maximum of the sorted X_k (i.e. the last key) into T_k so that the latter gets rP keys.

3: Partitioning: Sample Sorting. Merge all T_k into a sorted sequence T of rP^2 keys. For this employ a non-sequential method such as Batcher's Bitonic-Sort.

4: Partitioning: Splitter Selection. Form the splitter sequence S of $P-1$ keys that contains the $(i \cdot s)$-th smallest keys of T, $1 \leq i \leq P-1$, where $s = rP$.

5: Partitioning: Splitters. Broadcast or form an array of $P-1$ splitters as needed.

6: Partitioning: Split input keys. Then split the sorted X_k around S into sorted subsequences $X_{k,j}$, $0 \leq j \leq P-1$, for all $0 \leq k \leq P-1$.

7: Merging. All sorted subsequences $X_{k,j}$ for all $0 \leq k \leq P-1$ are merged into Y_j, for all $0 \leq j \leq P-1$. The concatenation of Y_j for all j is Y. Return Y.

Lemma 1. *The maximum number of keys n_{max} per processor in* DSort *is $(1+1/\lceil \omega_n \rceil)(n/P) + \lceil \omega_n \rceil P + P$, for any ω_n such that $\omega_n = \Omega(1)$ and $\omega_n = O(\lg n)$, provided that $\omega_n^2 P = O(n/P)$ is also satisfied.*

Proof: Although it is not explicitly mentioned in the description of algorithm DSort we assume that we initially pad the input so that each processor owns exactly $\lceil n/P \rceil$ keys. At most one key is added to each processor (the maximum key can be such a choice). Before performing the sample selection operation, we also pad the input so that afterwards, all segments have the same number of keys that is, $x = \lceil \lceil n/P \rceil / s \rceil$. The padding operation requires time at most $O(s)$, which is within the lower order terms of the analysis of Proposition 1, and therefore, does not affect the asymptotic complexity of the algorithm. We note that padding operations introduce duplicate keys; a discussion of duplicate handling follows the proof of Proposition 1.

Consider an arbitrary splitter s_{is}, where $1 \leq i < P$. There are at least isx keys which are not larger than s_{is}, since there are is segments each of size x whose keys are not larger than s_{is}. Likewise, there are at least $(Ps - is - P + 1)x$ keys which are not smaller than s_{is}, since there are $Ps - is - P + 1$ segments each of size x whose keys are not smaller than s_{is}. Thus, by noting that the total number of keys has been increased (by way of padding operations) from n to Psx, the number of keys b_i that are smaller than s_{is} is bounded as follows.

$$isx \leq b_i \leq Psx - ((Ps - is - P + 1))x.$$

A similar bound can be obtained for b_{i+1}. Substituting $s = \lceil \omega_n \rceil P$ we therefore conclude the following.

$$b_{i+1} - b_i \leq sx + Px - x \leq sx + Px = \lceil \omega_n \rceil Px + Px.$$

The difference $n_i = b_{i+1} - b_i$ is independent of i and gives the maximum number of keys per split sequence. Considering that $x \leq (n + Ps)/(Ps)$ and substituting $s = \lceil \omega_n \rceil P$, the following bound is derived.

$$n_{max} = \left(1 + \frac{1}{\lceil \omega_n \rceil}\right) \frac{n + Ps}{P}.$$

By substituting in the nominator of the previous expression $s = \lceil \omega_n \rceil P$, we conclude that the maximum number of keys n_{max} per processor of function DSort is bounded above as follows.

$$n_{max} = \left(1 + \frac{1}{\lceil \omega_n \rceil}\right) \frac{n}{P} + \lceil \omega_n \rceil P + P.$$

The lemma follows. ■

Proposition 1. *For any n and $P \leq n$, and any function ω_n of n such that $\omega_n = \Omega(1)$, $\omega_n = O(\lg n)$ and $P^2\omega_n^2 \leq n/\lg n$, and for $n_{max} = (1 + 1/\lceil\omega_n\rceil)n/P + \lceil\omega_n\rceil P + P$, algorithm* DSort *if run on a BSP computer with P processors requires time, $(n/P)\lg(n/P) + n_{max}\lg P + O(\omega_n P + \omega_n P \lg^2 P)$ for computation and $gn_{max} + O(L\lg^2 P) + O(g\omega_n P \lg^2 P)$ for communication.*

Corollary 1. *For n, P and ω_n as in Proposition 1, algorithm* DSort *requires computation time $(n\lg n/P)\left(1 + \lg P/(\lceil\omega_n\rceil \lg n) + \lg P/\lg n + O(1/(\omega_n \lg^2 n) + \lg^2 P/(\omega_n \lg^2 n))\right)$ and communication time $(n\lg n/P)\left((1 + 1/\lceil\omega_n\rceil)g/\lg n + O(LP\lg^2 P/(n\lg n)) + O(gP^2\omega_n \lg^2 P/(n\lg n))\right)$, for $L \leq 2n/(P\lg^2 P)$.*

