Scaling Graph Traversal to 281 Trillion Edges with 40 Million Cores

Huanqi Cao
caohq18@mails.tsinghua.edu.cn
Department of Computer Science and Technology & BNRist
Tsinghua University
Beijing, China

Yuanwei Wang
wangyw20@mails.tsinghua.edu.cn
Department of Computer Science and Technology & BNRist
Tsinghua University
Beijing, China

Haojie Wang
wanghaojie@tsinghua.edu.cn
Department of Computer Science and Technology & BNRist
Tsinghua University
Beijing, China

Heng Lin
linheng@pku.edu.cn
School of Computer Science
Peking University
Beijing, China

Zixuan Ma
ma-zx19@mails.tsinghua.edu.cn
Department of Computer Science and Technology & BNRist
Tsinghua University
Beijing, China

Wanwang Yin
26700402@qq.com
National Supercomputing Center in Wuxi
Wuxi, Jiangsu, China

Wenguang Chen
cwg@tsinghua.edu.cn
Department of Computer Science and Technology & BNRist
Tsinghua University
Beijing, China

Abstract

Graph processing, especially high-performance graph traversal, plays a more and more important role in data analytics. The successor of Sunway TaihuLight, New Sunway, is equipped with nearly 10 PB memory and over 40 million cores, which brings the opportunity to process hundreds of trillions of edges graphs. However, the graph with an unprecedented scale also brings severe performance challenges, including load imbalance, poor locality, and irregular access of graph traversal workload.

To address the scalability problem, we propose a novel 3-level degree-aware 1.5D graph partitioning, which benefits from both delegated 1D and 2D partitioning. By delegating extremely heavy vertices globally and other heavy vertices on columns and rows in the processes mesh, we break the scalability wall of previous partitioning methods. Together with sub-iteration direction optimization, core group -aware core subgraph segmenting, and a new on-chip sorting mechanism using RMA, we achieve 180,792 GTEPS on a graph with 281 trillion edges, using 103,912 processors with over 40 million cores, achieving 1.75× performance and 8× capacity compared to the previous state of the art and conforming to the Graph 500 BFS benchmark[14].

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508403

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**.

Keywords: massively parallel algorithm, breadth-first search, heterogeneous architecture

1 Introduction

Graph processing has the potential to solve critical data analytics problems across different scenarios, including financial risk management, epidemic trajectory analysis, protein sequence prediction, search engines ranking, knowledge graphs, etc., and becomes more and more important. As the graph scales up, graph processing is recognized as a challenging problem due to the access irregularity, lack of locality, and inherent load imbalance[12, 13, 19, 26]. This paper focuses on Breadth-First Search (BFS), a basic graph traversal algorithm regarded as one of the most representative graph workloads. Meanwhile, BFS is proposed by the Graph 500 Benchmark [14] as a computation kernel to evaluate the data analysis ability of different machines, especially supercomputers. Graph 500 runs BFS on a synthetic small-world graph with 2^S vertices and 16×2^S edges when configured with SCALE S.

Current supercomputers are primarily designed to adapt typical HPC workloads, usually floating-point intensive scientific and engineering computing applications. While supercomputers enable unprecedented capacity and performance for typical HPC workloads, the scalability issue of data-intensive workloads in terms of capacity and performance remains unanswered. The skewness of graph data, which causes severe load imbalance and unnecessary communication, is one of the most challenging problems in running the BFS algorithm efficiently on supercomputers. It requires well-designed graph partitioning methods to address the issue.

Researchers have proposed various solutions in partitioning the graph, including heavy vertices delegation on 1D partitioning [7, 17, 17], and 2D partitioning [8, 21]. A vanilla 1D partitioning contiguously assigns intervals of vertices to the nodes and assigns adjacent edges to where the vertex is, based on which vertices with higher degrees are delegated on every process. 2D partitioning is equivalent to delegating all vertices on rows and columns in the virtual mesh of processes. They both aim at resolving load balance and redundant communication issues at scale, but in different ways. We list the previous records and corresponding partitioning methods in Table 1.

While successful at their problem scale, those choices on graph partitioning would face scalability issues at the unprecedented large graphs. Methods based on 1D partitioning would require too many heavy vertices to be delegated globally. Methods based on 2D partitioning similarly have too many vertices on rows and columns to delegate. The emerging large graphs and supercomputers raise new challenges on scalable graph partitioning methods.

Irregular workloads such as graph processing also pose significant challenges to heterogeneous architectures widely adapted by recent supercomputers. Regarding Graph 500 BFS top 10, Lin et al. [12] created the only record with heterogeneous architecture on Sunway TaihuLight. It fell behind the K Computer, even with a more powerful hardware system. It requires more effort to optimize graph processing on heterogeneous architectures.

This paper targets Breadth-First Search (BFS) optimization, conforming to Graph 500 benchmark specification[14], on New Sunway, a new supercomputer with over 40 million cores and about 10 PB main memory, the successor of Sunway TaihuLight[11]. This supercomputer is equipped with the successor of SW26010, a new model with 390 heterogeneous cores in SW series many-core chips, namely SW26010-Pro. It replaces the previous register communication with a new inter-core communication mechanism called Remote Memory Access (RMA), allowing intra- Core Group (CG) peer-to-peer communication. SW26010-Pro consists of 6 CGs, each with 64 Computing Processing Elements (CPEs), which provide the main computing power of the chip. SW26010-Pro has optional Local Data Cache (LDCache) in accelerator cores. The vast main memory of the machine makes it possible to tackle SCALE 44 in Graph 500 benchmark, which means 281 trillion edges.

To efficiently implement a massively parallel BFS on New Sunway, we propose four novel techniques as follows:

3-Level Degree-Aware 1.5D Graph Partitioning. Many real-world graphs expose extremely skewed degrees, causing load imbalance and redundant communication in graph processing. The R-MAT graph generating algorithm employed in the Graph 500 benchmark also simulates such features. Graph partitioning methods targeting extreme-scale graph traversal have been proposed to address those problems, including degree-aware 1D partitioning and 2D partitioning. As mentioned above, both the partitioning methods have specific problems limiting the problem size. We thus propose 3-level degree-aware 1.5D graph partitioning. The vertices in the graph are divided into three levels of degree. Vertices with the highest degree level, identified as Extremely heavy (E), are delegated on all nodes. These in the second level, namely Heavy (H), are delegated only on columns and rows of the communication mesh, following the delegation strategy in 2D partitioning. Other Light (L) vertices are treated the same as 1D partitioning. This novel partitioning method ensures scalability and reduces communication.

