



BSP MODEL DERIVATIVES IMPLEMENTATION AND APPLICATION: AN OVERVIEW

Dr. Mounir Kechid*

Department of Computer Engineering, Faculty of Engineering, Al-Baha University, Saudi Arabia.

Article Info

Article Received: 18 December 2024,
 Article Revised: 07 January 2025,
 Published on: 01 February 2025.



*Corresponding author:

Dr. Mounir Kechid
 Department of Computer
 Engineering, Faculty of Engineering,
 Al-Baha University, Saudi Arabia.
 Email: cibachir@hotmail.com

ABSTRACT

The Bulk Synchronous Parallel (BSP) model has established itself as a cornerstone of parallel processing for over three decades. Its simplicity, coupled with advantages like determinism, portability, and efficiency, has fueled its continued relevance and adaptation to diverse computing landscapes. This paper delves into the rich history of the BSP model, exploring its core principles, extensions, and implementations. We survey the various incarnations of BSP, from its original formulation to modern adaptations for multi-core systems, distributed architectures, and specialized platforms like GPUs. By elucidating the characteristics and strengths of each variant, we aim to equip parallel processing solution developers with a clearer understanding of the BSP landscape and its diverse tools, enabling them to harness its potential for crafting effective and efficient algorithms.

KEYWORDS: Bulk Synchronous Parallel, BSP model, BSP derivatives, parallel computing, portability, parallel algorithm design.

1. INTRODUCTION

The insatiable appetite for computational power fueled by scientific and engineering advancements has propelled parallel computing to the forefront of technological innovation. Amidst the plethora of parallel programming paradigms, the Bulk Synchronous Parallel (BSP) model stands out as a beacon of simplicity and effectiveness. Born in 1990 from the visionary mind of Leslie Valiant (Valiant, 1990), BSP captivated the parallel processing community with its elegance and practicality. Unlike its predecessors grappling with unrealistic abstractions or sacrificing portability for efficiency, BSP embraced a balanced approach. Its core premise revolves around dividing computations into supersteps: alternating phases of independent local work punctuated by synchronized communication and barrier synchronization. This deterministic and well-defined structure offered several captivating advantages: predictability, fostering efficient debugging and performance analysis; platform independence, enabling effortless portability across diverse architectures; and inherent potential for communication optimization, paving the way for efficient algorithm design.

While the original BSP model laid a solid foundation, the evolving landscape of parallel architectures demanded further refinement. Consequently, numerous extensions and specializations arose, tailoring the model to specific computing environments and algorithm classes.

This paper delves into the Bulk Synchronous Parallel (BSP) model, tracing its historical evolution, exploring its diverse extensions, and examining its practical implementations. We aim to illuminate the rich tapestry of the BSP model, empowering parallel processing developers to leverage its strengths in crafting efficient and portable solutions for a wide range of computational challenges.

2. ORIGINAL BSP MODEL

2.1. Philosophical Underpinnings of BSP

The BSP model is based on the principle of dividing parallel computation into successive supersteps. Each superstep consists of a local computation phase, where processing units work independently on assigned tasks, followed by a global communication phase, where processing units exchange data and synchronize. The first superstep takes input from the user's data, and each subsequent superstep takes input from the previous one. The Figure 1 shows the general structure of BSP algorithm.