Proof: Initially, the input is assumed to be evenly but otherwise arbitrarily distributed among the P processors. Moreover, the keys are distinct since in an extreme case, we can always make them so by, for example, appending to them the code for their processor/memory location. We later explain how we handle duplicate keys without doubling the number of comparisons performed. Parameter ω_n determines the desired upper bound in processor key imbalance during the key routing operation. The term $1+1/\lceil\omega_n\rceil$ is also referred to as bucket expansion in sample-sort based randomized sorting algorithms ([13]).

In step 1, each processor sorts the keys in its possession. As each processor holds at most $\lceil n/P\rceil$ keys, this step requires time $\lceil n/P\rceil \lg \lceil n/P\rceil$. Algorithm Is is any sequential sorting algorithm of such performance. In step 2, each processor selects locally $\lceil\omega_n\rceil P - 1 = rP - 1$ evenly spaced sample keys, that partition its input into $\lceil\omega_n\rceil P$ evenly sized segments. Additionally, each processor appends to this sorted list the largest key in its input. Let $s = \lceil\omega_n\rceil P = rP$ be the size of the so identified list. Step 2 requires time $O(s)$.

In step 3, the P sorted lists, each consisting of s sample keys, are merged by one of Batcher's methods [7], appropriately modified [56] to handle sorted sequences of size s. This means that instead of comparing individual elements and swapping them if needed, we merge two sorted sequences of size s and splitting them into a lower half and upper half, each one also of size s. The computation and communication time required for step 3 is respectively, $2s(\lg^2 P + \lg P)/2$ and $(\lg^2 P + \lg P)(L + gs)/2$ and will dominate parallel prefix and broadcasting operations in step 5 and step 6 to follow. Let sequence $\langle s_1, s_2, \ldots, s_{Ps}\rangle$ be the result of the merge operation; by assumption it is evenly distributed among the P processors, i.e., subsequence $\langle s_{is+1}, \ldots, s_{(i+1)s}\rangle$, $0 \leq i \leq P-1$, resides in the local memory of the i-th processor.

In step 4, a set of evenly spaced splitters is formed from the sorted sample. A broadcast operation is initiated in step 5, where splitter s_{is}, $1 \leq i < P$, along with its index (address) in the sequence of sample keys is sent to all processors. Steps 4 and 5 require time $O(1)$ and $\max\{L, gO(P)\} + T_{brd}^{P-1}(P)$ for computation and communication respectively where the latter term expresses the cost of broadcasting to P processors $P-1$ pieces of data [44, 33, 38, 43] and it is shown to be [44] equal to $T_{brd}^{w}(P) = (\lceil w/\lceil w/h \rceil \rceil + h - 1)\max\{L, gt\lceil w/h \rceil\}$, for any integer $2 \leq t \leq P$, where $h = \lceil \log_t((t-1)P+1) \rceil - 1$. (In our case $w = P-1$, and the cost of broadcasting is absorbed by the cost of step 3 that uses Batcher's methods.)

In step 6, each processor decides the position of every key it holds with respect to the $P-1$ splitters it received in step 5, by way of sequential merging the splitters with the input keys in $P-1+n/P$ time or alternately by performing a binary search of the splitters into the sorted keys in time $P\lg(n/P)$, and subsequently counts the number of keys that fall into each of the P so identified buckets induced by the $P-1$ splitters. Subsequently, P independent parallel prefix operations are initiated (one for each subsequence) to determine how to split the keys of each bucket as evenly as possible among the processors using the information collected in the merging operation. The P disjoint parallel prefix operations in step 6 are realized by employing a pipelined parallel prefix operation on a tree along the line of [44, 33] thus resulting in a time bound of $P\lg(n/P) + T_{ppf}^{P}(P)$ for step 6, where the latter term expresses the cost of parallel prefix using P processors of P independent prefix operations [44, 38, 43]. By way of [44], computation and communication time is combined into $T_{ppf}^{w}(p) = 2(\lceil w/\lceil w/h \rceil \rceil + h - 1)\max\{L, t\lceil w/h \rceil\} + 2(\lceil w/\lceil w/h \rceil \rceil + h - 1)\max\{L, 2gt\lceil w/h \rceil\}$, for any integer $2 \leq t \leq p$, where $h = \lceil \log_t p \rceil$. (We only use $w = P$, and likewise with step 5, the cost of parallel prefix is also absorbed by the cost of step 3.)

