HPC – SSSP

Stanisław Bitner

June 8, 2025

HPC SSSP

Implementation

The single-source shortest path (SSSP) implementation is based on the delta-stepping algorithm.

Communication

The MPI library is used for inter-process communication. In addition to standard reductions, the algorithm requires a method to exchange shortest path updates. Several approaches were considered, but the most efficient strategy proved to be using `MPI_Alltoall` and its variants, particularly `MPI_Alltoallv`.

To implement this communication strategy, the algorithm doesn't relax edges immediately but instead collects updates in a buffer before exchanging them between processes. Since buffer sizes may vary, the `MPI_Alltoallv` variant is necessary.

This function behaves similarly to scattering followed by gathering buffers. Its implementation depends on the MPI library and, as testing showed, is significantly more efficient than alternative strategies for processes on the same node versus different nodes.

Optimizations

The implemented optimizations include *Edge Classification*, *IOS*, *Push-Pull Pruning*, and *Hybridization*. *Load Balancing* was not implemented.

Edge Classification

Edge classification was implemented for all versions as it can be done without overhead. The classification involves maintaining separate arrays for *short* and *long* edges. Initially, only *short* edges are relaxed in multiple phases, followed by relaxing all the *long* edges in a single phase.

Note that *Edge Classification* is already part of the baseline implementation, which explains why the difference between baseline and IOS performance is minimal.

IOS

IOS (Inner/Outer Short edges) improves upon *Edge Classification*. During the short edge phase for vertex u, only edges with weights $w \in [0, (k+1)\Delta - \delta(u))$ are considered, where k is the current bucket index and $\delta(u)$ is the tentative distance from the source to vertex u. This divides short edges into inner and outer categories.

Since the tentative distance changes during execution, we cannot simply split edges into two arrays. Instead, edges are sorted by weight to efficiently process only relevant edges during each phase. For inner edges, processing starts from the beginning and stops when weights exceed the range. Similarly, outer edges are processed from the end during their phase.

Push-Pull Pruning

When Δ is sufficiently small, the long edge phase dominates execution time. To reduce communications in this phase, we observe that long edges can only reduce distances to vertices in later buckets. Thus, instead of pushing updates from the current bucket to later buckets, it may be more efficient to pull them. Each vertex in a later bucket can query whether it's connected to the current bucket via a long edge and receive updates accordingly.

We must approximate which strategy minimizes communications. The push model always performs $\sum_{u \in B_k} \deg_{\text{long}(u)}$ communications. The pull model performs at most $2 \cdot \sum_{u \in B_{\text{later}}} |\{(u, v, w) \mid v \in B_k \ \text{ and } \ w \in [(k+1)\Delta - \delta(v), \delta(u) - k\Delta)\}|$ communications.

Since $\delta(v)$ is unknown but $v \in B_k$ implies $\delta(v) < (k+1)\Delta$, we can bound the pull communications by $2 \cdot \sum_{u \in B_{\text{later}}} |\{(u, v, w) \mid v \in B_k \ \text{ and } \ w < \delta(u) - k\Delta\}|$. Without *IOS*, short edges can be skipped entirely in the pull model.

The push communication volume is straightforward to compute. For the pull model, since edges are sorted by weight, binary search can efficiently determine the communication volume. To choose the better strategy, we compute and globally reduce the difference between push and pull volumes, saving one global reduction operation.

Hybridization

The hybridization optimization switches to the Bellman-Ford algorithm after processing $\tau \cdot |V|$ vertices. This is implemented by setting Δ to infinity and moving all later buckets into B_0.

Load Balancing

This optimization was not implemented.

Testing and Tuning

Data

Testing was performed on rmat-1 and rmat-2 type graphs with nonnegative weights drawn from a uniform distribution ($w_{\max} = 255$). Tests were conducted with 24, 48, 72, and 96 processes, corresponding to four nodes at maximum capacity (without hyperthreading). Graph sizes ranged from 2^{10} to 2^{22} vertices, with $|E| = 16|V|$. Multiple graphs were generated for each size and type, totaling approximately 180GiB of data.

Timing Methodology

For each combination of parameters (graph size, graph type, process count and so on), the results presented below are based on the average of 5 independent runs, with a 95% confidence interval.