Sub-Iteration Direction Optimization. Direction optimization [1, 2] has been a critical technique for optimizing BFS. It involves two kinds of iterations: top-down and bottom-up. In dense iterations, bottom-up is used instead of conventional top-down. We observe that due to degree skewness, hub vertices including E and H are intensively visited earlier than vertices with lower degrees. To this end, we propose sub-iteration direction optimization, applying different directions on different degree-aware subgraphs. It allows us to efficiently visit E and H by bottom-up in early iterations while preventing us from iterating all of L. It also eliminates unnecessary E or H visits from L vertices in late iterations by applying bottom-up in this case.

CG-Aware Core Subgraph Segmenting. The E and H vertices form a core subgraph, usually containing over 60% edges in Graph 500 generated graphs, whose computation deserves further optimization. Hot spot analysis shows one most significant part is the largest iteration on the core subgraph in the bottom-up direction, which random reads all E and H vertices on the communication column. Inspired by previous graph segmenting techniques [20, 23, 26] optimizing against cache or NUMA, we propose segmenting the subgraph by source, stored by the destination index, and scheduling the segments processed on the six core groups (CGs) simultaneously in SW26010-Pro. In bottom-up, the activation bit vector of a single segment can thus be distributed to 64 acceleration cores and fit into the fast on-chip memory of a CG. Utilizing RMA to read the bit vector from other cores in the same CG, we achieve 9× higher performance on that kernel.

Table 1. Results of recent works on large-scale distributed BFS.

Authors	Year	Num. Edges	GTEPS	Num. Proc.	Num. Cores	Arch.	Part. Method
Checconi [7]	2014	17.6T	15,363	65,536	1.05M	Blue Gene/Q	1D with heavy delegates
Ueno [21]	2015	17.6T	38,621.4	82,944	663.5K	SPARC64 VIIIfx	2D
Lin [12]	2016	17.6T	23,755.7	40,768	10.6M	SW26010	1D with heavy delegates
Nakao [15]	2021	35.2T	102,956	158,976	7.6M	A64FX	2D
Our Work	2021	**281T**	**180,792**	103,912	**40.5M**	SW26010-Pro	**degree-aware 1.5D**

On-Chip Sorting with RMA. Bucket sort is required as a multi-purpose meta-kernel in messaging by edges, as was pointed out in ShenTu[13]. Based on the RMA mechanism, we design On-Chip Sorting with RMA (OCS-RMA), serving the needs of sorting random messages into buckets. OCS-RMA divides cores into producers and consumers and uses RMA to send batched messages from producers and consumers. Replacing the similar meta-kernel on SW26010 in ShenTu, OCS-RMA achieves higher bandwidth utilization at 47.0% thanks to the new flexible on-chip communication.

Combining all the efforts mentioned, we ultimately achieved 180,792 giga-traversed edges per second (GTEPS), preceding the best result with 1.75× performance, which is of 102,956 GTEPS on the June 2021 Graph 500 BFS list. It also shows unprecedented graph size with 281 trillion undirected edges, 8× to the current largest on the Graph 500 list. This result fully conforms to the Graph 500 specification.

2 Background and Motivation

2.1 BFS Algorithm and Optimizations

We will discuss the parallelization and optimization of BFS in this subsection. Algorithm 1 presents a basic BFS.

2.1.1 Graph partitioning and parallel BFS. Researchers have proposed various methods to parallelize the nested loops in line 10. We visualize the two typical partitioning and parallelization strategies in Figure 1.

Algorithm 1: BFS Algorithm

```
1  fn BFS(G(V,E), root):
2      Curr ← ∅ ;                    // Current active frontiers
3      Next ← root ;                 // Vertices to be visited next
4      Prt[:] ← −1, Prt[root] ← root ;   // Parent array
5      while Next ≠ ∅ do
6          Curr ← Next; Next ← ∅;
7          BFS-Iteration-Top-Down(E, Curr, Next, Prt);
8      return Prt;
9  fn BFS-Iteration-Top-Down(E, Curr, Next, Prt):
10     forall u ∈ Curr do
11         forall v ∈ {v|(u,v) ∈ E} do
12             if Prt[v] = −1 then
13                 Prt[v] ← u;
14                 Next ← Next ∪ v;
```

In 1D partitioning, only the outer loop is parallelized, yielding a graph partitioning by only the source vertices. Each process owns vertices in a contiguous interval. The processes send a message to the owner process of v to update neighbor v from each local vertex. While simple to implement and works well with direction optimization (see Section 2.1.2), it has significant load balancing issues due to the extreme skewness in vertices degree distribution. Degree-aware methods [7, 17] are thus applied to solve this. Delegates are created on each node for the vertices with higher degrees, namely heavy vertices. As is shown in Figure 1 (a), edges between heavy vertices and normal ones are instead connected to the local delegate, distributing adjacency lists of heavy vertices globally. It effectively solves the load imbalance at scale.

2D partitioning was first introduced to graph traversal by Yoo et al.[22]. In 2D partitioning, the processes construct a virtual mesh of size $R \times C$. Vertices are assigned to an owner the same as in 1D partitioning and logically delegated on rows and columns. It thus connects edges to a source delegate on the row and a destination delegate on the column. We visualize the delegation strategy in Figure 1 (b). In implementations, the adjacency matrix is partitioned by both rows and columns in a block-cyclic flavor. When applied to the algorithm, this partitioning parallelizes both levels of loops. The outer loop on the current frontier is parallelized on each column of the processor mesh. The inner loop on neighbors, following the graph partitioning, is parallelized on each row. The communication is thus limited on columns and rows during traversal and can be accelerated through efficient collective operations in MPI.

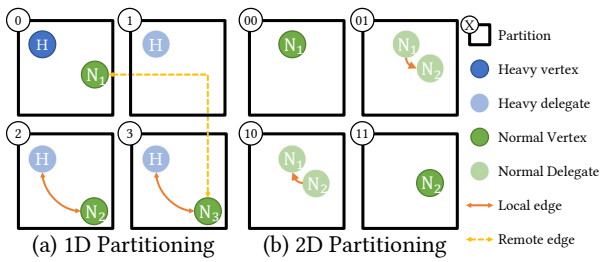

Figure 1. Existing graph partitioning methods for BFS. Numbers in the top-left corner illustrate process numbering, sequential in 1D, and columns with rows in 2D.

Algorithm 2: Bottom-Up BFS Iteration

```
1 fn BFS-Iteration-Bottom-Up(E, Curr, Next, Prt):
2     forall v ∈ {v ∈ E|Prt[v] = −1} do
3         forall u ∈ {u|(u, v) ∈ E} do
4             if u ∈ Curr then
5                 Prt[v] ← u;
6                 Next ← Next ∪ v;
7                 break;
```

2.1.2 Direction optimization.

Direction optimization is proposed by Beamer et al.[2] and is widely used in implementing BFS efficiently. It is based on the observation that when the frontier is large, traversing the graph reversely from v to u (namely bottom-up) instead of conventional direction (top-down, shown in Line 10 of Algorithm 1) requires fewer edges to be touched. Also, when an unvisited v has found a reversed neighbor in frontier, other incoming edges can be skipped, which we call early exit. The iteration in bottom-up is shown in Algorithm 2. State-of-the-art works on parallel BFS all leverage this technique, despite whether 1D or 2D partitioning is employed. Nevertheless, to our best knowledge, 1D partitioning methods have to drop or limit the early exit optimization: inter-node messaging requires batching messages, preventing it from performing the finest grain early exit. More heavy vertices help reduce the disadvantage.

