Composition of Efficient Nested BSP Algorithms: Minimum Spanning Tree Computation as an Instructive Example*

Olaf Bonorden   Friedhelm Meyer auf der Heide   Rolf Wanka
Dept. of Mathematics & Computer Science and
Heinz Nixdorf Institute, Paderborn University, 33095 Paderborn, Germany.
{bono|fmadh|wanka}@upb.de

Abstract

We report on the results of an automatic configuration approach for implementing complex parallel BSP algorithms. For this approach, a parallel algorithm is described by a sequence of instructions and of subproblems that have to be solved by other parallel algorithms called as subroutines, together with a mathematical description of its own running time. There also may be free algorithmic parameters as, e. g., the degree of trees in used data structures that have an impact on the running time. As the running time of an algorithm depends on several machine parameters, on some fixed and on the choice of the free algorithmic parameters and on the choice of the parallel subroutines for which the same statement applies in turn, the actual composition of the parallel program for an actual parallel machine from all these ingredients is a difficult task. We have implemented such a configuration system in the Paderborn University BSP library and present as an instructive example the theoretical and experimental results of implementations of sophisticated minimum spanning tree algorithms.

1 Introduction

Motivation. The actual running time of implementations of parallel algorithms depends on two groups of parameters, namely “software” parameters such as, e. g., the number of memory accesses or the degree of a used broadcast tree, and “hardware” parameters such as, e. g., the time necessary to set up a communication or the number of available processors. This leads to the observation that for different parameter constellations different “plain” algorithms are the fastest ones. Furthermore, sophisticated parallel algorithms often make recursive calls and call subroutines that have similar dependencies as mentioned above and that themselves solve complex subproblems for which clever algorithms have been designed and implemented. E. g., efficient parallel algorithms for minimum spanning tree computation use parallel sorting algorithms, broadcast methods and make recursive calls.

Hence, in order to have efficient parallel programs for an actual machine, they have to be *configured*. That means that one has to decide which parallel algorithm has to be taken and which subroutines have to be used on what portion of the parallel machine depending on both types of parameters.

For the Bulk Synchronous Parallel (BSP) model and its extensions, a large variety of efficient parallel algorithms for many problems has been developed and quite accurately analyzed. There are BSP environments as, e. g., the Paderborn University BSP library (PUB) [4] and the Oxford BSPlib [9]

*Partially supported by DFG SFB 376 “Massively Parallel Computation” and by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

that provide a platform that can efficiently execute BSP-like programs. This enables the programmer to choose for the composition of the final program from a pool of available algorithms for his or her parallel machines. However, configuring a parallel program becomes quickly very complex due to the various parameters and (possibly mutually influencing) dependencies. Therefore, it should be done automatically by a program we call *configurator*.

Our contribution. We have implemented a comparatively simple prototype of a configurator that, for a given problem Π, calculates from a library of descriptions of BSP algorithms for Π a composite algorithm for a given machine and input size. All algorithms are written for an extension of Valiant's Bulk-Synchronous Parallel model (BSP, [14]) for which we have the C-library PUB ([4]) that supports all functions for the implementation of BSP programs with subalgorithms. To evaluate the algorithms and to be able to forecast the running time, we appropriately extend the BSP model. The composite program obtained by the configurator is always optimal for this BSP extension.

In order to verify the soundness of the configuration approach and the BSP extension, we present the results of implementations of various algorithms for minimum spanning tree (MST) computation and compare their running times with the measured and predicted running time of the program that is output by the configurator. Our MST algorithms use complex subroutines as, e. g., different integer sorting routines, broadcast tasks, prefix operations, etc., which are also subject to the configuration. For a full overview of the bundle of problems and algorithms, see the "map" provided in Fig. 4. Rather than inventing new MST algorithms or to devise a fast configurator, the aim of this paper is to present what can be achieved by the composition of an efficient parallel MST program from the existing MST algorithms (and the necessary subroutines). Further results on automatically configured programs for the broadcast problem can be found in [3].