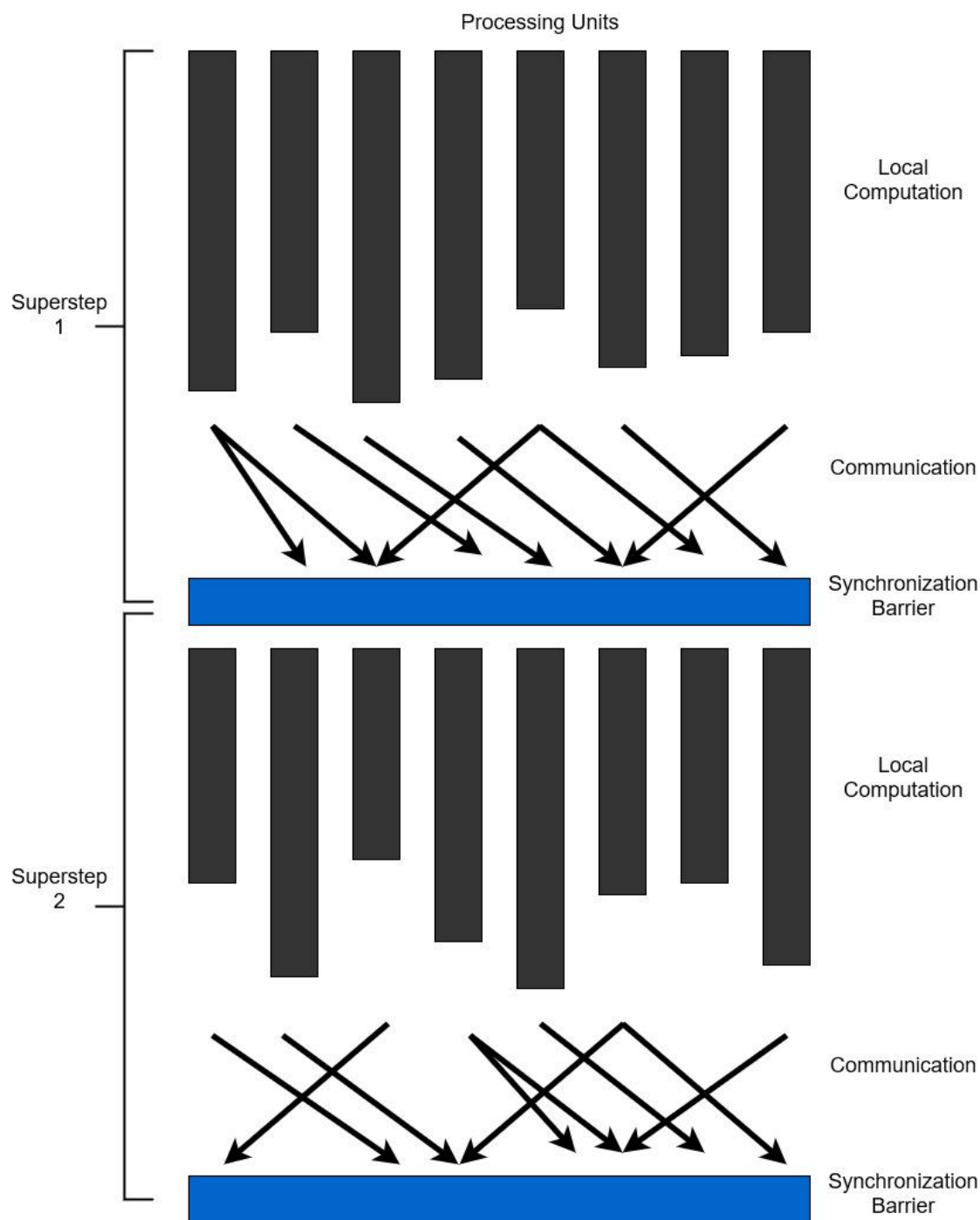


Figure 1. BSP Algorithm General Structure.

In essence, BSP embodies a philosophy of parallel algorithm design that focuses on clear communication boundaries, deterministic execution, and platform independence. In fact, this clear structure of the parallel algorithm offers several key advantages:

- **Efficiency:** By explicitly separating communication and computation, BSP facilitates the identification and optimization of communication bottlenecks, a critical aspect of parallel algorithm performance.
- **Determinism:** The synchronous nature of BSP guarantees predictable and reproducible algorithm

execution, improving debugging and performance analysis.

- **Portability:** BSP algorithms are hardware-agnostic and can be easily adapted to different parallel architectures without significant modification.

2.2. BSP Cost Model

The BSP Cost Model is a fundamental tool for BSP algorithm designers. It is used to analyze and predict the efficiency of parallel algorithms. This model focuses on three key metrics:

- **Supersteps:** The total number of supersteps required by the algorithm, directly impacting communication overhead.
- **Local work:** The computational workload performed by each processing unit per superstep.
- **Global communication volume:** The total amount of data exchanged between processing units in each superstep.

The BSP cost model measures the performance of a parallel algorithm in terms of three parameters: **p**, the number of processing units; **w**, the maximum amount of local work done by any processing unit in a superstep; **h**, the maximum number of messages sent or received by any processing unit in a superstep; and **g**, the communication bandwidth. The BSP complexity model assumes that the communication cost of a superstep is proportional to **h** times **g**, and that all processing units synchronize at the end of each superstep.