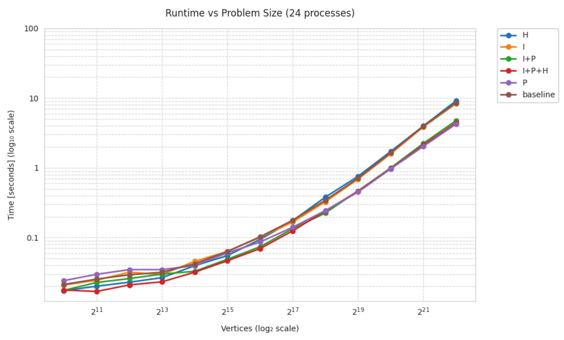
Choosing Configuration

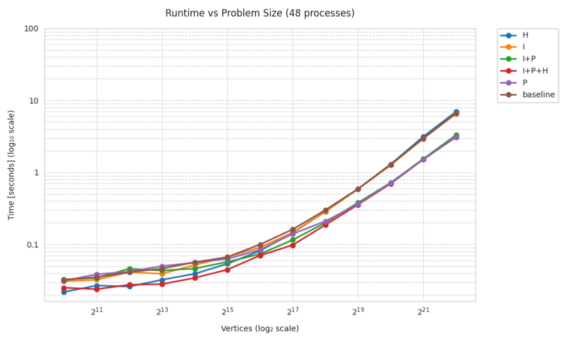
Initial testing aimed to identify the best-performing optimization configuration using $\Delta = 25$ and $\tau = 0.4$ as suggested in the paper. The optimal configuration (OPT) included all implemented optimizations.

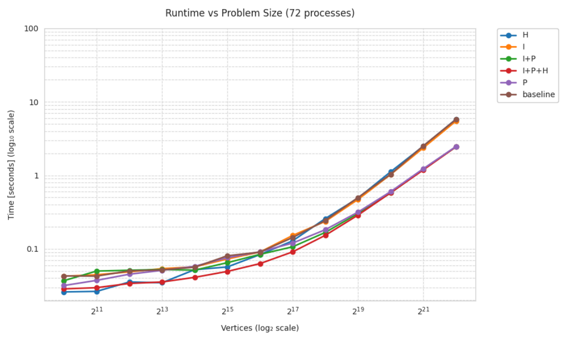
Legend abbreviations:

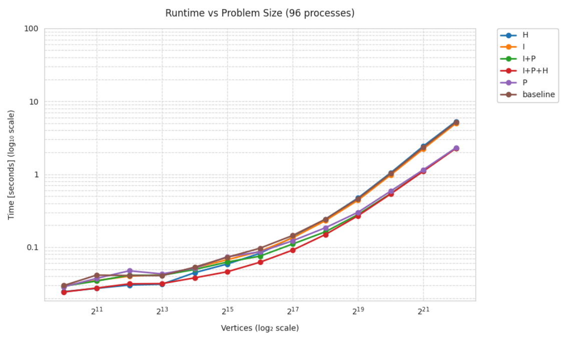
- `baseline`: No optimizations
- `I`: Edge classification with inner/outer short edges
- `P`: Push-pull pruning