In step 7, each processor uses the information collected by the parallel prefix operation to perform the routing in such a way that the initial ordering of the keys is preserved (i.e. keys received from processor i are stored before those received from j, $i < j$, and also the ordering within i and j is also preserved). Step 7 takes time $\max\{L, gn_{max}\}$. Subsequently in step 7, each processor merges the at most P sorted subsequences that it received through routing. When this step is executed, each processor, by way of Lemma 1 to be shown, possesses at most $P = \min\{P, n_{max}\}$ sorted sequences for a total of at most n_{max} keys, where $n_{max} = (1 + 1/\lceil \omega_n \rceil)(n/P) + \lceil \omega_n \rceil P$. The

cost of this step is that of sequential multi-way merging n_{max} keys by some deterministic algorithm [56], which is $n_{max} \lg P$, as $\omega_n^2 P = O(n/P)$. Summing up all contributions, the result follows. Summing up all the terms for computation and communication and noting the conditions on L and g and assigning a cost of $n \lg n$ to the performance of IS the result follows. ■

Duplicate handling. The computation and communication overhead of duplicate handling is within the lower order terms of the analysis and therefore, the optimality claims still hold unchanged. Algorithm Is is implemented by means of a stable sequential sorting algorithm. Two tags for each input key, processor identifier and memory location/index of the key, are already implicitly available by default, and no extra memory is required to access them. Such tags are only to be used for sample and splitter-related activity by augmenting every sample key into a a record that includes this tagging information. As the additional tags affect the sample only, and the sample is $o(1)$ of the input keys, the memory overhead incurred is small, as is the computational overhead. The tagging information is used in steps 2,3,4,5 and 6. In addition, the merging operation in step 7 must also be stable. If one key of say $X_{u,j}$ is compared to a key of say $X_{v,j}$, then the one that will appear in the output first will be from the lower indexed processor between u, v.

6. MBSP sorting

We sort n keys on an MBSP computer with parameter set (p, l, g, m, L, G, M). For the sake of this brief example we shall employ algorithm DSort as a starting step. We assume that each individual processor can accommodate M keys. Thus whether $2M$ or $M/2$ is needed for that, it will not matter. This is equivalent to using a parameter set $(p, l, g, m, L, G, \Theta(M))$ instead. Algorithm MCDSort is the first and simpler instance of a class of algorithms that can be derived from DSort or [30, 40]. We assume that $n > Mp$ or everything could be sorted using DSort by setting $P = p$ in Algorithm 1. We initially store the n keys into the m memory units of MBSP arbitrarily (and at the end of the presentation we may claim uniformly at random). We assume that we can easily and conceptually split the n keys into consecutive blocks of size M through simple array indexing. The n keys require n/M blocks and we have fewer processors than blocks i.e. $n/M > p$. Each processor will work on an equal number of blocks as any other processor equal to $n/(Mp) = P/p$.

Theorem 1. *For an MBSP computer with parameter set (p, l, g, m, L, G, M), let $r = \lceil \omega_n \rceil$, $\omega_n = \Omega(1)$, and $\omega_n = O(\lg n)$, with $\omega_n^2\ n = O(M^2)$. Algorithm* MCDSort *sorts n keys on a MBSP computer with computation, communication and alternative memory costs that are given by the following expressions $C(n), M(n)$ and $A(n)$ respectively, provided that $r(n/M)^2/p < M$.*

$$C(n) = \left(1 + \frac{1}{\omega_n}\right)\frac{n \lg n}{p} + O\left(\frac{n \lg (n/M)}{\omega_n p} + \frac{rn^2}{pM^2} + M \lg (\frac{n}{Mp}) + \frac{rn^2}{pM^2} \lg^2 p + p + \frac{n}{p} + \frac{n^2}{pM^2}\right),$$

$$M(n) = O\left(l + g\frac{n}{M} + l \lg^2 p + g\frac{rn^2}{M^2 p} \lg^2 p\right),$$

$$A(n) = O\left(L\frac{n}{Mp} + \max\{\frac{Gn}{m}, \frac{Gn}{p}\}\right).$$

Proof: We describe the steps of MCDSort in terms of those of DSort. Let $n/M = P$.

1. InitialSorting. Each processor brings into its local memory a total of P/p blocks one after the other. Each block is of size M and sorting with iS requires $M \lg M$ comparisons. The total computation time for all blocks is $PM \lg M/p$ per processor.