The cost of a superstep in the BSP model can be expressed given by:

$$C = w + g \times h + L$$

Where **L** is the synchronization latency cost.

The cost of a parallel algorithm in the BSP model is given by the sum of the costs of all the supersteps, plus the cost of the input and output.

By evaluating these metrics, the designer can identify potential performance bottlenecks and optimize algorithms for efficient execution on diverse parallel systems.

3. BSP MODEL DERIVATIVES

Valiant's original BSP model has undergone several modifications and extensions to enhance its efficiency and applicability to diverse computing environments.

3.1. Extended BSP (Juurink and Wijshoff, 1996)

An extension of the BSP model for parallel computing is the e-BSP model, which takes into account the locality and communication imbalance of the processing units, which are often ignored by the BSP model. Compared to the BSP model, the e-BSP model can improve the performance and reduce the cost of parallel algorithms. Some experimental results on different parallel architectures validate the e-BSP model.

Two concepts are used by the e-BSP model to account for locality and communication imbalance of the processing units: communication volume and communication distance.

Communication volume is the total amount of data sent or received by a processing unit in a superstep. Communication distance is the number of hops between two processing units in the interconnection network. The e-BSP model assumes that the communication cost of a

superstep is proportional to the product of the communication volume and the communication distance. Therefore, the e-BSP model can capture the effects of unbalanced communication patterns and general locality on the performance of parallel algorithms.

3.2. Decomposable BSP (Beran, 1999)

An improvement of the **BSP model** for parallel computing is the **dBSP model**. The dBSP model allows the processing units to **decompose** into smaller subgroups during a superstep, and then **recompose** into larger groups in the next superstep. The benefit of this ability for the processing units to decompose into smaller subgroups during a superstep, and then recompose into larger groups in the next superstep, is that it can adapt to the communication pattern of the parallel algorithm. By decomposing and recomposing the processing units, the dBSP model can reduce the communication cost and improve the load balance among the processing units. For example, if the parallel algorithm requires a lot of communication between a few processing units in a superstep, the dBSP model can decompose the processing units into smaller subgroups that contain those processing units, and then recompose them into larger groups in the next superstep. This way, the communication volume and distance can be minimized, and the communication imbalance can be avoided. The dBSP model can also handle dynamic changes in the communication pattern, by adjusting the decomposition and recombination of the processing units according to the input data or the intermediate results. The dBSP model belongs to the class of **weak parallel machines**, which means that it can simulate any other parallel machine with a polynomial slowdown.

3.3. Oblivious BSP (Gonzalez et al., 2000)

Another improvement of the BSP model is **Oblivious BSP model**. This model introduces a zero-cost synchronization mechanism, called oblivious synchronization. This improvement can enhance the performance of some parallel algorithms that have asynchronous or dynamic communication patterns. This mechanism enables different processing units to be in different supersteps at the same time, if they know how many messages they have to receive. An extension of the BSP complexity model is also proposed, as explained below, to handle oblivious barriers and its accuracy is demonstrated (Gonzalez et al., 2001).

The oblivious BSP model extends the BSP complexity model by introducing a new parameter: **k**, the maximum number of messages that a processing unit can receive without synchronizing. The oblivious BSP model assumes that the communication cost of a superstep is proportional to **h** times **g**, plus a constant overhead **α** for each message received out of order. The synchronization cost is assumed to be zero, as long as the number of messages received is less than or equal to **k**.

The oblivious BSP model can improve the performance of some parallel algorithms that have asynchronous or dynamic communication patterns, by reducing the number of synchronization points and allowing more overlap between computation and communication. However, the oblivious BSP model also introduces more variability in the communication cost, depending on the order and timing of the messages. Therefore, the oblivious BSP model requires more careful analysis and tuning of the parameters **h** and **k** to achieve optimal performance.

3.4. Heterogeneous BSP (Williams and Parsons, 2000)

Leveraging the foundation of the Bulk Synchronous Parallel (BSP) model, Williams and Parsons' HBSP model ventures into the realm of heterogeneous clusters, where diverse processing power and capabilities reign supreme. Unlike its predecessor, HBSP caters specifically to the complexities of such environments, guiding the development and analysis of parallel algorithms tailored for heterogeneous platforms.