2.2 Graph 500 Benchmark

Complementary to Top 500[10], Graph 500[14] establishes a benchmark for large-scale data-intensive workloads. It was first announced in November 2010 and has gained wide acceptance. It includes two typical graph applications, breadth-first search (BFS) and single-source shortest path (SSSP), as problems for the benchmark. Among the two, BFS has a more extended history as a benchmark kernel and is also our focus in this paper. The two kernels run on an undirected randomly generated graph. Graph 500 specifies an algorithm for the graph generation. It is called R-MAT[6], which recursively manipulates non-zeros on the adjacency matrix. The specified configuration is $A = 0.57, B = C = 0.19, D = 1 - (A + B + C) = 0.05$, with an edge factor of 16. The yielded graph simulates real-world graphs, exposing extremely skewed degree distribution. Despite the skewness, it is also highly discrete: multiple hypergeometric distributions centered at numerous peaks construct the whole degree distribution, as is shown in Figure 2. Adopting the above graph generator, we are limited by this phenomenon in the tuning of thresholds, which we will discuss in Section 6.2.1.

2.3 Parallel BFS on Larger Graphs

Works based on 1D and 2D partitioning have scaled to over trillion vertices and tens of trillions edges. They are not yet sufficient if we head to larger graphs. 1D partitioning

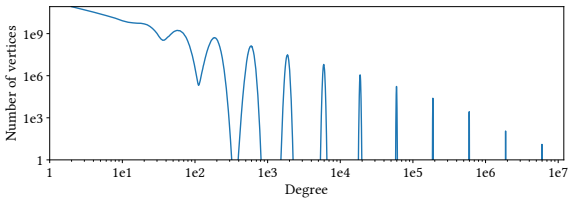

Figure 2. Degree distribution of a Graph 500 specified synthetic graph at SCALE 40.

requires about 0.1% vertices to be treated as heavy ([12] uses 32768 per 27 million), or the performance would degrade due to load imbalance and less early exit. In our case, to achieve SCALE 44 in Graph 500 benchmark, it would require each node to handle $2^{44} \times 0.1\% \approx 1.76 \times 10^{10}$ vertices, which is unacceptable given the main memory size of 96 GiB. Even if we manage to store the vertices with proper compression, it is still a huge impact on performance to collect them through interconnect. For 2D partitioning, a similar problem occurs on column and row vertex sharing. The number of shared vertices on column and row is around $|V_{local}| \times \sqrt{P}$, in which V_{local} is the vertices number per process and P is the process count. In our case, it goes even further to 5.56×10^{10}.

Given the above observation, new parallel partitioning and algorithm need to be proposed to tackle BFS on larger graphs with hundred-trillions of edges.

3 Challenges on New Sunway Architecture

The Newest Generation Sunway Supercomputer (abbrev. New Sunway) is a supercomputer with heterogeneous architecture. We will discuss its processor, large-scale interconnect, and the challenges they raise in this section.

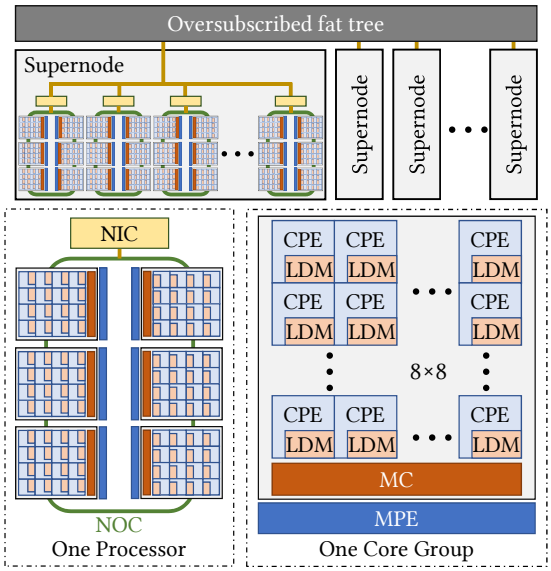

Figure 3. Architecture overview of New Sunway

3.1 The SW26010-Pro Processor

The system is equipped with over 100,000 SW26010-Pro many-core chips. SW26010-Pro is the successor of SW26010, which is used in Sunway TaihuLight. We discuss the unchanged and changed designs, respectively.

3.1.1 Core designs inherited. We list below the important designs SW26010-Pro inherits.

On-chip heterogeneous architecture. The SW many-core series processors are equipped with Management Processing Elements (MPEs) and Computing Processing Elements (CPEs). The MPEs are full-functioned RISC cores with lower computing power, responsible for resource management, I/O, networking, etc. Each Core Group (CG) consists of 64 CPE cores responsible for high-performance computation. Specific to SW26010-Pro, it consists of 6 MPEs and 6 CPE CGs, 50% more than SW26010.

Local Data Memory (LDM) and Direct Memory Access (DMA) for CPEs. While CPEs can access data in memory directly with load and store instructions, they are designed to do this through DMA in and out from LDM. The LDM is a piece of scratchpad memory dedicated to each CPE core. The CPEs can initiate asynchronous DMA requests, copy chunks of data between main memory and LDM. The measured peak bandwidth of the whole chip is 249.0 GB/s. Good bandwidth utilization can be exploited through large enough DMA grain sizes. On the contrary, direct access to main memory, equivalent to uncached memory access on usual architectures, is marginally slower. We name such direct access GLD (Global Load) and GST (Global Store).

3.1.2 New features. As the successor of SW26010 after multiple years, SW26010-Pro also introduces various significant changes. We discuss the two most relevant aspects.

Remote Memory Access (RMA). SW26010-Pro provides a new inter-core communication mechanism called RMA. It enables intra-CG communication through direct LDM access. It employs a typical one-sided communication interface, including "get" from and "put" to another CPE's LDM. RMA has significantly lower latency and higher throughput than the main memory as an on-chip communication mechanism.

Local Data Cache. Instead of only DMA, SW26010-Pro can now efficiently fetch data from the main memory through Local Data Cache (LDCache). LDCache is an optional feature that user programs can easily reconfigure at runtime. It shares physical space with LDM and serves as a data cache for loads and stores to the main memory address.

