



A MACHINE LEARNING BSP-BASED APPROACH FOR AUTOMATIC PARALLEL ALGORITHM DESIGN

Dr. Mounir Kechid*

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

Article Info

Article Received: 17 December 2025,
 Article Revised: 06 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

Current automatic parallel algorithm design tools often struggle with limitations like confined functionality, limited portability, and performance bias for specific architectures. This paper proposes a novel approach that leverages the Bulk Synchronous Parallel (BSP) model and machine learning to overcome these hurdles. We explore the feasibility of creating a comprehensive dataset of parallel algorithms using the BSP model to train a machine learning model. This model can then automatically generate efficient and portable parallel algorithms tailored to specific computational problems. Our multi-stage approach extracts parallelism and dependencies from code structure, enriches this information with target architecture features, and feeds it to a GNN for parallel algorithm generation. Although still in the exploratory stage, this research has promising implications for expanding access to automatic parallel algorithm design.

KEYWORDS: Parallel algorithm design, Bulk Synchronous Parallel (BSP) model, machine learning, Big Data, Code performance optimization, Automatic parallelization.

1. INTRODUCTION

Efficient parallel algorithms are crucial for harnessing the full potential of modern computing systems. However, designing such algorithms remains a complex task, demanding expertise in parallel programming models and architectures. This creates a significant barrier to entry, hindering the widespread adoption of parallel processing and its potential impact across diverse fields.

While traditional automatic parallelization tools offer some assistance, their effectiveness and broad applicability are often limited due to inherent constraints like:

- **Confined Functionality:** Many tools focus on specific types of parallelism, limiting their applicability to diverse code structures.
- **Lack of Portability:** Generated parallel code often relies on platform-specific libraries or hardware features, hindering portability across different architectures.
- **Performance Bias:** Some tools prioritize performance for individual architectures, neglecting broader applicability and optimization potential.

Machine learning (ML) has emerged as a promising avenue for addressing these limitations and improving automatic parallelization. Existing research has explored its potential in various ways:

- Tournavitis et al. (2009) developed an ML-powered prediction mechanism to overcome traditional target-

specific and inflexible mapping heuristics (Tournavitis et al., 2009).

- Shen et al. (2021) proposed a framework for ML-based parallelism detection (Shen et al., 2021).
- Chen et al. (2022) proposed multi-view framework to automatically detect potential parallelism opportunities (Chen et al., 2022).

However, existing research on ML-based automatic parallelization primarily focuses on specific aspects like parallelism detection or code/mapping optimization. To the best of our knowledge, no comprehensive approach yet exists for automatic parallel algorithm design (APAD), encompassing the entire process from sequential code to parallel algorithm generation. This gap motivates our work.

We propose a novel ML-driven approach to APAD that leverages the well-defined structure of the Bulk Synchronous Parallel (BSP) model. The BSP model offers several advantages for APAD. It defines a structured approach for algorithm design and analysis, characterized by its well-defined supersteps and synchronization points. This provides predictability, portability across various parallel paradigms through dedicated libraries and frameworks, and clear communication patterns for efficient performance analysis and optimization.

Our multi-stage approach leverages the strengths of BSP and machine learning to automatically generate efficient and portable parallel algorithms:

- (a) Data Acquisition and Preparation:** We construct a comprehensive dataset of sequential code examples, their corresponding BSP-parallelized versions with associated BSP Superstep Graphs (BSPGs) and target parallel architectures. This data fuels the machine learning model.
- (b) XFG Analysis:** Extracted Contextual Flow Graphs (XFGs) undergo thorough analysis to identify potential parallelism, understand dependencies, and optimize communication patterns.
- (c) XFG Augmentation and GNN Input:** The XFGs are enriched with insights gained from the analysis and fed to a Graph Neural Network (GNN) for parallel algorithm generation.
- (d) Model Selection and Training:** The GNN is trained on the enriched XFGs, corresponding BSPGs, and target architecture information to optimize for efficient and portable parallel algorithms.