Organization of paper. The paper is organized as follows: In Section 2, we give a short overview of the used extensions of the BSP model, the PUB library, and the parallel computer on which we performed our experiments. In Section 3, we specify the input, output and used library and algorithm of our prototypical configurator. In Sections 4 and 5, we present the implementation details and discuss the running times of a collection of minimum spanning tree algorithms.

2 The BSP Model and the Used Platforms

The BSP model. Now we give a short sketch of the BSP model. A discussion of this and other parallel bridging models and their advantages and disadvantages can be found in [10]. In Valiant's Bulk Synchronous Parallel (BSP) model [14], a parallel computer consists of a set of processors with local memory, an interconnection mechanism that allows point-to-point communication, and a mechanism for barrier-style synchronizations (see Figure 1(a)). This is the so-called BSP machine. The hardware parameters are p, the number of available processors, g, the bandwidth inefficiency i. e., the (normalized) time necessary to hand over a Byte to the interconnection mechanism, sometimes also called the *gap*, and L, the time the machines needs for the barrier synchronization. For concrete machines, g and L have to be determined experimentally.

The execution of an BSP algorithm is a sequence of *supersteps*, each terminated by a barrier synchronization. In every superstep, all processors do local computations and send messages to other processors. These messages are not available until the next synchronization has been finished (see Figure 1(b)). The cost, i. e., the running time, of the t-th superstep on a BSP machine is given by $w_t + g \cdot h_t + L$, where, for superstep t, w_t is the maximum local work performed by any processor and

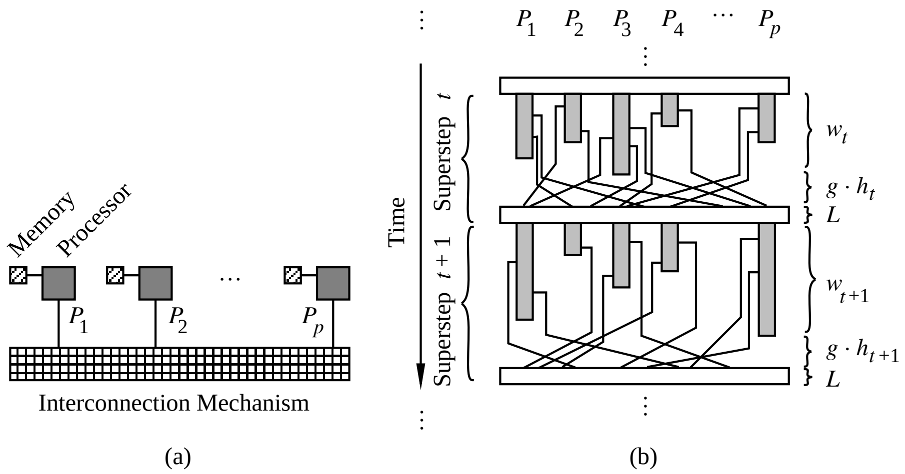


Figure 1: The BSP model. (a) The BSP machine. (b) Execution of a BSP algorithm.

h_t is the maximum number of Bytes sent or received by any processor. After devising a BSP algorithm, the total amount of local work, the total communication volume and the number of supersteps have to be determined by careful analysis.

A variant of the BSP cost model that rewards block-wise communication is BSP* introduced in [2]. It has an additional machine parameter, the critical block size B. In this extension, the communication cost of a single packet to be sent is $\max\{s, B\}$ where s denotes the number of Bytes of the packet. The maximum accumulated communication cost incurred in any processor during a superstep supersedes the number of Bytes from the plain BSP model. Note that g has to be adapted accordingly.

Another extension is the *decomposable* BSP model (D-BSP) [6]. Here the BSP machine may be divided into several partitions, each acts like an independent BSP machine of smaller size. The BSP parameters g and L now are functions of the partitions' size. Hence, algorithms for the D-BSP model can exploit locality. Moreover the partitions can execute different algorithms independently.