Communication time is absorbed into alternative memory access time. The alternative memory time is throughput bound and upper bounded by Gn/m. This is because there are m units storing all the data and the assumption ("arbitrarily") is that initially all memory units are evenly loaded with input data. Memory units transfer memory in blocks of M keys; each one of them maintains $n/(Mm) = P/m$ blocks. The processors read the blocks one at a time; P/p rounds are needed inducing a synchronization/latency cost P/pL. Each processor receives M keys at a potential cost of MG and a total cost of all P/p rounds of $MG \cdot P/p$. Given that $P = n/M$ the former is $MGP/p = Gn/p$. This is analogous to the cost of Gn/m charged to the memory units. An interesting question is whether Gn/m or Gn/p should be charged. We prefer instead a $\max\{Gn/m, Gn/p\}$; this is akin to assigning to a traditional BSP superstep a cost gh if an h-relation is realized where h is the maximum amount of data transmitted or received by a processor. Thus to summarize, the cost of the first phase is $PM \lg M/p = n \lg M/p$ for computation and $LP/p + \max\{Gn/m, Gn/p\}$ for alternative memory access.

2. Partitioning: Sample Selection. For r as in DSort we pick a regular sample of rP keys for each block X_k of M keys. Total sample size is rP^2 for the whole input, yet each processor maintains rP^2/p sample keys. We may assume that $rP^2/p < M$ so that all the sample keys of a processor can be accommodated and maintained in main memory as a single block. This is equivalent to

$n^2 < M^3 p/r$. Given that $r = O(\omega_n)$ and $\omega_n = O(\lg n)$, this is equivalent to $n < M^{3/2}\sqrt{p}/\sqrt{\lg n}$. Thus such an approach can limit the maximum size of input data MCDSort can handle. If instead of using the one iteration version of [40] outlined in Algorithm 1 we had used a t-iteration variant larger input sizes can be accommodated; an $r/pP^{2/t} < M$ sample size per processor would be required leading to problem sizes as large as $n = o(M^{1+t/2})$, for constant values of t, implying an iterative or recursive MCDSort of t or $t+1$ iterations. The computation time of this step is rP^2/p and can be accommodated within memory M.

3. Partitioning: Sample Sorting. Sample sorting involves sequential multi-way merging or parallel merging of sorted sequences. The starting size of a sequence is rP, the sample size of a sorted block X_k of M keys. The P/p sample sequences residing in a processor can be sorted sequentially using multi-way merging in $(rP^2/p)\lg(P/p)$ time. As soon as a processor obtains a single sorted sequence of M keys, Batcher's bitonic sorter is employed to merge the remaining p sequences across all processors each of size M. This will take time $O((rP^2/p)\lg^2 p)$ for computation and $O((l + g(rP^2/p))\lg^2 p)$ for communication. Total time is $M\lg(P/p) + O((rP^2/p)\lg^2 p)$ and $O((l + g(rP^2/p))\lg^2 p)$ for computation and communication respectively.

4. Partitioning: Splitter Selection. As there are $P-1$ splitters and since it is $P < M$ by way of $P > p$ and $rP^2/p < M$, the time bound of this step can be absorbed by the time bounds of previous steps.

5. Partitioning: Splitters. The $P-1$ splitters are broadcast to all p processors. An efficient broadcasting method is the two superstep approach [38, 43] that requires time $O(l + gP)$. Note that $P < M$.

6. Partitioning: Split Input Keys. In this step we perform two operations. First with a copy of the splitters we split every X_k operated by a given processor (there are P/p of them by step 1) into no more than P subsequences $X_{k,j}$, $0 \leq j \leq P-1$. The size $n_{k,j}$ of $X_{k,j}$ is also recorded. For a given j in step 7, already sorted subsequences $X_{k,j}$, for all k, will be merged into sequence Y_j. At the same time the allocation of Y_j into the m memory units is to be performed in a balanced way. For this a parallel prefix on the $n_{k,j}$ values will be realized which is equivalent to P parallel prefix operations using an algorithm along the lines of the two step broadcasting algorithm of step 5. Lemma 1 guarantees that the maximum imbalance incurred is asymptotically negligible and the

maximum size of any Y_j is at most $(1 + O(1/\omega_n))n/P$.

More specifically, each one of the original blocks X_k involved in step 1 above is re-read. Then either a binary seach on the $P-1$ splitters is performed that takes time $M \lg P$ for block X_k or a merging of the $P-1$ splitters and the M keys in time $M + P - 2$. As there are P/p blocks per processor the total computation time is either $(P/p)M \lg P = n \lg P/p$ or $(P/p)(M + P - 2) = n/p + n^2/(M^2 p)$. By the end of this, the sorted sequences $X_{k,j}$, $0 \leq j \leq P-1$, have been identified and also the number of keys $n_{k,j}$ of each $X_{k,j}$. Subsequently in step 7, the P blocks X_k would give rise to Y_j by merging all $X_{k,j}$, for all $0 \leq k \leq P-1$. The latter operation involves multi-way merging. By way of Lemma 1 each Y_j is at most $(1 + O(1/\omega_n))n/P$ and there are no more than P subsequences $X_{k,j}$. Thus multi-way merging sorts Y_j in time $(1 + O(1/\omega_n))(n/P) \lg P$.