The rest of this paper is organized as follows. The next section presents a detailed exploration of the BSP model and its suitability for our proposed approach. Next, a section delves into the feasibility of constructing a comprehensive BSP-based dataset. Then, a section

presents the multi-stage methodology that leverages the BSP model and machine learning for automated algorithm generation. The conclusion section reiterates the potential impact of the proposed approach.

2. THE BSP MODEL AS FOUNDATION FOR APAD

The heart of our proposed approach for automatic parallel algorithm design (APAD) lies in the Bulk Synchronous Parallel (BSP) model, a widely embraced computational model/framework for designing parallel algorithms. This section delves into the theoretical underpinnings, efficiency considerations, and diverse implementations of BSP, ultimately highlighting its suitability as a universal tool for automatic parallel algorithm design.

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.

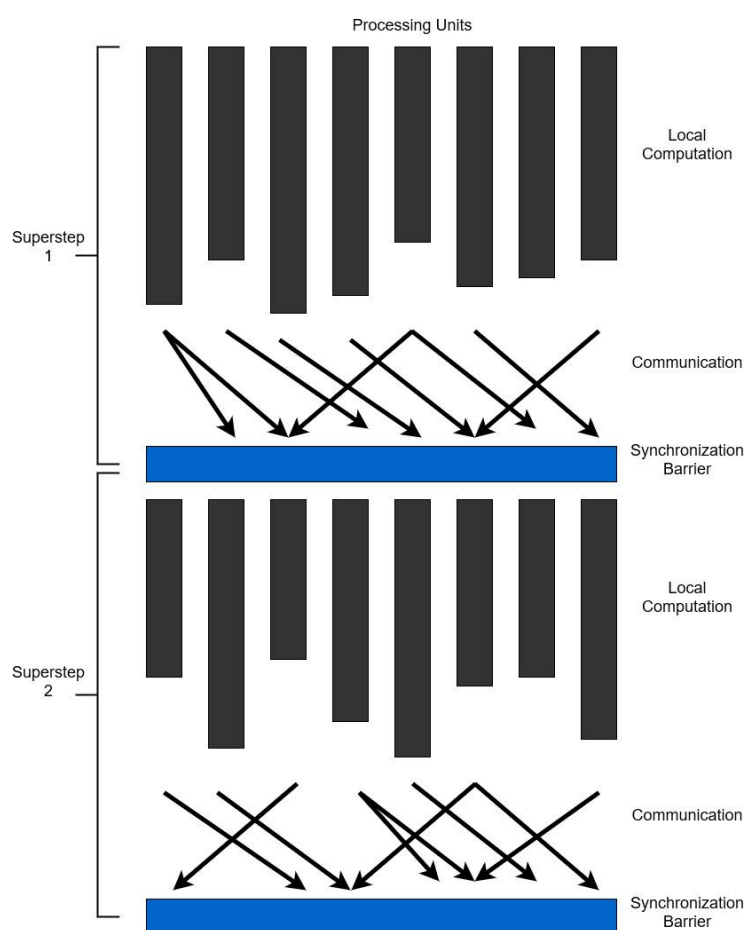


Figure 1: BSP Algorithm General Structure.

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.

2.3. BSP Model Derivatives

Valiant's original Bulk Synchronous Parallel (BSP) model has been refined and extended to address specific challenges in parallel computing environments. Several derivative models offer enhancements in algorithm efficiency and adaptability.

2.3.1. Extended BSP (e-BSP) Model (Juurink and Wijshoff, 1996): The e-BSP model extends the original BSP model by considering locality and communication imbalances among processing units. It introduces concepts such as communication volume and distance to address these issues. Therefore, the e-BSP model can capture the effects of unbalanced communication patterns and general locality on the performance of parallel algorithms.

2.3.2. Decomposable BSP (dBSP) Model (Beran, 1999): The dBSP model improves upon the BSP model by allowing processing units to decompose into smaller subgroups during a superstep, adapting to the communication pattern of parallel algorithms. This decomposition and recombination capability minimizes communication volume and distance, reducing communication costs and improving load balance.

2.3.3. Oblivious BSP Model (Gonzalez et al, 2000): The Oblivious BSP model introduces a zero-cost synchronization mechanism called oblivious synchronization, enhancing the performance of algorithms with 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. The model extends the BSP cost model by introducing a new parameter, **k**, representing the maximum number of messages a processing unit can receive without synchronization.