The key distinction lies in HBSP's ability to account for the varying speeds of processing units and the communication network through specialized parameters. This nuanced approach unlocks the full potential of heterogeneous clusters, enabling efficient parallel algorithms that seamlessly navigate the differences in processing power and communication overhead.

3.5. Combination with Other Models (Kothapalli et al., 2009)

Building upon the foundation of the Bulk Synchronous Parallel (BSP) model, Kothapalli et al. present a novel performance prediction model for the CUDA GPGPU platform. Their approach ingeniously bridges the gap between the general BSP framework and the intricacies of the GPU architecture by seamlessly integrating elements from both PRAM and QRQW models. This hybrid approach retains the core principles of BSP while incorporating GPU-specific considerations like scheduling complexities, memory hierarchy nuances, and pipelining effects. Consequently, the resulting model captures a much more refined picture of performance on the CUDA platform compared to the basic BSP framework.

3.6. Multi-BSP (Valiant LG, 2011)

Addressing the real-world challenges of multi-core architectures, Valiant and LG's 2011 brainchild, Multi-BSP, transcends the limitations of the BSP model. This practical framework tackles the complexities of memory and cache hierarchies, simplifying multi-core programming through key resource parameter identification and a structure for efficient algorithm development across diverse hardware. Processing unit count, memory/cache sizes, communication/synchronization costs, and more are all accounted for.

A unique aspect of this model is the idea of parameter-aware portable algorithms. These are algorithms that can

run efficiently across a variety of architectures, regardless of the number of levels or combination of parameters. The model defines a parameter-free notion of optimality for these algorithms.

The model demonstrates its effectiveness by providing optimal portable algorithms for several fundamental problems, including standard matrix multiplication, the Fast Fourier Transform, and comparison sorting. These algorithms are shown to run efficiently across all combinations of machine parameters. This suggests that despite the complexity of multi-core architectures, it is possible to find general and elegant algorithmic solutions within this model.

3.7. Parallel GPU Model (PGM) (Kirtzic, 2012)

The Parallel GPU model (PGM), originally proposed by Kirtzic in his doctoral research, stands as a specialized adaptation of the BSP model for GPU architectures. Moving beyond the general framework offered by BSP, PGM boasts features targeted towards GPU characteristics and capabilities. It introduces innovative concepts and techniques, along with a multi-grained approach, to support the development of highly efficient algorithms that can extract the maximum potential from diverse GPU configurations.

3.8. BSP-based Model for GPU (Amaris et al., 2009)

Amaris et al propose a simple and intuitive model to predict the execution time of GPU applications based on the BSP (Bulk Synchronous Parallel) model. This model uses the number of computations and memory accesses of the GPU, as well as cache usage information obtained from profiling, to estimate the execution time. The model also includes a single parameter that adjusts for scalability, divergence, optimization effects, and architectural differences.

4. BSP MODEL IMPLEMENTATIONS

To facilitate the development and execution of BSP algorithms the need for manual implementation of the underlying communication and synchronization mechanisms, the BSP programming model has been implemented as API libraries and programming languages.

4.1. BSP Worldwide Standard (Goudreau et al., 1996)

For the BSP original model, a minimum set of core functions was defined by Goudreau et al., as a standard to improve the compatibility. Below are the three more pertinent implementations of this standard.

4.1.1. The Paderborn University BSP (PUB) library (Bonorden et al., 1999)

A parallel C library based on the BSP model. The library provides and a mechanism for synchronizing the processing units in a barrier style and supports buffered and unbuffered asynchronous communication, collective communication, partition operations, and zero-cost synchronization.

4.1.2. Oxford BSP toolset (Hill et al., 1998)

A collection of tools and implementations of BSPlib, a parallel programming library based on the BSP model. The toolset includes profiling tools that measure the computation and communication costs of BSP programs, and native and generic implementations of BSPlib for various machines.

4.1.3. BSPonMPI (Suijlen, 2007)

A BSPlib implementation on top of MPI, a widely used message passing library. BSPonMPI aims to provide a simple and efficient way of developing scalable parallel programs using the BSP model.