The parallel prefix on the counters $n_{k,j}$ would guarantee that each of the m memory units will receive approximately the same number of keys, if not number of sequences Y_k. The parallel prefix involves P independents prefix operations [38, 43]. This is realized by a two-step algorithm similar to that used for the broadcasting in step 5, except for the additional prefix-related computations. The cost of $O(P + l + gP)$ for the parallel prefix is absorbed by the bound derived in step 5. The alternative memory access time of this step is that of step 1. The total computation cost of this step is thus $n \lg P/p$ or $n/p + n^2/(M^2 p)$ plus $O(P)$ for parallel prefix. The total communication time is $O(l + gP)$. Finally the alternative memory access time is $LP/p + \max\{Gn/m, Gn/p\}$.

7. MERGING. Each one of the blocks Y_j contains no more than $P < M$ sequences of total size $(1 + O(1/\omega_n))n/P = (1 + O(1/\omega_n))M$. Thus each such block can be sorted/merged in time $(1+O(1/\omega_n))n/P \cdot \lg P$, as already explained in step 6. The total computation cost of this step is thus $(P/p)((1 + O(1/\omega_n))n/P) \lg P$ and alternative memory access time is $LP/p + \max\{Gn/m, Gn/p\}$. This completes the description of the algorithm. We derive separately below expressions for the computation, communication, and alternative memory time of algorithm MCDSORT.

Computational time. We start with the computation time. Step 1 contributes a factor $n \lg M/p$ and step 7 a factor $((1 + O(1/\omega_n))n/p) \lg P$. The sum of the two is the dominant factor in the computational time. It is $(1 + 1/\omega_n)n \lg n/p + O(\lg(n/M)n/(p\omega_n))$. Thus we have for computation

time $C(n)$ of MCDSort.

$$C(n) = \left(1 + \frac{1}{\omega_n}\right)\frac{n \lg n}{p} + O\left(\frac{n \lg (n/M)}{\omega_n p} + \frac{rn^2}{pM^2} + M \lg (\frac{n}{Mp}) + \frac{rn^2}{pM^2}\lg^2 p + p + \frac{n}{p} + \frac{n^2}{pM^2}\right).$$

Communication time. For the communication time $M(n)$ of MCDSort we have the following.

$$M(n) = O\left(l + g\frac{n}{M} + l \lg^2 p + g\frac{rn^2}{M^2 p}\lg^2 p\right).$$

Alternative memory time. For the alternative memory access time $A(n)$ of MCDSort we have the following.

$$A(n) = O\left(L\frac{n}{Mp} + \max\{\frac{Gn}{m}, \frac{Gn}{p}\}\right).$$

The conditions in the statement of Theorem 1 are from Lemma 1. ■

Some interesting corollaries can be derived from Theorem 1. The first one Corollary 2 deals with optimality in communication. Corollary 3 deals with optimality in alternative memory access. Then we develop one more derivation from Theorem 1 that indicates what happens if $M = n/p$, i.e. there is enough internal memory per processor to accommodate all of the input.

Corollary 2. *Algorithm* MCDSort *as defined in Theorem 1 is optimal in communication for* $g = o(M \lg n/p)$, $g = o(M^2 \lg n/(nr \lg^2 p))$ *and* $l = o(n \lg n/p \lg^2 p)$.

Proof: From Theorem 1 and the expression for $M(n)$ given by

$$M(n) = O\left(l + g\frac{n}{M} + l \lg^2 p + g\frac{rn^2}{M^2 p}\lg^2 p\right).$$

for the claim to be true it suffices that $gn/M = o(n \lg n/p)$ and $g\frac{rn^2}{M^2 p}\lg^2 p = o(n \lg n/p)$. Also $l \lg^2 p = o(n \lg n/p)$. The results follows. ■

Corollary 3. *Algorithm* MCDSort *as defined in Theorem 1 is optimal in alternative memory access time for* $G = o(m \lg n/p)$ *and* $G = o(\lg n)$ *and* $L = o(M \lg n/p)$.