2.3.4. Heterogeneous BSP (HBSP) Model (Williams and Parsons, 2000): The HBSP model, designed for heterogeneous clusters, accommodates diverse processing power and capabilities. It incorporates specialized parameters to account for varying speeds of processing units and communication network differences, enabling the development of efficient parallel algorithms tailored for heterogeneous platforms.

2.3.5. Combination with Other Models (Kothapalli et al, 2009): Kothapalli et al. present a performance prediction model for the CUDA GPGPU platform by integrating elements from both PRAM and QRQW models into the BSP framework. This hybrid approach captures GPU-specific considerations, such as scheduling complexities and memory hierarchy nuances, providing a refined performance analysis compared to the basic BSP framework.

2.3.6. Multi-BSP (Valiant LG, 2011): Multi-BSP, addressing challenges in multi-core architectures, overcomes the original BSP model limitations. It considers

key resource parameters, including processing unit count, memory/cache sizes, communication/synchronization costs, to simplify multi-core programming and enable efficient algorithm development. The model introduces parameter-aware portable algorithms that run efficiently across diverse architectures.

2.3.7. Parallel GPU Model (PGM) (Kirtzic, 2012): PGM, a specialized adaptation of the BSP model for GPU architectures, introduces innovative concepts and techniques tailored to GPU characteristics. Its multi-grained approach supports the development of highly efficient algorithms, extracting maximum potential from diverse GPU configurations.

2.3.8. BSP-based Model for GPU (Amaris et al, 2009): Amaris et al. propose a simple and intuitive model based on the BSP model to predict the execution time of GPU applications. This model uses computations, memory accesses, and cache usage information to estimate execution time, including a single parameter to account for scalability, divergence, optimization effects, and architectural differences.

2.4. 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. Notable implementations of the BSP model include.

2.4.1. BSP Worldwide Standard (Goudreau et al, 1996): Goudreau et al. established a BSP Worldwide Standard, defining a core set of functions to enhance compatibility. Key implementations of this standard include.

2.4.1.1. Paderborn University BSP (PUB) Library (Bonorden et al, 1999): A parallel C library adhering to the BSP model, PUB offers synchronization mechanisms, buffered and unbuffered asynchronous communication, collective communication, partition operations, and zero-cost synchronization.

2.4.1.2. Oxford BSP Toolset (Hill et al, 1998): Comprising tools and implementations of BSPlib, this toolset supports profiling tools to measure computation and communication costs. It provides native and generic implementations of BSPlib for various machines.

2.4.2. BSPonMPI (Suijlen, 2007): An implementation of BSPlib on top of MPI, BSPonMPI simplifies the development of scalable parallel programs using the BSP model efficiently.

2.4.3. BSGP (Hou et al, 2008): BSGP, a programming language designed for general-purpose GPU computing, emphasizes a C-like syntax, minimal GPU-specific code, performance optimization, competitive performance,

reduced code complexity, and accelerated development time. It has been successfully applied to complex GPU applications.

2.4.4. Multi-core Architecture BSP Implementations: Several implementations cater to the Multi-BSP model, with two notable examples.

2.4.4.1. MulticoreBSP for C (Yezelman et al, 2014): This BSP library, tailored for C programming on multicore architectures, introduces features enhancing capabilities for checkpointing, debugging, profiling, and flexibility in development. It facilitates the creation of efficient and adaptable parallel programs.

2.4.4.2. Multi-ML (Allombert and Gava, 2018): A parallel programming language designed for the ml programming language (ocaml), Multi-ML facilitates the implementation of algorithms based on the MULTI-BSP model.

2.4.5. Generic-Architecture Implementation: Bulk (Buurlage et al, 2018): Bulk, a modern C++ interface for BSP programming, combines familiarity with efficiency. Its intuitive syntax integrates with modern C++ practices, enabling data-race-free algorithms across shared-memory, distributed-memory, and hybrid systems. Emphasis on code clarity and portability results in understandable, maintainable, and adaptable programs.