The impact of these extension on algorithm design is discussed in [12].

The PUB library. All implementations use the Paderborn University BSP library PUB. A detailed description of PUB can be found in [4] and, very detailed, in [13]. PUB is a C library that supports the development and implementation of parallel algorithms designed for the BSP model. It incorporates the use of block-wise communication as suggested by BSP*, and it provides dynamic partition of the machine as suggested by D-BSP.

The used parallel machines. For all experiments, we used a cluster of 96 Linux workstations with two different communication mechanisms operated by the Paderborn Center for Parallel Computer (PC^2). Every workstation has two Pentium III processors with 850 MHz clock rate and 512 MB memory. As communication mechanism, there are two alternatives: One can choose to use a Fast Ethernet with a Cisco Catalyst 5509 switch, and one can decide to choose as interconnection networks a 2-dimensional torus of SCI links.

Table 1 presents the BSP* parameters of these two parallel computers. They were used in the cost prediction for our implementations. A compilation of the BSP* parameters of further machines as the Cray T3E and Parsytec CC can be found in [13].

Table 1: The BSP* parameters of the workstation cluster.

(a) TCP/IP with Fast Ethernet				(b) MPI with SCI			
p	B [B]	L [μ s]	g [ns^{-1}]	p	B [B]	L [μs]	g [ns^{-1}]
2	3092	98.9	117.1	2	316	2.10	32.4
4	1467	145.5	121.8	4	509	4.49	37.5
8	264	140.7	629.2	8	389	6.28	43.9
16	239	152.7	617.9	16	21	9.08	194.5

3 The Configurator: Input/Output Specification and Algorithm

In what follows, we define how to describe algorithms and introduce the term *schedule* for a given problem. A schedule fixes the algorithms and all free parameters to be used to solve a problem and all occurring subproblems. The input of the configurator is the name Π of the problem, the name of the machine (i. e., which BSP parameters apply), and a specification of the input of Π. The configurator works on a library of algorithm descriptions. It outputs the schedule.

An *algorithm description* A consists of the following five components: [1] Π is the name of the problem the algorithm solves. [2] p_{count} is the set of feasible machine sizes, i. e., the machine sizes for which the algorithm works depending on the input for A. [3] v_{count} is the number of different possible choices for fixing the free algorithmic parameters of A. E. g., in a broadcast algorithm this might be the number of feasible tree degrees. [4] t is the function that computes, depending on a given algorithmic variant, the running time of A without the time A will spent in subcalls. Note that in the algorithm description, it is not known in advance which subcalls will be used in the schedule. [5] r is the function depending on the actual variant that returns the name, input size and number of involved processors for every subproblem. A more formal specification of the components with all details on the parameters involved can be found in [3].

We say A *solves* problem Π.

Given a set $\mathcal{P}$ of problems and a set $\mathcal{A}$ of algorithm descriptions, a *valid schedule* S for a problem $\Pi \in \mathcal{P}$ and its input description is defined recursively. An algorithm $A \in \mathcal{A}$ solving $\Pi \in \mathcal{P}$ is fixed, as well as all free parameters, and there are valid schedules for all occurring subproblems $\Pi'_i \in \mathcal{P}$. The recursion terminates when there are no further subproblems. So a valid schedule S can be viewed as a *schedule tree* directed from the root to the leaves.

Let S be a valid schedule for a problem Π that has to be executed on a p processor BSP machine given by its machine parameters. The *cost* of S, i. e., its running time on the BSP machine, is defined along the schedule tree. The cost of the root is the cost of A (without the subroutine calls) given by t (see point [4] above) plus the sum of the cost of all children of the root.

We have implemented a prototypical configurator that computes a schedule tree and, hence, a valid schedule with minimum cost by a brute force search testing all possible valid schedules. Note that this computation is offline, i. e., it is done only once before the schedule is used for many inputs. The configurator works for arbitrary problems Π and algorithm descriptions A. The running time of the configurator was for our MST experiments just a few minutes.