Proof: From Theorem 1 and the expression for $A(n)$ given by

$$A(n) = O\left(L\frac{n}{Mp} + \max\{\frac{Gn}{m}, \frac{Gn}{p}\}\right).$$

for the claim to be true it suffices that $Gn/m = o(n \lg n/p)$ and $Gn/p = o(n \lg n/p)$. In addition, $Ln/Mp = o(n \lg n/p)$. ■

Corollary 4. *Algorithm* MCDSort *as defined in Theorem 1 for $M = n/p$ exhibits the following performance.*

$$C(n) = \left(1 + \frac{1}{\omega_n}\right) \frac{n \lg n}{p} + O\left(rp \lg^2 p + \frac{n}{p}\right),$$

$$M(n) = O\left(l \lg^2 p + grp \lg^2 p\right),$$

$$A(n) = O\left(L + \frac{Gn}{p}\right).$$

Proof: The result can be derived immediately from the expressions for $C(n), M(n), A(n)$ available in Theorem 1. Lower order terms are asymptotically ignored. As a sidenote, the alternative memory time bound $A(n)$ contains a term Gn/p. This reflects the operations performed in step 7 of algorithm MCDSort. The alternative memory access is reduced to interprocessor communication to communicate the $X_{k,j}$ to the processors that will merge them into Y_j. Thus L, G are in fact l, g respectively. ■

7. Conclusion

We introduced MBSP, an extension of the BSP model to abstract and model parallelism in the presence of multiple memory hierarchies and multiple cores. The performance of MBSP is abstracted by a septuplet (p, l, g, m, L, G, M) with the new additional parameters (m, L, G, M) reflecting the behavior of the alternative memory configurations that can be modeled. As opposed to other approaches that attempt to accurately predict the performance on all levels of systems with multiple memory hierarchies ours does not do so. We believe that such accurate prediction would be difficult or the design of algorithms to take advantage of such memory hierarchies would be difficult, cumbersome or hardware dependent. Our intent was to be able to design and analyze the performance of algorithms and predict their behavior on any platform that can be abstracted under MBSP. Towards this we designed and analyzed the performance of a sorting algorithm. Its operations are kept simple enough for its performance behavior to be understood. It is interesting to see how the performance of an implementation of such an MBSP algorithm would track the theoretical analysis encapsulated in Theorem 1.

[1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures,* pp. 129-136, 1995.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. Communications of the ACM, September 1988, Volume 31, Number 9, pp 1116-1127, ACM Press.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ steps. *Combinatorica*, 3:1-19, 1983.

[4] A. G. Alexandrakis, A. V. Gerbessiotis and C. J. Siniolakis. "Portable and Scalable Algorithm Design on the IBM SP2: The Bulk-Synchronous Parallel Paradigm". In Proceedings of the SUP'EUR '96 Conference, Krakow, Poland, September 1996.

[5] Anonymous. A measure of transaction processing power. Datamation , 31(7):112-118, 1985.

[6] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized merge-sort on parallel disks. Parallel Computing, 23, 4(1997), pp. 601-631.

[7] K. Batcher. Sorting Networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference,* pp. 307-314, 1968.

[8] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, 1995.

[9] A. Baumker, and W. Dittrich. Fully Dynamic Search Trees for an Extension of the BSP Model. In *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, ACM Press 1996.

[10] A. Baumker, W. Dittrich, F. Meyer auf der Heide and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of EUROPAR'96*, LNCS volume 1124, August 1996.

[11] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs. LogP. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 25-32, 1996.

[12] R. H. Bisseling and W. F. McColl. Scientific computing on Bulk-Synchronous Parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[13] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 3-16, 1991, ACM Press.

[14] Y-J. Chiang, M. T. Goodrich, E.F.Grove, R.Tamassia, D.E.Vengroff, J.S.Vitter. External memory graph algorithms. Proc. 6th Symposium on Discrete Algorithms, pp. 139-149, ACM-SIAM, 1995.

[15] R. Cole. Parallel merge sort. *SIAM Journal on Computing,* 17(4):770-785, 1988.

[16] T. H. Cormen and M. T. Goodrich. Strategic directions in computing research, Working Group on storage I/O issues in large-scale computing, position statement, `http://www.cs.jhu.edu/~goodrich/cgc/pubs/io.htm`.

[17] D. E. Culler and R. Karp and D. Patterson and A. Sahay and K. E. Schauser and E. Santos and R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

[18] R. Cypher and J. Sanz. Cubesort: A parallel algorithm for sorting n-data items with s-sorters. *Journal of Algorithms*, 13:211-234, 1992.

[19] F. Dehne, W. Dittrich, D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. Proc. ACM Sumposium on Parallel Algorithms and Architectures, 1997, pp. 106-115.

[20] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari, Parallel virtual memory. In Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, M D, 1999, pp. 889-890.

[21] F. Dehne, W. Dittrich, D. Hutchinson, and A.Maheshwari. Reducing I/O complexity by

simulating coarse grained parallel algorithms, In Proceedings International Parallel Processing Symposium, pages 14-20, 1999, IEEE Computer Society Press.