2.5. BSP Model Applications

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

- MapReduce (Dean & Ghemawat, 2008)
- Apache Hama (Siddique et al., 2016)
- Pregel (Malewicz et al., 2010)

2.6. Conclusion

This in-depth investigation elucidates the versatility and appropriateness of the Bulk Synchronous Parallel (BSP) model concerning our Automated Parallel Algorithm Design (APAD) objectives. The amalgamation of its well-defined structure, practical implementations, and adaptable derivatives establishes a robust underpinning for the development of efficient and portable parallel algorithms, applicable to a broad spectrum of computational problems and architectures.

The enduring significance and effectiveness of the BSP model are apparent in its utilization within acclaimed parallel frameworks and systems like MapReduce, Apache Hama, and Pregel. These diverse applications underscore the model's capability to handle intricate algorithms and its adaptability to diverse computing environments.

With its clear structure, analytical tools, diverse implementations, and demonstrated applications, the BSP model provides a robust foundation for our APAD

approach. Its inherent strengths directly address the limitations of existing techniques, paving the way for the development of more efficient, portable, and universally applicable parallel algorithms. Subsequent sections will delve deeper into the feasibility of constructing a comprehensive BSP-based dataset and the multi-stage methodology that leverages the BSP model and machine learning for automated algorithm generation.

3. FEASIBILITY OF CREATING A COMPREHENSIVE DATASET OF PARALLEL ALGORITHMS USING THE BSP MODEL

The success of our proposed approach hinges on the availability of a rich and diverse dataset of parallel algorithms built upon the BSP model. This section delves into the feasibility of constructing such a dataset, exploring its challenges and potential solutions while highlighting its transformative impact on automatic parallel algorithm design.

3.1. Motivation

Training a robust machine learning model for automatic parallel algorithm generation requires a comprehensive training set encompassing diverse algorithms and problem domains. A carefully curated BSP-based dataset offers several advantages:

- **Uniform Foundation:** The BSP model's clear structure and analytical tools provide a standardized framework for capturing and representing algorithms, facilitating consistent data structure and efficient model training.
- **Portability and Adaptability:** Algorithms represented within the BSP framework inherit its inherent portability, enabling their adaptation to various parallel architectures without significant modifications.
- **Richer Learning Potential:** Access to a wider range of parallel algorithms exposes the machine learning model to diverse solution strategies and communication patterns, enhancing its ability to generalize and generate efficient algorithms for unseen problems.

3.2. Challenges

Constructing a comprehensive BSP dataset presents several noteworthy challenges:

- **Scope and Variety:** Capturing a broad spectrum of algorithms across diverse domains (e.g., scientific computing, image processing, machine learning) and problem sizes presents a significant undertaking.
- **Quality and Consistency:** Ensuring the correctness, efficiency, and portability of every included algorithm requires rigorous validation and verification procedures.
- **Data Acquisition and Representation:** Efficiently gathering parallel algorithms from various sources, representing them in a standardized format within the BSP paradigm, and storing them in a readily accessible database requires careful consideration.

3.3. Potential Solutions

To address these challenges, we suggest a multi-pronged approach.

- **Collaboration:** Partnering with BSP algorithm designers and developers and the large BSP paradigm researcher community can significantly expand the scope and variety of algorithms included in the dataset. This collective endeavor stands to capitalize on the utilization of pre-existing BSP algorithms that have been meticulously designed and implemented across various domains, thereby enhancing the dataset with meaningful real-world instances. It is essential to underscore the considerable abundance of algorithms designed and implemented in BSP model and its derivatives.
- **Benchmarking and Validation:** Implementing rigorous benchmarking and validation procedures, coupled with peer review and community vetting, ensures the quality and consistency of the included algorithms, bolstering the dataset's credibility.

3.4. Expected Benefits

The potential benefits of a comprehensive BSP dataset extend beyond training our proposed machine learning model:

- **Improved Model Training:** Access to a broader and more diverse dataset can significantly enhance the accuracy and generalizability of the machine learning model, leading to the generation of more efficient and effective parallel algorithms.
- **Benchmarking and Evaluation:** The dataset can serve as a valuable resource for benchmarking and evaluating existing and newly generated parallel algorithms. This facilitates comparative analysis, performance optimization, and identification of promising research directions.
- **Community Resource:** A publicly accessible and regularly updated BSP dataset can act as a valuable resource for researchers and developers working in the field of parallel algorithm design, promoting collaboration and accelerating advancements in the field.