4 An Example: Parallel Minimum Spanning Tree Algorithms

As example for a complex problem with a rich combinatorial structure, a usually irregular communication pattern, and a bunch of sophisticated algorithms that in turn use clever subalgorithms, we use the problem of computing a minimum spanning tree (MST) for an undirected weighted graph. For the algorithmic background of the MST problem, see [5]. In this section, we give some notes on the implemented algorithms, in the next section, we report on the experiments by running the composed program and the plain programs, and we compare their running times and relate them to the cost computed by the configurator.

Each of the implemented algorithms is based on the following three basic modules:

Kruskal: Kruskal's sequential algorithm ([11]) tests for each edge in order of increasing weights whether it can be included in the minimal spanning tree, i. e., whether it connects two connected components created by the edges chosen so far.

Borůvka steps: In a Borůvka step (for a nice and detailed description, see [8]) each (super-) node selects its cheapest edge. These edges are added to the MST edges, avoiding circles (these circles have at most length two by construction). After that, the new supernodes are calculated as the connected components and the edges are relabeled according to the new supernodes (removing all edges in the same supernode). This step reduces the number of nodes by a factor of at least two.

We have implemented two algorithms for this problem, namely DENSEBORUVKASTEP which is step (2) of algorithm MST-DENSE in [1], and BORUVKASTEP which is in essence from [8, 7]. DENSEBORUVKASTEP is designed for dense graphs, it calculates the minima for all locally stored edges and then executes a parallel prefix operation to get the edges with global minimal weights for each supernode.

BORUVKASTEP creates adjacent lists for all nodes by grouping edges of the same nodes by integer sorting. Then the minima of each group are calculated by a parallel segmented prefix operation (see [8], Section 4.1.4, for details). Our implementation uses a sequential algorithm for computing the connected components.

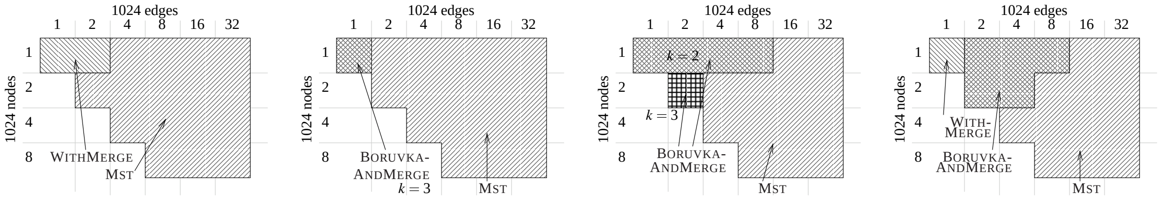
MstMerge: This algorithm (Step (2) of MST-MERGE in [1]) merges local MSTs. It uses a d-ary communication tree. d is a free parameter to be set by the configurator. Each node sends its MST to its predecessor, all predecessors merge the MSTs by calculating an MST of all edges received. In the end, the root of the communication tree knows the global minimum spanning tree.

Figure 4 in the appendix shows a full map of all used algorithms, the generated subproblems and their relationships. In the following, we give short remarks on the three parallel MST algorithms that can be found in the map.

MSTWITHMERGE: This algorithm solves the MST-Problem by calculating the local spanning tree and merge all these trees with MSTMERGE. Then the result is distributed from the root to all nodes.

MST: This algorithm executes Borůvka steps until the number of nodes is 1. Since each step reduces the number of nodes by at least a factor of 2, there are at most $\lceil \log_2 n \rceil$ of these steps.

MSTBORUVKAANDMERGE: This algorithm combines Borůvka steps and merging. First it executes some number k of Borůvka steps to reduce the number of nodes, then it calculates the minimum spanning tree using MSTMERGE. The number k of Borůvka steps is a free parameter of the algorithm and has to be set by the configurator.