[22] F. Dehne, W. Dittrich, D. Hutchinson, and A.Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35:567-597, Springer Verlag, NY, 2002.

[23] F. Dehne, W. Dittrich, D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36:97-122, Springer-Verlag, NY, 2003.

[24] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing,* pp. 114-118, 1978.

[25] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM,* 17(3):496-507, July 1970.

[26] Brian A. Garber, Dan Hoeflinger, Xiaoming Li, Maria Jesus Garzaran, and David Padua. Automatic Generation of a Parallel Sorting Algorithm. IEEE International Symposium on Parallel and Distributed Processing, 2008, 14-18 April 2008, p 1-5.

[27] A. V. Gerbessiotis and C. J. Siniolakis. Increasing the efficiency of existing sorting algorithms by using randomized wrappers. The Computer Journal, Vol 46(5), pp 498-504, 2003.

[28] A. V. Gerbessiotis and C .J. Siniolakis. Probabilistic Integer Sorting. Information Processing Letters, 90(4), pp. 187-193, 2004, Elsevier B.V.

[29] A. V. Gerbessiotis and S. Y. Lee. Remote memory access : A case for portable, efficient and library independent parallel programming. Scientific Programming, Volume 12, Number 3, pp. 169-183, 2004.

[30] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 223-232, Padua, Italy, June 1996.

[31] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EURO-PAR'96*, Lyon, France, Lecture Notes in Computer Science, Springer-Verlag, August 1996.

[32] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Tech. Rep. PRG-TR-13-96, Computing Laboratory, Oxford University, 1996.

[33] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the Bulk-Synchronous Parallel model with applications to priority queues. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, 1996.

[34] A. V. Gerbessiotis and C. J. Siniolakis. Efficient Deterministic Sorting on the BSP Model. Tech. Rep. PRG-TR-19-96, Oxford University Computing Lab., 1996.

[35] A. V. Gerbessiotis and C. J. Siniolakis. An Experimental Study of BSP Sorting Algorithms. In Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, January, IEEE Computer Society Press, 1998.

[36] A. V. Gerbessiotis and C. J. Siniolakis. A Randomized Sorting Algorithm on the BSP model. In Proceedings of the International Parallel Processing Symposium, Geneva, Switzerland, IEEE Press, 1997.

[37] A. V. Gerbessiotis, and F. Petrini. Network Performance Assessment under the BSP Model, In *International Workshop on Constructive Methods for Parallel Programming*, June 1998, Gothenborg,Sweden.

[38] A. V. Gerbessiotis. Practical considerations of parallel simulations and architecture independent parallel algorithm design. In *Journal of Parallel and Distributed Computing*, 53:1-25, Academic Press, 1998.

[39] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis and K. R. Sujithan. PRAM Programming:

Theory vs. Practice. In Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, pp. 164-170, Madrid, Spain, January, 1998, IEEE Computer Society Press.

[40] A. V. Gerbessiotis and C. J. Siniolakis. Efficient deterministic sorting on the BSP model. Parallel Processing Letters, Vol 9 No 1 (1999), pp 69-79, World Scientific Publishing Company.

[41] A. V. Gerbessiotis and C. J. Siniolakis. BSP Sorting: An experimental study. `http://www.cis.njit.edu/~alexg/pubs/exsort.ps`.

[42] A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP Model. Parallel Computing, 27(2001), pages 809-822, Elsevier Science, BV.

[43] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel Priority Queue and List Contraction: The BSP Approach. Computing and Informatics, Vol. 21, 2002, 59-90.

[44] A. V. Gerbessiotis and C. J. Siniolakis. Architecture Independent Parallel Selection with Applications to Parallel Priority Queues. Theoretical Computer Science, 301/1-3, pp 119-142, 2003.

[45] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous algorithms. *Journal of Parallel and Distributed Computing*, 22:251-267, Academic Press, 1994.

[46] A. V. Gerbessiotis. `http://www.cs.njit.edu/~alexg/cluster/software.html`.

[47] M. T. Goodrich. Communication-Efficient Parallel Sorting. *Proceedings 26th ACM STOC*, 1996.

[48] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, `http://www.bsp-worldwide.org/`, April 1996.

[49] D. R. Helman, J. JaJa, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Tech. Rep. UMIACS-TR-96-54/CS-TR-3670, The University of Maryland Institute for Advanced Computer Studies, August 1996.

[50] D. R. Helman, D. A. Bader, and J. JaJa. A randomized parallel sorting algorithm with an experimental study. Tech. Rep. UMIACS-TR-96-53/CS-TR-3669, The University of Maryland Institute for Advanced Computer Studies, August 1996.

[51] D. R. Helman, D. A. Bader, and J. JaJa. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211-220, Padua, Italy, June 1996.