3.5. Conclusion

Despite the challenges, creating a comprehensive dataset of parallel algorithms using the BSP model is a well-worth endeavor. By leveraging innovative data acquisition strategies, rigorous quality control measures, and collaborative efforts, we can establish a valuable resource for training our machine learning model. The potential benefits for both research and development, make this pursuit a crucial step in paving the way for a new era of universal and efficient automatic parallel algorithm design.

The next section will delve deeper into our multi-stage methodology for utilizing this dataset to automatically generate efficient and portable parallel algorithms.

4. MULTI-STAGE METHODOLOGY FOR AUTOMATIC PARALLEL ALGORITHM GENERATION

Our proposed approach breaks down automatic parallel algorithm generation into a meticulously crafted multi-stage process, seamlessly weaving together the strengths of the Bulk Synchronous Parallel (BSP) model, Contextual Flow Graphs (XFGs), and Graph Neural Networks (GNNs). Each stage serves a distinct mission and takes us closer to our ultimate goal: bridging the gap between sequential code and its efficient BSP-parallelized counterpart.

4.1. Stage 1: Data Acquisition and Preparation

- **Building a Diverse Sequential Code Dataset:** We gather a rich repository of sequential code samples spanning diverse domains and problem sizes. This foundational dataset serves as the raw computational material for training the GNN and ensuring its generalizability across various scenarios.
- **BSP-parallelized Counterparts Acquisition:** Corresponding to each sequential code sample, we acquire pre-existing BSP-parallelized versions, meticulously vetted for correctness and efficiency. These pre-existing BSP-parallelized versions act as crucial benchmarks for the GNN, guiding its learning process towards generating high-quality parallel algorithms.
- **Extracting BSP Communication Patterns:** From each parallelized algorithm, we extract its corresponding BSP Superstep Graph (BSPG). This graph structure captures the intricate dance of communication and computation within each superstep, providing invaluable insights for the GNN to model and optimize communication patterns.
- **Target Architecture Characterization:** We gather specific details about the parallel architectures for which we aim to generate algorithms. This includes processor specifications, network topology, and memory access patterns. By factoring these architectural nuances into the training process, we ensure the generated algorithms are tailored for optimal performance on the target hardware.

4.2. Stage 2: XFG Analysis - Unveiling the Potential for Parallelism

- **XFG Construction:** We construct Contextual Flow Graphs (XFGs) from the sequential code examples. These directed acyclic graphs represent the control flow, data dependencies, and computational units within the code, laying bare its inherent potential for parallel execution.
- **Identify independent tasks:** Utilize graph traversal algorithms in tools like Papyrus, XLang, or NetworkX to find independent subgraphs, representing potential parallel tasks.
- **Detect shared data:** Leverage data flow analysis capabilities in Papyrus, XLang, or PyTorch Geometric to track data flow and dependencies, highlighting shared variables and potential synchronization points.

- **Analyze loop structures:** Analyze loop iterations for parallelization potential using dedicated functionalities in XLang or custom algorithms based on XFG structures.
- **Determine synchronization points:** Analyze data dependencies and potential conflicts in tools like Papyrus, XLang, or NetworkX to suggest synchronization locations for data consistency.
- **Minimize communication overhead:** Employ graph partitioning techniques in PyTorch Geometric or develop custom algorithms based on identified task dependencies and communication patterns to optimize task distribution and reduce communication overhead.

4.3. Stage 3: XFG Augmentation and GNN Input

- **Enriching the XFGs:** Insights gleaned from the XFG analysis are used to enrich the original XFGs. This includes:
 - **Node Features**
 - Flag for independent tasks.
 - List of shared variables.
 - Loop characteristics (e.g., iteration count, parallelization potential).
 - **Edge Features**
 - Communication cost between tasks.
 - Data dependency strength.
 - Synchronization requirement.
- **GNN Input Preparation:** The enriched XFGs are further processed and transformed into a format suitable for input to the GNN. This involves encoding structural information, extracted insights, and target parallel architecture specifics into appropriate vectors and adjacency matrices, enabling the GNN to effectively understand and process the code and its parallelization potential.