(a) $p = 2$ (b) $p = 4$ (c) $p = 8$ (d) $p = 16$

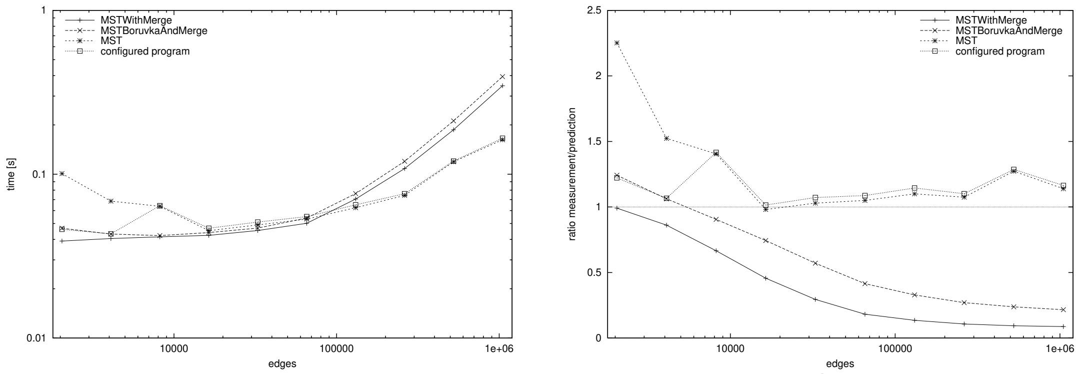
Figure 2: Results of the configuration on a workstation cluster with SCI

5 Experimental Evaluation of MST Implementations

5.1 Configured Program

Figure 2 shows as the result of the configurator the selected algorithms and the fixing of the free parameters for different sizes of the input graphs and different number of processors for the parallel machine interconnected as a 2-dimensional torus of SCI links. The results for the Fast Ethernet interconnection can be found in [3].

On a computer with a fast network, the algorithm MSTWITHMERGE is chosen on small graphs only. This algorithm has a small number of supersteps ($\log_d p$), but calculates a minimum spanning tree sequentially on $(n-1)d$ edges for n nodes in every node of the d-ary communication tree. Also most of the processors (i. e., $p - p/d^{i-1}$) are idle in round i. If the network is slow (as it is the case with, e. g., Fast Ethernet), the change from one chosen algorithm to another occurs for larger graph sizes.



(a) running time (b) accuracy of the predictions

Figure 3: Results for random graphs with 2048 nodes on 16 processors, SCI

5.2 Measured Running Time

Figure 3 shows the results of the measurements on the workstation cluster with 16 processors. Our graphs consist of 2048 nodes and randomly chosen edges. Figure 3 (a) shows the running times, (b) the ratio of the measurements and the predicted times. The divergence for the algorithms with merging is due to inaccurate predictions of the local work. In these algorithms the sequential merging of minimum spanning trees dominates the running time. Our model counts the local memory accesses of these parts. The algorithms run much faster than predicted due to cache effects. More measurement results can be found in [3].

References

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutyłowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proc. 10th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 27–36, 1998.

[2] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *3rd European Symp. on Algorithms (ESA)*, pages 17–30, 1995.

[3] O. Bonorden. Ein System zur automatischen Konfiguration effizienter paralleler Algorithmen im BSP-Modell. Diplomarbeit (in German), Universität Paderborn, 2002.

[4] O. Bonorden, B. H. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library – Design, Implementation and Performance. In *Proc. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP)*, pages 99–104, 1999. Full version as Technical report tr-rsfb-98-063, 1998, Paderborn University.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[6] P. De La Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *EUROPAR: Parallel Processing, 2nd International EURO-PAR Conference*, pages 352–358. LNCS, 1996.

[7] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems, Advances in Parallel and Distributed Systems*, pages 366–371, 1998.