Given the powerful inter-core communication, applications usually do not require atomic operations for synchronization. SW26010-Pro provides only functionally essential atomic instructions, with similar inefficiency to SW26010. In the absence of a shared cache, SW26010-Pro can only accomplish atomic operations through inefficient direct accesses to main memory.

3.2 Interconnect

New Sunway is equipped with 200 Gbps interconnection in each processor. The processors are then connected via an oversubscribed fat-tree network. Every 256 nodes form a supernode, and inside the supernode, the communication is unblocked. Inter-supernode communication, on the contrary, has lower available bandwidth.

3.3 Challenges on Graph Traversal

The upgraded unique architecture and large scale interconnect introduce multiple challenges to the BFS design:

Inefficient atomic instructions. During remote edges processing, manipulated messages are appended into a corresponded buffer for later sending, requiring atomic append; when updating vertices in top-down traversal in BFS, random writes to bit vectors also require an atomic set bit. We should avoid the inefficient atomic instructions. Therefore we need to utilize inter-core communication mechanisms to ensure performance.

Inefficient random access. Random accesses to main memory on CPEs are inefficient due to the lack of a large shared cache. With LDCache disabled, accessing main memory involves one GLD/GST, resulting in bad performance. With LDCache enabled, the cache size is also not large enough to hold the hot data given millions of vertices each node is responsible for.

Inefficient inter-supernode communication. The network topology exposes lower inter-supernode bandwidth. To minimize communication across supernodes, a topology-aware partitioning method is needed.

Previous works on Sunway TaihuLight[12, 13] have proposed various techniques targeting the challenges. Yet, new techniques are required due to additional challenges of the larger graph and upgraded machine.

4 Methodology

4.1 3-Level Degree-Aware 1.5D Graph Partitioning

As is mentioned, different partitioning strategies can be seen as different delegating strategies, unifying the view of degree-aware 1D partitioning and 2D partitioning. We will express our partitioning with delegates for a clear view.

The processes are organized into a $R \times C$ virtual mesh, as is in 2D partitioning. The rows are mapped to supernodes in our case. We first classify vertices into three levels of degrees: Extremely heavy (E), Heavy (H), and Light (L), with degrees from high to low.

Delegate *E* vertices globally. Vertices with extremely high degrees are expected to touch neighbors on nearly every node during the traversal. For those vertices, creating delegates on all nodes helps reduce communication.

Delegate *H* vertices on column and message outgoing edges along the rows. Vertices with a medium level of degrees tend to touch neighbors on every supernode during

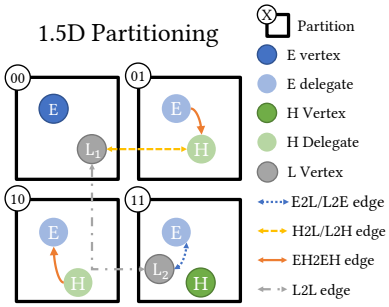

Figure 4. Partition strategies via delegation.

the traversal. Those vertices are significantly more than *E*, leading to a large amount of unnecessary communication if delegated globally like *E* vertices. Considering the tiered network topology, not creating delegates leads to repeated data flowing across supernodes, creating a network bottleneck. Inspired by the delegation strategy of 2D partitioning, we delegate those vertices on rows and columns. It helps to eliminate replicated send to the same supernode while avoiding the costly global delegation.

No delegate for *L* vertices. An *L* vertex has a small number of edges connected to it, in which rarely multiple destinations reside on the same process. Delegating them offers little profit but only takes time and space managing the delegates. We thus leave them connected to other vertices with remote edges. These remote edges require per-edge messaging when accessed during traversal.

The classification between *E, H, L* involves two thresholds on degrees. It is worth noticing that the number of neighbors touched by a vertex during traversal is only softly constrained by its degree. With the aforementioned direction optimization presenting, changing direction by heuristics makes the number of adjacent edges to be accessed vary. Thus, instead of using the number of supernodes directly, the degree thresholds are subjected to further tuning, which we will discuss in Section 6.2.1.

In our implementation, the *E* and *H* vertices are selected out of all vertices, sorted per node by the degree, and given a new ID among the higher degree vertices. The rest vertices are *L*, remaining original vertex IDs. We split the original edge set into six components according to the vertices classification. Since *E* and *H* have delegates on column and row, the subgraph with both ends being *E* or *H* (*EH2EH*) is 2D-partitioned. The two subgraphs from *E* to *L* (*E2L*) and *L* to *E* (*L2E*) are attached with *L*'s owner, just as heavy vertices in degree-aware 1D partitioning. Similarly named, we have *H2L* distributed on the column of the owner of *H*, restricting the messaging of edges intra-row. *L2H* stores solely on the owner of *L*, as a reverse of *H2L*. Finally, *L2L* is the most naive component, just as original 1D partitioning. The resulting partitioning can be visualized as is in Figure 4. Each subgraph is well balanced between nodes, even at full scale. We will present the balancing of this partitioning method in Section 6.2.2.

With $|H| = 0$, our approach degenerates to a partitioning similar to 1D with heavy delegates, except that edges between heavy vertices are 2D-partitioned. Compared to 1D partitioning with heavy delegates, our approach isolates the heavy vertices further into two levels, resulting in a topology-aware data partitioning. It retains sufficient heavy vertices for a better early exit in direction optimization. It also avoids the communication for globally sharing all those heavy vertices by delegating *H* only on rows and columns.

With $|L| = 0$, it degenerates to 2D partitioning with vertex reorder. Compared to 2D partitioning, our approach avoids the inefficient delegation on lower degree (*L*) vertices, eliminating the $O(|V_{local}|\times\sqrt{P})$ space limitation of 2D partitioning. It also constructs global delegates for the super-heavy (*E*) vertices, reducing communication.

4.2 Sub-Iteration Direction Optimization

Direction optimizing BFS is to switch to the bottom-up direction when the frontier is large. Usually, only two to three densest iterations are subjected to bottom-up. With vertices split by degree, we notice that the densest activation time for different degree levels is different. The vertices with higher degrees (*E* and *H*) tend to be activated earlier, while *L* vertices are usually densely activated in later iterations. A Kronecker graph specified by the Graph 500 benchmark shows a typical activation breakdown in BFS in Figure 5.