[52] Hoare C. A. R. Quicksort. *The Computer Journal*, 5:10-15, 1962.

[53] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. *IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, November 1983.

[54] D.A. Hutchinson Parallel Algorithms in External Memory. Ph.D Thesis, School of Computer Science, Carleton University, May 1999.

[55] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating unbalanced communication and general locality into the BSP model. In *Proceedings of EURO-PAR'96*, Lyon, France, Lecture Notes in Computer Science, Springer-Verlag, August 1996.

[56] Knuth D. E. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1973.

[57] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344-354, 1985.

[58] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, California, 1991.

[59] X. Li, G. Linoff, S.J.Smith, C.Stanfill, K.Thearling A practical external sort for shared disk MPPs Proceedings of Supercomputing '93, pp. 666-675, November 1993.

[60] Haibo Lin, Chao Li, Qian Wang, Yi Zhao, Ninghe Pan, Xiaotong Zhuang, Ling Shao. Automated tuning in parallel sorting on multi-core architectures. Proceedings of the 16th interna-

tional Euro-Par conference on Parallel processing: Part I, Pages 14-25, Springer-Verlag Berlin, Heidelberg 2010.

[61] H. Li and K. C. Sevcik. Parallel Sorting by Overpartitioning. In *Proceedings of the 6-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 46-56, 1994.

[62] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation,* (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.

[63] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

[64] W. F. McColl. An architecture independent programming model for scalable parallel computing. *Portability and Performance for Parallel Processors,* (J. Ferrante and A. J. G. Hey, eds.), John Wiley and Sons, 1994.

[65] M. H. Nodine and J. S. Vitter. Deterministic Distribution Sort in Shared and D istributed Memory Multiprocessors, Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93), Velen, Germany, June-July 1993, 120-129.

[66] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomer. AlphaSort: A cache-sensitive parallel external sort. VLDB Journal, 4(4):603-627, 1995. Also in 1994 ACM SIGMOD International Conference on Management and of Data, Minneapolis, MN, 1994.

[67] D. Pasetto A. Akhriev. A Comparative Study of Parallel Sort Algorithms. In Cristina Videira Lopes & Kathleen Fisher, ed., OOPSLA Companion, ACM, pp. 203-204, 2011.

[68] C. G. Plaxton. Efficient Computation on Sparse Interconnection Networks. PhD Thesis, Department of Computer Science, Stanford University, 1989.

[69] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM,* 34:60-76, January 1987.

[70] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing,* 14(2):396-409, 1985.

[71] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. FastSort: A distributed single-input single-output external sort. SIGMOD Record, 19(2):94-101, June 1990.

[72] P.Sanders, S. Egner, and J.Korst. Fast concurrent access to parallel disks. Proceedings of the 11th annual ACM-SIAM Symposium on Discrete Algorithms, pp. 849-858, San Fransisco, January 2000.

[73] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In Proc. IEEE International Symposium on Parallel & Distributed Processing, May 2009, pp. 1-10, IEEE.

[74] J. E. Savage and M. Zubair. A unified model for multicore architectures. Procs. 1st Int. Forum on Next-Generation Multicore/Manycore Technologies, Nov. 24-25, 2008 (Cairo, Egypt).

[75] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362-372, 1992.

[76] J. F. Sibeyn and M. Kaufmann. BSP-like External-Memory Computation. Proc. 3rd Italian Conference on Algorithms and Complexity, LNCS 1203, pp 229-240, 1997.

[77] C. J. Siniolakis. On the Complexity of BSP Sorting. Tech. Rep. PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996.

[78] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, Vol 6, pp 249-274, 1997.

[79] A. Tiskin. The Bulk-Synchronous Parallel Random Access Machine. Theoretical Computer Science, 196(1-2), pp 109-130, 1998.

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

[81] L. G. Valiant. A bridging model for multi-core computing. In Proceedings of ESA 2008, 16th Annual European Symposium on Algorithms, Dan Halperin, Kurt Mehlhorn (Eds.), Karlsruhe,

Germany, September 15-17, 2008, Lecture Notes in Computer Science 5193, Springer 2008. Also J. Comput. Syst. Sci. 77(1): 154-166 (2011).

[82] L. G. Valiant. A bridging model for multi-core computing. J. Comput. Syst. Sci. 77(1): 154-166 (2011), Elsevier.

[83] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. Proceedings of the 12th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA '01), Washington, DC, January 2001.

[84] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3), 110-147, 1994.

[85] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Computing Surveys, March 2000, ACM Press.

[86] J. S. Vitter. Algorithms and Data Structures for External Memory. Series on Foundations and Trends in Theoretical Computer Science, Now Publishers, Hanover, MA, 2008.