[8] S. Götz. Communication-efficient parallel algorithms for minimal spanning tree computations. Diploma Thesis, Universität Paderborn, http://www.scs.carleton.ca/∼sylvie/work.html, 1998.

[9] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[10] B. H. Juurling and H. A. Wijshoff. A quantitive comparision of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318, Aug. 1998.

[11] J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. In *Proceedings of the American Mathematical Society*, 7, pages 48–50, Feb. 1956.

[12] F. Meyer auf der Heide and R. Wanka. Parallel bridging models and their impact on algorithm design. In *Proc. Int. Conf. on Computational Science (ICCS)*, volume II, pages 628–637, 2001.

[13] I. Rieping. *Communication in Parallel Systems – Models, Algorithms and Implementations*. Ph. Thesis, Paderborn University, Paderborn, 2000.

[14] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

Appendix

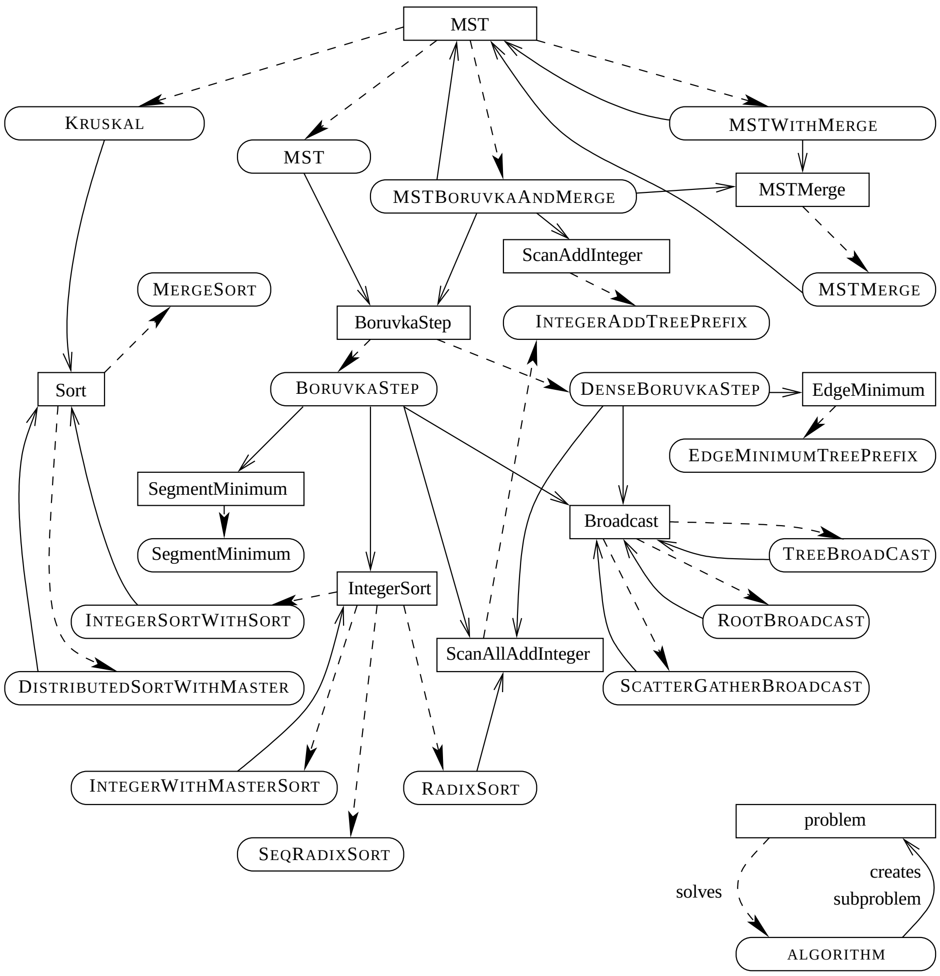


Figure 4: Map of the problems and algorithms for the MST problem