Conforming to this observation, we propose to isolate the direction selection of different subgraphs. In each iteration, the traversing direction is selected individually with different heuristics, allowing different directions to be applied to different levels of degrees. For subgraphs involving edges crossing nodes (*H2L*, *L2H*, *L2L*), we estimate optimal direction by comparing the ratio of active source vertices and unvisited destination vertices among all vertices in their class. The ratios directly reflect the number of messages required to communicate. Unlike those, only the source active ratio is used to select the direction for subgraphs with node-local edges (*EH2EH*, *E2L*, *L2E*). The consideration is that the

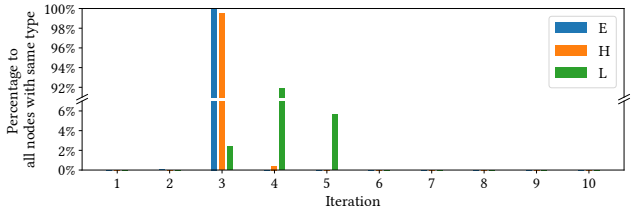

Figure 5. Active Vertices Percentage for Sub-Iterations. Some iterations have bars invisible due to being magnitudes smaller.

pull workload can hardly be estimated by destination unvisited ratio due to early exit, which is available only for node-local edges. With sub-iteration direction optimization, we can start bottom-up on the *EH2EH* core subgraph much earlier without dragging the mostly unvisited *L* vertices into the bottom-up procedure.

Sub-iterations on components with higher-degree source and destination will be executed earlier in a single iteration. The bottom-up procedure on any subgraph takes the latest visited status, possibly updated by former subgraphs in the same iteration. By doing so, we avoid activated vertices being pulled. Thus, the direction selection procedure uses the latest unvisited count to conclude a strategy for each sub-iteration after the previous is done. It helps the direction optimization make a more accurate choice. For example, *L2E* and *L2H* will choose bottom-up for fewer edges to be touched once after a dense *EH2EH* sub-iteration has activated nearly all *E* and *H*.

Thanks to degree-aware partitioning, the subgraphs explicitly split by degree provide the basis for sub-iteration direction optimization. Through this optimization, we can further reduce the edges to be touched in BFS beyond vanilla direction optimized BFS. We will discuss the performance impact in Section 6.4.

4.3 CG-Aware Core Subgraph Segmenting

As the densest subgraph, the *EH2EH* subgraph is considered the *core subgraph*, whose computation is first-class for optimizing. And in the heaviest iterations, the iteration direction is expected to be bottom-up, focusing on pulling optimization on this core subgraph. The bottom-up procedure on the core subgraph involves local random read from destination vertices and sequential write to source vertices. Due to lack of locality, the random read is of poor performance. We notice that the read range is limited: the working footprint is only column *E* and *H* activeness bit vector. Limited by local storage, we expect the total *E* and *H* vertex count assigned to nodes in each column to be no larger than 100M, resulting in a bit vector of size smaller than 12.5 MB.

The limited size provides an insight to fit it into the LDM. To achieve this without hurting LDM space usable for other

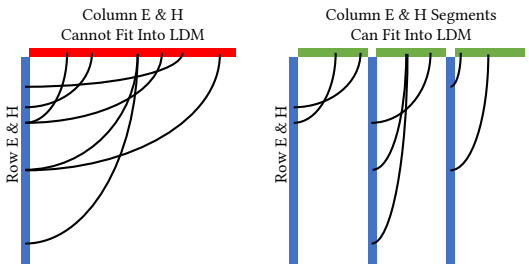

Figure 6. CG-Aware Subgraph Segmenting. Here only three segments are rendered just for visualization; the actual number corresponds to the number of CGs (i.e., 6).

Figure 7. Offset mapping for destination vertex offset in a segment. Each cell represents a bit in the offset.

data fetching, we segment the core subgraph by destination into six pieces, each for a CG, as is shown in Figure 6. The destination range of each segment then corresponded to about 2 MB bit vector. While it still does not fit into single-core LDM of 256 KB, we can distribute it over the 64 CPEs in one CG and utilize the RMA feature to access it. To map the contiguous range of the activeness bit vector to LDM, we split the bit vector into lines of 1024 bytes. Lines are round-robin scheduled between 64 CPEs. The resulting offset mapping is shown in Figure 7. With such mapping, we can efficiently manipulate the CPE number and the local address in LDM through bit operations for a vertex to access. Thus we interchange GLD from main memory with RMA "get" from other CPEs with much lower latency.

The six CGs process the six core subgraph segments in parallel. It raises a question on the concurrent safety of writing to source vertices. To resolve it, we further divide the source vertices into six virtual intervals and schedule them to the CGs in a round-robin flavor. Each CG dedicates to one segment but only processes one of the intervals once. Through synchronizing across CGs, we guarantee that multiple CGs never process the same interval.

4.4 On-Chip Sorting with RMA

The *H2L*, *L2H*, and *L2L* subgraphs require messaging by remote edges. Each accessed remote edge requires a peer-to-peer message, either on the row in *H2L* and *L2H* or globally in *L2L*. We can efficiently utilize the network bandwidth by batching the messages by destination and using alltoallv from MPI for communication. To efficiently utilize the main memory bandwidth, we further fuse message generation and bucket sorting into one kernel, avoiding redundant memory access for writing back the messages and sorting. It raises a requirement for a generic message sorting kernel.

Conventionally parallel bucket sort either requires atomic operations per message or redundant main memory accesses. Buffering the messages to write is also inefficient, given the grain size requirement ($> 1KB$) of DMA multiplied with the hundreds of buckets would require a total buffer size exceeding LDM capacity. To avoid those inefficient operations, we utilize the on-chip communication mechanism, RMA, to build an on-chip sorting algorithm, presenting OCS-RMA. The 64 cores are divided into 32 producers and 32 consumers. Each consumer is responsible for a group of buckets. Bucket x is assigned to consumer $x \mod 32$. Each core reserves 32 buffers of 512 bytes for sending or receiving messages. Messages generated on producer i sending to consumer j

will first be buffered in the jth local send buffer and be sent through an RMA-put operation to the ith receive buffer of consumer j. The messages are thus sorted into different consumers, available for exclusive operations while avoiding inefficient atomic memory access. This kernel is shown in Figure 8.

Previous works on graph processing on Sunway[12, 13] also cover similar bandwidth-efficient sorting kernels. Compared with SW26010, which allows only column and row communication, RMA in SW26010-Pro allows arbitrary pairs of cores to communicate. Hence, we can use all cores as producers or consumers without introducing other roles to do routing. The new on-chip sorting mechanism yields significantly better performance than the previous on SW26010.

The OCS-RMA serves as a generic kernel template in our implementation and is used for multiple purposes:

Message generation. As is mentioned above, message generation by remote edge requires sorting the generated message by destination, which is the most straightforward use case.

Forwarding in global messaging. In *L2L* messaging, global peer-to-peer communication requires a hierarchical implementation to achieve better network bandwidth utilization and less active RDMA connections. Instead of a traditional hierarchical alltoallv, we do manual forwarding on the intersection node of the source column and destination row. The source node only sorts messages by which forwarding node to go. The forwarding nodes are responsible for sorting them by destination. OCS-RMA also covers this message sorting.