4.2. BSGP (Hou et al., 2008)

A new programming language designed for general-purpose computing on GPUs. BSGP offers the following key features:

- **C-like syntax:** BSGP is designed to resemble sequential C programming, making it familiar and easy to learn for programmers with C experience.
- **Minimal GPU-specific code:** BSGP requires only a small amount of additional code to express parallel processing, reducing complexity and enhancing maintainability.
- **Performance optimization:** The BSGP compiler efficiently maps BSGP programs to GPU kernels and optimizes temporary stream allocation, ensuring high performance.
- **Competitive performance:** BSGP programs have demonstrated performance comparable to or even exceeding well-optimized CUDA programs in benchmarks.
- **Reduced code complexity and development time:** BSGP significantly simplifies source code and accelerates programming time compared to other GPU programming languages.
- **Ease of implementation for complex GPU applications:** BSGP has successfully been used to implement a variety of GPU applications, including a sophisticated X3D parser that would be challenging to develop using existing GPU programming languages.

4.3. Multi-core architecture BSP Implementations

Several implementations for Multi-BSP model were developed. Below, are the two most relevant ones:

4.3.1. MulticoreBSP for C (Yezelman and al., 2014)

MulticoreBSP is a BSP library specifically designed for C programming on multicore architectures. The version (2.0.3, 2018) introduces new features that enhance its capabilities for checkpointing, debugging, profiling, and flexibility in development. It provides a valuable tool for creating efficient and adaptable parallel programs on multicore systems.

4.3.2. Multi-ML (Allombert and Gava, 2018) is a parallel programming language to implements MULTI-BSP model algorithms for the ml programming language (ocaml).

4.4. Generic-Architecture Implementation: Bulk (Burlage et al., 2018)

Bulk is a modern C++ interface for Bulk-Synchronous Parallel (BSP) programming, seamlessly blends familiarity with efficiency. Its intuitive syntax integrates with modern C++ practices, targeting BSP programmers who value speed, safety, clarity, and portability. Bulk enables data-race-free algorithms that work across shared-memory, distributed-memory, and hybrid systems. Emphasis on code clarity and portability leads to understandable, maintainable, and adaptable programs.

5. BSP MODEL APPLICATIONS

The Bulk Synchronous Parallel (BSP) model serves as a foundational framework for several renowned parallel computing systems, each demonstrating its applicability across diverse domains.

5.1. MapReduce (Dean & Ghemawat, 2008)

While not a traditional BSP system by design, MapReduce leverages principles from the model to achieve automatic scalability and fault tolerance. Its core operators, mappers and reducers, act as computing units within consecutive BSP supersteps, with intermediate data shuffling and HDFS I/O facilitating communication and synchronization phases.

5.2. Apache Hama (Siddique et al., 2016)

Extending beyond MapReduce's scope, Apache Hama explicitly builds upon the BSP model to enable a broader range of parallel applications. It offers three distinct computing unit abstractions: general, vertex-based (for efficient graph processing), and neuron-based (tailored for machine learning). This flexibility allows Hama to tailor communication and synchronization optimizations to the specific BSP application at hand.

5.3. Pregel (Malewicz et al., 2010)

Pregel and its iterations represent a specialized BSP application, excelling in distributed large-scale graph computations. Each vertex within the input graph acts as a dedicated computing unit. During graph iterations, vertices exchange messages about their state and edge weights to achieve updated states. Pregel leverages the predictable communication patterns (vertices only communicate with direct neighbors) for performance optimization in later iterations. Additionally, it utilizes commutative and associative aggregation functions to reduce message sizes and minimize network overhead. Pregel algorithms progress through multiple BSP iterations until vertex states converge or a predetermined iteration limit is reached.

6. DISCUSSION

The BSP model's enduring appeal lies in its ability to strike a delicate balance between abstraction and practicality. While its core principles remain firmly rooted in the original supersteps structure, the plethora of extensions showcases its remarkable adaptability to diverse

computing environments and algorithm classes. This section delves into the key takeaways from our exploration, highlighting the strengths and limitations of each approach.