4.4. Stage 4: Model Selection and Training

- **GNN Architecture Design**
 - Choose a GNN architecture capable of processing augmented XFGs, such as Graph Convolutional Networks (GCNs) for local feature extraction or Message Passing Neural Networks (MPNNs) for global information propagation.
 - Integrate attention mechanisms to focus on critical dependencies and relevant code sections for parallelization.
- **Training and Optimization**
 - Train the GNN on triplets of:
 - Augmented XFGs
 - Target parallel architectures
 - Corresponding BSPGs
 - Design custom loss functions to optimize for desirable BSP characteristics (minimizing communication overhead, reducing supersteps, balancing workload).

4.5. Stage 5: Output Representation and Generation

- **BSPG Decoding:** The GNN's output, a novel BSPG, encodes the structure and communication patterns of the generated parallel algorithm. We develop a dedicated decoding mechanism to translate this information into platform-specific parallel code.
- **Leveraging Existing Tools:** To expedite code generation and potentially optimize further, we integrate existing BSP-based frameworks and optimization tools into the final stage. This allows for seamless translation of the generated BSPG into efficient and platform-specific code.

5. CONCLUSION

This paper has presented a novel approach to automatic parallel algorithm design, leveraging the Bulk Synchronous Parallel (BSP) model, Contextual Flow Graphs (XFGs), and Graph Neural Networks (GNNs). Our multi-stage methodology offers a promising roadmap for bridging the gap between sequential code and its efficient BSP-parallelized counterpart, aiming to democratize access to high-performance computing across diverse domains.

Through rigorous data acquisition, XFG-based parallelism identification and optimization, and GNN-powered algorithm generation, our approach addresses the limitations of existing automatic parallelization techniques. The incorporation of the BSP model ensures predictable and portable algorithm execution, while the GNN learns from ground truth examples to generate efficient and accurate parallel algorithms.

While this research constitutes a proposed framework and is not yet implemented, it lays a solid foundation for future development. The multi-stage methodology provides a clear roadmap for implementation, outlining the key steps and required resources. Future work will focus on building a comprehensive dataset of BSP algorithms, refining the GNN architecture and training process, and integrating automatic code generation and optimization tools.

6. 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. Brelloch, 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. L. Chen, Q. I. Mahmud and A. Jannesari, "Multi-View Learning for Parallelism Discovery of Sequential Programs," 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lyon, France, 2022; 295-303. doi: 10.1109/IPDPSW55747.2022.00059.
8. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107-113.
9. Tournavitis, G., Wang, Z., Björn, F., Michael F. P. O'Boyle: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *PLDI 2009*; 177-187.
10. González, J. A., León, C., Piccoli, F., Printista, M., Roda, J. L., Rodríguez, C., & de Sande, F. (2000). *Oblivious BSP*. Springer, Berlin, Heidelberg.
11. 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.
12. Goodrich, M. T. (1999). Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2): 416-432.
13. 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.
14. Holl, J., McColl, B., Goudreau, M., et al. (1998). Standard: BSPLib: the BSP Programming Library. *Parallel Computing*, 24(14): 1947-1980.
15. 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
16. 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.
17. 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).
 18. 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
 19. 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
 20. 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>.
 21. Suijlen, W. J. (2007). BSPonMPI. <http://bsponmpi.sourceforge.net>.
 22. 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
 23. Valiant, L. G. (1990). A bridging model for parallel computation. Communications of the ACM, 33(8): 103-111.
 24. Valiant, L. G. (2011). A bridging model for multi-core computing. Journal of Computer System Science, 77(1): 154-166.
 25. Williams, T. L., & Parsons, R. J. (2000). The heterogeneous bulk synchronous parallel model. Springer, Berlin, Heidelberg.
 26. Yuanyuan Shen, Manman Peng, Shiling Wang, Qiang Wu. Towards parallelism detection of sequential programs with graph neural network. Future Generation Computer Systems, 2021; 125: 515-525, ISSN 0167-739X.