Two-stage sorting in destination updating. After messages arrive at the destination node, it requires random writes to the vertices array to update the destination. As GST and atomic operations are inefficient on SW26010-Pro, we coarse sort the messages into fixed-length ranges, then iterate over the ranges and update in LDM. The first stage is a straightforward bucket sort. In the second stage, updating over a range of vertices, we again on-chip sort it into 32 sub-ranges and make each consumer responsible for updating over one of them.

The above usages of on-chip sorting are similar to the works on Sunway TaihuLight[12, 13]. But unlike those works, we do not introduce a messaging pipeline connecting different sorting steps. Instead, we implement the messaging procedure synchronously to utilize all 6 CGs better and more balanced. Thus, it requires synchronizing across CGs (but not in the same CG), involving atomic operations that rarely conflicts. The bandwidth utilization of OCS-RMA on 6 CGs is thus expected to be slightly lower than a single CG. We will cover the performance details in Section 6.3.

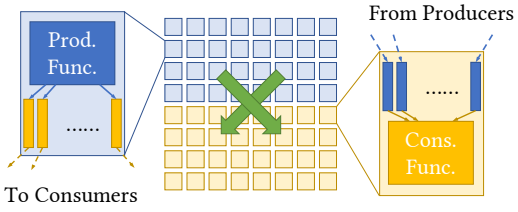

Figure 8. On-Chip Sorting with RMA (OCS-RMA). Blue for producers and yellow for consumers.

5 Implementation

Alongside the above-mentioned techniques, we also introduce auxiliary designs to achieve efficient extreme-scale graph traversal.

In-place global sort. The 3-level degree-aware 1.5D graph partitioning requires complicated preprocessing to construct the final data structure from the edges list. It requires in-place preprocessing to preprocess a graph nearly occupying all the main memory. To unify the in-place splitting and construction of all 6 subgraphs, we abstract the core process into generic in-place global sort. We implement it based on Parallel Sorting by Regular Sampling[18], with local sort implemented with PARADIS[9].

Delayed reduction of delegated parent array. Traditionally parents collected by delegates are reduced after each iteration. Due to our partitioning dramatically limiting the number of local delegates, it is possible to persist the parent array locally. Thus we can delay the reduction of delegated parent array until the BFS run finishes. It does not change the result of graph traversal: only frontier information is required for later iterations of the traversal. It significantly reduces collective communication volume during the BFS run.

Edge-aware vertex-cut load balancing in *EH2EH* push. Usually, we observe a tremendous amount of E and H vertices visited by only a small fraction of E and H vertices in the second or third iteration. In local top-down computation, simply cutting by active vertices results in severe load imbalance between CPEs, while there are enough edges to fulfill the parallelism. To achieve load balancing in this case, we adopt the edge-aware vertex-cut method proposed by GraphIt[25]. We first calculate the prefix sum of locally available frontier vertices' degree at each *EH2EH* top-down traversal. Given the frontier size is small in a top-down iteration, this will not cost much. Then we divide the frontier by accumulated degrees, generating a balanced workload for each CPE. It provides reasonable performance in the CPE implementation of *EH2EH* top-down.

6 Evaluation

In this section, we first discuss the overall performance with the scaling, then validate core techniques, including graph partitioning and on-chip sorting with RMA. The core design relies on the 3-level degree-aware 1.5D graph partitioning,

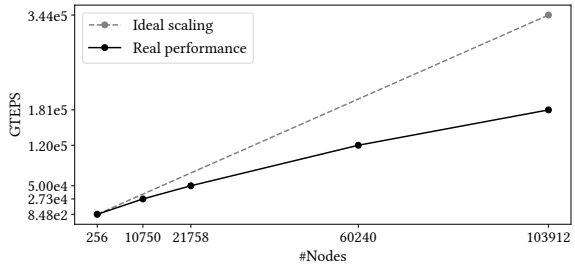

Figure 9. Weak scalability.

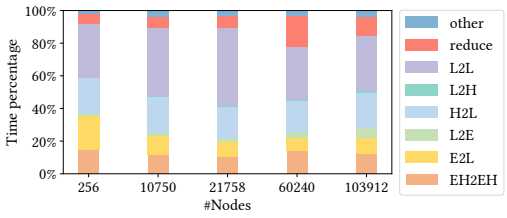

Figure 10. Time breakdown for the scaling runs, categorized by subgraphs and parent reduction.

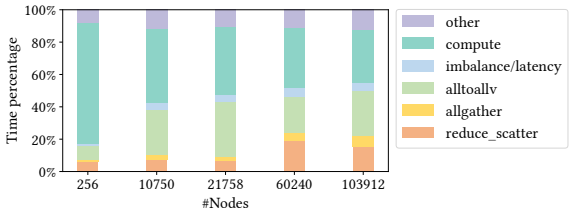

Figure 11. Time breakdown for the scaling runs, categorized by different communication types and computation.

and our implementation extensively uses the on-chip sorting meta-kernel to implement messaging. Thus, being impractical to test without the two core techniques, we exclude the two techniques from the later discussion on techniques impacts. Impacts on other techniques focus on sub-iteration direction optimization and core subgraph segmenting.

6.1 Overall Performance and Details

Our implementation of BFS traverses a Graph 500 Specification conforming randomly generated graph at SCALE 44 with $2^{44} \times 16 \approx 281$ trillion edges in 1.55 seconds on average of 64 random roots. It yields 180,792 GTEPS (giga- traversal edges per second) on 103,912 nodes, with 40.5 million cores. The result is validated according to Graph 500 Specification 2.0. We here discuss the weak scalability and time consumption of each part in our algorithm.

6.1.1 Weak scalability. We present the weak scalability in Figure 9. The maximum possible SCALE is selected for each test, being 35 and 41-44, respectively. We made our best effort to tune E and H degree thresholds. It shows 52% relative parallel efficiency at the largest scale over a single supernode, proving the effectiveness of our approach on graph partitioning. Traditionally communication is considered the bottleneck of graph algorithms. With the peer-to-peer communication pattern, performance is expected to hurt a lot due to the 8× fat-tree oversubscription in the interconnect of New Sunway. But with our 3-level degree-aware 1.5D partitioning, we greatly reduce the network traffic crossing supernodes, avoiding the bottleneck in the top-level tree network. It provides a strong basis for the scalability of our BFS implementation.

6.1.2 Execution time breakdown. We then present the time breakdown for the scaling. In Figure 10, time components include different subgraphs, delayed reduction of parent array, and other unrecognized time like barrier cost, etc. We see that *L2L* costs notable time while being the smallest subgraph. The extremely sparse access pattern makes it the most inefficient to process. *L2L* involves nearly all the iterations and usually shows extremely low parallelism due to the super small frontier at sparse iterations. As the hardware and software system is not optimized for latency, a lower efficiency is expected. The *EH2EH* subgraph, which is the largest one, takes a notably shorter time at larger scales, showing the effectiveness of the optimizations, including reduced communication thanks to the partitioning and better work efficiency thanks to sub-iteration direction optimization. We will discuss these in detail in Section 6.4. Also, when the scale bounces from 21758 nodes to 60240 nodes during the scaling, the percentage of *L2L* drops. The discrete degree distribution of Graph 500 generated graph and tuning of the H degree threshold contributes to this phenomenon.