Strengths and Limitations

- **Extended BSP (e-BSP):** While effective in mitigating communication imbalances and improving performance for certain algorithms, e-BSP introduces additional overhead and complexity compared to the original model. Its reliance on communication volume and distance estimations necessitates careful analysis and may not be suitable for all scenarios.
- **Decomposable BSP (dBSP):** This dynamic approach offers excellent flexibility for adapting to shifting communication patterns, particularly beneficial for algorithms with asynchronous or irregular communication needs. However, the overhead associated with processing unit decomposition and recomposition can outweigh the benefits in some cases.
- **Oblivious BSP:** The introduction of zero-cost synchronization enhances performance for asynchronous algorithms but can lead to greater variability in communication costs due to the dependence on message order and timing. Careful analysis of communication patterns and parameter tuning are crucial for optimal performance.
- **Heterogeneous BSP (HBSP):** This specialized model leverages the diverse processing power and capabilities of heterogeneous clusters but introduces additional complexity in program development and analysis. Choosing the appropriate HBSP implementation requires careful consideration of the specific cluster architecture and application requirements.
- **Combination with Other Models:** While hybrid approaches like BSP-CUDA offer promising performance gains for GPU architectures, they require expertise in both BSP and GPU programming and may not be easily portable across different platforms.
- **Multi-BSP:** This practical framework simplifies multi-core programming by accounting for resource parameters like processing unit count, memory/cache sizes, and communication costs. However, its effectiveness can be limited by the specific algorithms and hardware configurations involved.
- **Parallel GPU Model (PGM):** Tailored for GPU architectures, PGM offers efficient algorithms and performance optimization but lacks the general-purpose applicability of the original BSP model.

Guidelines for Parallel Solution Designers

Mismatching the BSP variant to the problem can lead to suboptimal performance, inefficient resource utilization, and increased complexity. Careful analysis of algorithm characteristics, target architecture, and performance goals is crucial for informed BSP selection.

To provide actionable insights for parallel solution designers, we recommend the following guidelines:

- (A) **Algorithm Characterization:** Identify the dominant communication patterns, synchronization frequency, and data locality requirements of your algorithm. This will help narrow down the most suitable BSP derivatives.
- (B) **Target Architecture Analysis:** Consider the available processing power, memory hierarchy, and communication network characteristics of your target platform. Choose the BSP variant and implementation compatible with that architecture.
- (C) **Performance Optimization:** Analyze the communication patterns and potential bottlenecks in your algorithm. Consider BSP derivatives like Extended BSP or Oblivious BSP that optimize communication patterns.
- (D) **Portability Considerations:** If portability is a concern, prioritize standard BSP implementations or models like Multi-BSP with broader applicability across platforms.
- (E) **Libraries and Tools**
 - **Leverage existing libraries and frameworks:** Utilize established BSP libraries like BSPlib, PUB, or BSGP to avoid reinventing the wheel and benefit from mature implementations and tools. These tools can simplify program development, debugging, and performance analysis.
 - **Stay updated on advances:** Explore new BSP derivatives and implementations tailored for specific architectures or algorithm types.
 - **Utilize performance profiling tools:** Analyze communication patterns and resource utilization to tune BSP parameters and optimize performance for your specific application.

Remember, these guidelines offer a starting point. Thorough evaluation of algorithm requirements and target architecture is crucial for optimal BSP variant selection, ensuring efficient and scalable parallel solution development.

7. CONCLUSION

In conclusion, the BSP model's journey over the past three decades exemplifies its remarkable adaptability and enduring relevance. Its core principles have inspired a flourishing ecosystem of extensions and implementations, each tailored to address specific challenges and unlock new possibilities. This paper has served as a roadmap, navigating the intricacies of this diverse landscape and highlighting the unique characteristics of each BSP variant. By understanding these nuanced adaptations, developers are empowered to make informed decisions when selecting the most suitable BSP solution for their parallel processing needs. As the world of parallel computing continues to evolve, we can be confident that the BSP model, with its inherent flexibility and versatility, will remain a guiding light, illuminating the path towards efficient and scalable parallel solutions for the challenges of tomorrow.