In Figure 11, we categorize the time consumption by computation and communication types, including alltoallv, allgather, and reduce-scatter. The total time for communication routines increases during scaling. As the scale goes up, time consumption for communication increases as expected. The main source of communication cost is from alltoallv and reduce-scatter. As discussed in the breakdown by subgraphs, the tuning of the H degree threshold introduces the expansion of delegation cost (hence reduce-scatter) and the shrinking of remote edge messaging cost (hence alltoallv) between 21758 and 60240 nodes. Thanks to our proposed partitioning method, the load imbalance mixed with barrier latency keep constant at larger scales, showing good load balance over scaling.

6.2 The Graph Partitioning

6.2.1 Selection of degree thresholds. In Figure 12, we present the impact of degree thresholds on BFS performance. The results are tested on 256 nodes with Graph 500 generated graph at SCALE 35, with a 16×16 mesh. As is mentioned in Section 2.2, the degree of vertices in an R-MAT graph

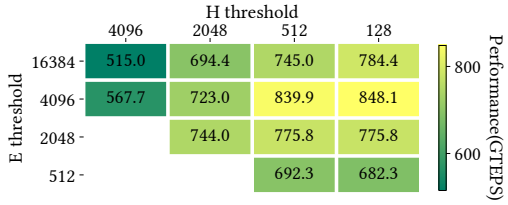

Figure 12. BFS performance at SCALE 35, 256 nodes with different degree thresholds.

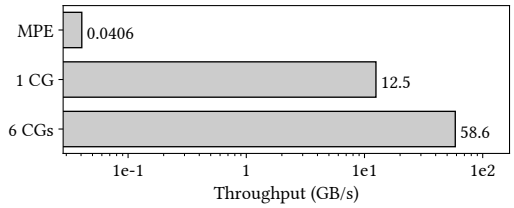

Figure 14. Throughput of different bucketing implementations.

distributes around several peaks. It limits the selection of the H and E degree thresholds. Only thresholds between the peaks are meaningful, and anywhere between two peaks performs identically. Thus we evaluate 128, 512, 2048, 4096 for H threshold and 512, 2048, 4096, 16384 for E threshold, conforming to the degree distribution at SCALE 35.

The first key observation is that even with the number of nodes only 256 such that the network oversubscription is absent, the existence of H vertices brings performance improvement. Sub-iteration direction optimization allows using different directions for H and L, yielding fewer edges to be accessed with H presented. The second is that threshold for E impacts performance a lot: it simultaneously influences communication and accessed edges.

We use 262144 and 2048 for E and H thresholds in the 180792 GTEPS test at the largest scale. Due to budget concerns, we do not grid-search over different thresholds.

6.2.2 Load balance on 281 Trillion Edges. In this subsection, we discuss the load balance of our approach on graph partitioning at the largest scale. We partition an R-MAT synthetic graph with 281 trillion edges and 17.6 trillion vertices to 103912 nodes. Vertices are first evenly distributed across nodes. Edges are distributed according to the proposed 1.5D partitioning method in Section 4.1. The resulting degree distributions of the 6 subgraphs are demonstrated in Figure 13. Comparing the nodes with minimum and maximum size, we

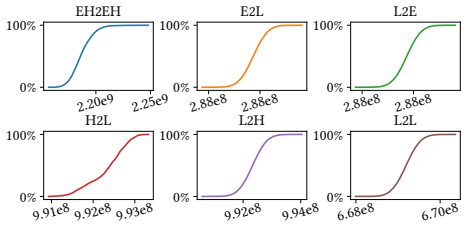

Figure 13. Distribution of partitioned subgraphs sizes. Plots shown are cumulative distribution functions of per-partition edges amount in each subgraph, with horizontal axes being edge counts and vertical axes being percentage of partitions that has less edges than the specified count in the specified subgraph.

see a 4.2% difference in *EH2EH* and up to 0.35% difference in the rests. Comparing maximum against average, we get 2.8% in *EH2EH* and up to 0.17% respectively. We observe that the resulting subgraph size distribution is already well balanced even without adjusting the vertex distribution over nodes. The result shows superior balancing in edge distribution, which gives a guarantee on the load balancing.

6.3 Performance of On-Chip Sorting with RMA

In implementing BFS, OCS-RMA is used for various types of messages with different byte lengths. Our OCS-RMA implementation leverages C++ templates to reuse the code. In this subsection, we evaluate the performance of OCS-RMA.

To test the performance of on-chip sorting with RMA, we set up a microbenchmark bucketing 4 GB uniformly random 64-bit integers by lower 8 bits. The throughput with 1 MPE, 1 CG (64 CPEs), and 6 CGs (384 CPEs) are listed in Figure 14. The MPE test uses sequential implementation on a single MPE. The single CG implementation is based on on-chip sorting with RMA, which fully eliminates atomic instructions with the exclusiveness guarantee provided by OCS. The 6 CGs implementation, also based on OCS-RMA, uses atomic instructions to synchronize across CGs, resulting slightly lower efficiency. With 6 CGs, we can achieve a 58.6 GB/s throughput, indicating 117.2 GB/s memory bandwidth utilization (one read and one write per message) out of the 249.0 GB/s peak. It provides 1443× speedup against MPE implementation. Compared to the on-chip sorting on SW26010 in ShenTu[13], OCS-RMA improves memory bandwidth utilization from 33.7% (9.76 GB/s of 28.9 GB/s) to 47.0%.

6.4 Impacts of Other Techniques

We analyze the impacts of the other two techniques, sub-iteration direction optimization and core subgraph segmenting, by measuring the time breakdown before and after applying the techniques. Time consumption is broken into five parts, including Push (a.k.a. Top-Down) and Pull (a.k.a. Bottom-Up) for *EH2EH* and other subgraphs, along with other procedures. It can help us determine the time consumption changing with different direction optimization strategies (whole-iteration or sub-iteration). Also, we can see the performance influence on *EH2EH* Pull of core subgraph segmenting.

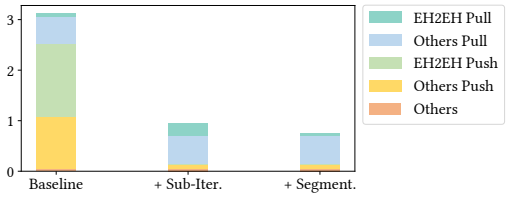

Figure 15. Time breakdown for different levels of optimization. **(a) Baseline:** vanilla direction optimization, no core subgraph segmenting; **(b) + Sub-Iter.:** with sub-iteration direction optimization, no core subgraph segmenting; **(c) + Segment.:** with both sub-iteration direction optimization and core subgraph segmenting.