8. REFERENCES

1. Allombert, V., & Gava, F. (2018). An ML Implementation of the MULTI-BSP Model. In 2018 International Conference on High Performance Computing & Simulation (HPCS) (pp. 500-507). Institute of Electrical and Electronics Engineers. doi: 10.1109/HPCS.2018.00085
2. Amaris, M., Cordeiro, D., Goldman, A., & De Camargo, R. Y. (2015). A Simple BSP-based Model to Predict Execution Time in GPU Applications. In 2015 IEEE 22nd International Conference on High Performance Computing (HiPC) (pp. 285-294). Institute of Electrical and Electronics Engineers. doi: 10.1109/HiPC.2015.34
3. Beran, M. (1999). Decomposable bulk synchronous parallel computers. In M. Chytil & L. Janik (Eds.), SOFSEM'99: Theory and Practice of Informatics (pp. 349-359). Springer, London.
4. Blleloch, G. E., Gibbons, P. B., Matias, Y., & Zagha, M. (1995). Accounting for memory bank contention and delay in high-bandwidth multiprocessors. In Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 95) (pp. 84-94). Association for Computing Machinery.
5. Bonorden, O., Juurlink, B., Von Otte, I., & Rieping, I. (1999). The Paderborn university BSP (PUB) library-design, implementation and performance. In International Symposium on Parallel and Distributed Processing (Proceedings) (pp. 99-104). Institute of Electrical and Electronics Engineers. doi: 10.1109/IPPS.1999.760442
6. Buurlage, J. W., Bannink, T., & Bisseling, R. H. (2018). Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs. In M. Aldinucci, L. Padovani, & M. Torquati (Eds.), Euro-Par 2018: Parallel Processing (pp. 291-303). Springer, Cham. doi: 10.1007/978-3-319-96983-1_37
7. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1): 107-113.
8. González, J. A., León, C., Piccoli, F., Printista, M., Roda, J. L., Rodríguez, C., & de Sande, F. (2000). Oblivious BSP. Springer, Berlin, Heidelberg.
9. González, J. A., López, P., Pena, A., & Rodríguez, C. (2001). Performance prediction of oblivious BSP programs. In R. Sakellariou, J. Gurd, L. Freeman, & J. Keane (Eds.), Euro-Par 2001: Parallel Processing (pp. 156-165). Springer, Berlin, Heidelberg.
10. Goodrich, M. T. (1999). Communication-efficient parallel sorting. SIAM Journal on Computing, 29(2): 416-432.
11. Hill, J. M. D., McColl, W., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T., & Bisseling, R. H. (1998). BSPlib: The BSP programming library. Parallel Computing, 24(14): 1947-1980.
12. Holl, J., McColl, B., Goudreau, M., et al. (1998). Standard: BSPlib: the BSP Programming Library. Parallel Computing, 24(14): 1947-1980.
13. Hou, Q., Zhou, K., & Guo, B. (2008). BSGP: bulk-synchronous GPU programming. In ACM SIGGRAPH 2008 Papers (SIGGRAPH '08) (pp. 19:1-19:12). Association for Computing Machinery. doi: 10.1143/SIGGRAPH.2008.19
14. Juurlink, B. H. H., & Wijshoff, H. A. G. (1996). The e-BSP model: incorporating general locality and unbalanced communication into the BSP model. In Proceedings of the second international Euro-Par conference on parallel processing-volume II (pp. 339-347). Springer, London.
15. Kirtzic, J. S. (2012). A Parallel Algorithm Design Model for the GPU Architecture. PhD thesis, Richardson, TX, USA. Available from UMI ProQuest Dissertation Publishing. (AAI3547670).
16. Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P. J., & Srinathan, K. (2009). A performance prediction model for the CUDA GPGPU platform. In 2009 International Conference on High Performance Computing (HiPC) (pp. 463-472). Institute of Electrical and Electronics Engineers. doi: 10.1109/HiPC.2009.71
17. Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD international conference on management of data (pp. 135-146). Association for Computing Machinery. doi: 10.1145/1837590.1837609
18. Mark W. Goudreau, Jonathan M. D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel, and Thanasis Tsantilas. (1996). A proposal for the BSP Worldwide Standard Library. Technical report, <http://www.bsp-worldwide.org>.
19. Suijken, W. J. (2007). BSPonMPI. <http://bsponmpi.sourceforge.net>.
20. Siddique, K., Akhtar, Z., Yoon, E. J., Jeong, Y. S., Dasgupta, D., & Kim, Y. (2016). Apache Hama: an emerging bulk synchronous parallel computing framework for big data applications. IEEE Access, 4: 8879-8887. doi: 10.1109/ACCESS.2016.2615574
21. Valiant, L. G. (1990). A bridging model for parallel computation. Communications of the ACM, 33(8): 103-111.
22. Valiant, L. G. (2011). A bridging model for multi-core computing. Journal of Computer System Science, 77(1): 154-166.
23. Williams, T. L., & Parsons, R. J. (2000). The heterogeneous bulk synchronous parallel model. Springer, Berlin, Heidelberg.