The result in Figure 15 is measured with a graph of SCALE 35 on 256 nodes, averaging over multiple runs. Applying sub-iteration direction optimization, we significantly reduce the time consumption of pushing E and H related subgraphs, replacing them with the lower cost in pulling. Further applying core subgraph segmenting, we observe 9× speedup in pulling *EH2EH* alone, further optimizing the end-to-end cost. At this scale, *EH2EH* pulling costs little. But with tens of thousands of processors, it grows larger, as we have seen in the time breakdown during weak scaling.

7 Related Work

With the increasing attention on unstructured data in HPC and Big Data research and industry, works have been done over years optimizing graph traversal. Direction-Optimized BFS by Beamer et al. [1, 2] exploited the low-diameter property of real-world graphs and showed significant performance advantages, becoming the algorithm basis of future BFS implementations.

Parallel partitioning is a critical design point in graph processing. As early research, Yoo et al. [22] proposed an efficient block-cyclic variant of 2D partitioning, eliminating the communication required by vector transpose. Buluc et al. [4] thoroughly discussed and evaluated the performance of vanilla 1D partitioning and 2D partitioning. Checconi et al. [8] further proposed efficient communication methods for 2D partitioning on BlueGene. They later [7] turned to 1D partitioning to adopt direction optimization and proposed a load balancing method equivalent to the heavy delegation. The resulting implementation achieved 16599 GTEPS on 65536 BlueGene/Q nodes. Pearce et al. [17] formally proposed distributed delegate partitioning, what we call 1D partitioning with heavy delegation. The "delegate" concept inspires us to go more general. A similar partitioning method was adopted by Lin et al. on Sunway TaihuLight [12], achieving 23755.7 GTEPS. In a different direction, Ueno et al. [21] extended the 2D partitioning method to support direction optimization. They achieved 38621.4 GTEPS, ensuring the 5-year long world championship of the K-Computer on Graph 500 BFS List. The same implementation further achieved 102956 GTEPS on Fugaku, which is the best record on Graph 500 BFS List to this date, reported by Nakao et. al [15].

Aside from the specific BFS implementation, Lin et al. also announced a general-purpose graph processing system on Sunway TaihuLight, namely ShenTu[13]. It splits the graph explicitly by vertex degree, inspiring us to push it further to 3 levels and 6 subgraphs. It also shapes the generic on-chip sorting for edge messaging, the spirit of which is inherited by our OCS-RMA.

Graph segmenting has been widely used in NUMA-aware graph processing, including [20, 23, 26]. It is also used in cache blocking/partitioning, improving cache hit in sparse linear algebra[16] or graph processing[24]. GraphIt[25] unifies the two optimizations with Segmented Subgraphs (SSGs). While the Core Groups architecture differs highly from NUMA and the shared cache is absent, the LDM with RMA can serve similar to the Last Level Cache. It inspires our CG-aware core subgraph segmenting.

8 Discussion

In this work we share our techniques in implementing efficient BFS on New Sunway. While our current implementation of BFS is ad-hoc to the algorithm, a general-purpose graph processing framework is possible to be built with the proposed techniques: **3-level degree-aware 1.5D partitioning** is a graph partitioning method neutral to the graph algorithm to run on. Also, it is designed for any graph with extremely skewed degree distribution, which is commonly found in social networks, web graphs, etc., and we expect it to work with those real-world graphs. **Sub-iteration direction optimization** is a BFS-specific optimization, but the push-pull selection behind it works on many graph algorithms, including SSSP[5], PageRank and more[3]. The idea to select difference iterating direction has the potential to gain performance improvement on other algorithms. **CG-aware core subgraph segmenting** requires tuning on number of segments to adapt more algorithms. More segments ensures sufficiently small destination footprint, allowing the RMA optimization. **On-chip sorting with RMA** implements an efficient bucket sort on SW26010-Pro. It is directly available for general-purpose graph processing. One of our future work will be designing and implementing the next-generation ShenTu[13] on New Sunway upon the proposed techniques.

Although only implemented on New Sunway, the first two techniques, 3-level degree-aware 1.5D partitioning and sub-iteration direction optimization, are applicable to other architectures as well. The two techniques are algorithm-level designs, instead of tailored against the architecture. The concepts of aggressive degree-aware partitioning and selecting direction by different degrees may help the design of future graph processing systems on other architecture.

9 Conclusion

We share our experience and methodology designing the highly scalable parallel BFS algorithm on New Sunway, the latest supercomputer in the Sunway series. We present a novel technique on graph partitioning, 3-level degree-aware 1.5D partitioning, to address the graph traversal at unprecedented machine scale and graph size. Together with sub-iteration direction optimization and other machine-specific techniques, we ultimately run BFS on a graph with 281 trillion edges, on 103,912 nodes with 40.5 million cores, achieving 180,792 GTEPS. This result outperforms the first place on Graph 500 June 2021 BFS List with 1.75× performance and 8× capacity.

Acknowledgments

This work was partially supported by National Key Research & Development Plan of China under grant 2017YFA0604500 and NSFC U20B2044. The corresponding author is Wenguang Chen.

References

[1] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[2] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1618–1627.

[3] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.

[4] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[5] Venkatesan T Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. 2016. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2016), 2031–2045.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[7] Fabio Checconi and Fabrizio Petrini. 2014. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 425–434.

[8] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. 2012. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[9] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. 2015. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1518–1529.

[10] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. TOP500 supercomputer sites. *Supercomputer* 13 (1997), 89–111.

[11] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (2016), 1–16.

[12] Heng Lin, Xiongchao Tang, Bowen Yu, Youwei Zhuo, Wenguang Chen, Jidong Zhai, Wanwang Yin, and Weimin Zheng. 2017. Scalable graph traversal on sunway taihulight with ten million cores. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 635–645.

[13] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 706–716.

[14] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[15] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. 2021. Performance of the Supercomputer Fugaku for Breadth-First Search in Graph500 Benchmark. In *International Conference on High Performance Computing*. Springer, 372–390.

[16] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. 2007. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing* 18, 3 (2007), 297–311.

[17] Roger Pearce, Maya Gokhale, and Nancy M Amato. 2014. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 549–559.

[18] Hanmao Shi and Jonathan Schaeffer. 1992. Parallel sorting by regular sampling. *Journal of parallel and distributed computing* 14, 4 (1992), 361–372.

[19] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[20] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[21] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. 2017. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering* 2, 1 (2017), 22–35.

[22] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 25–25.

[23] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*. 183–193.

[24] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[25] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[26] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 301–316.