- `H`: Hybridization



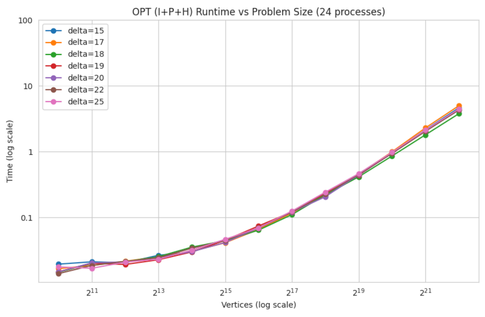


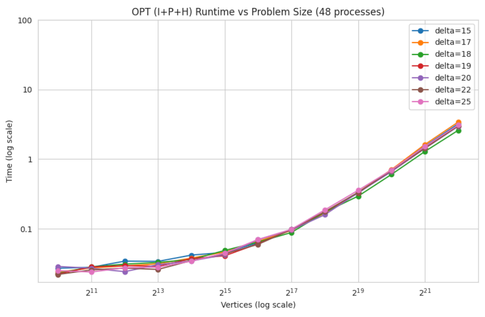


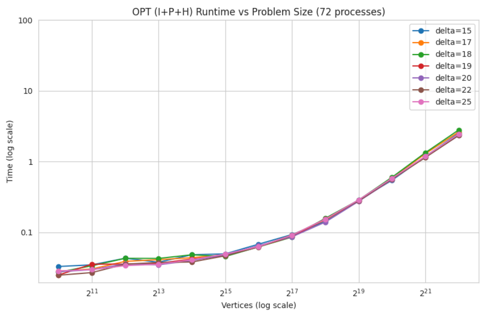


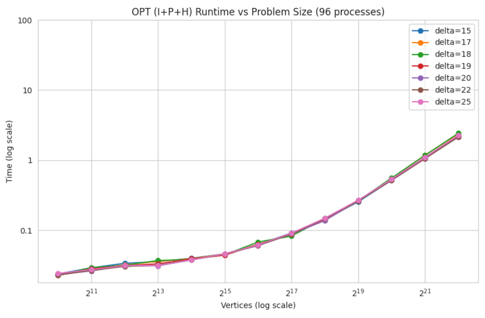
Tuning Δ

The Δ parameter was tuned using a gradient-descent-like approach. Starting with $\Delta = 25$, values were adjusted in decreasing steps. The optimal values were $\Delta = 18$ or $\Delta = 20$, with $\Delta = 20$ performing slightly better for larger process counts and chosen as the final value.



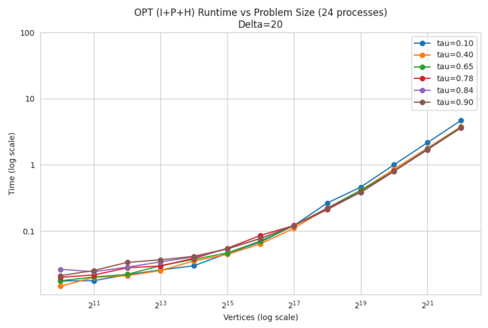


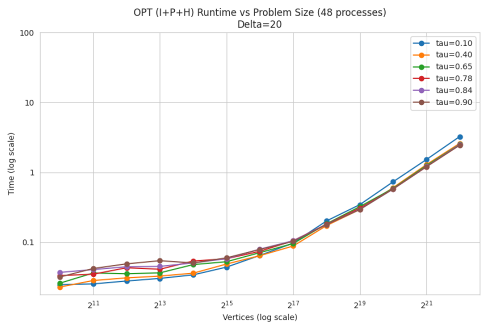


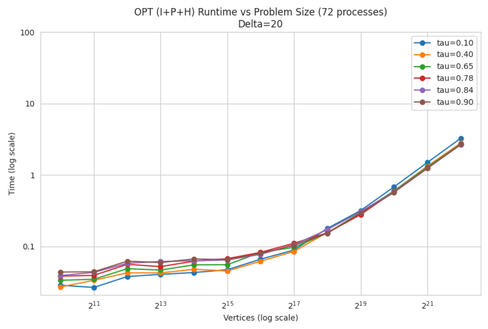


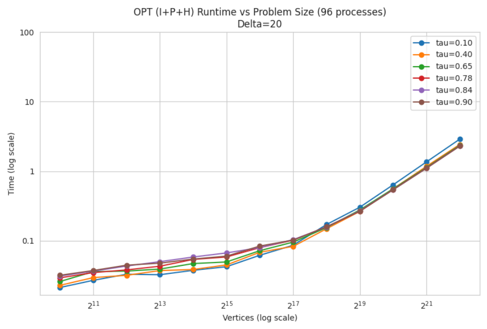
Tuning τ

The τ parameter was tuned using binary search. Surprisingly, the best performance occurred at $\tau = 0.90$, significantly higher than the paper's suggestion. This led to a suspicion that it might be best to disable *hybridization* completely. However, testing without *hybridization* showed worse performance, confirming the value of this optimization.







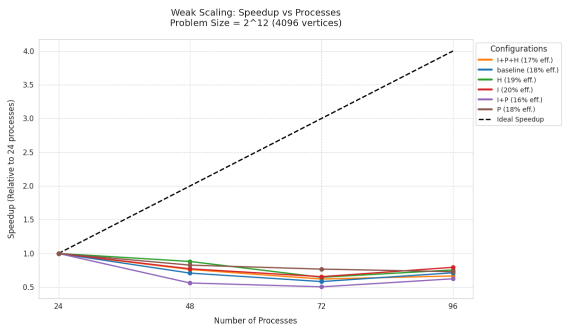


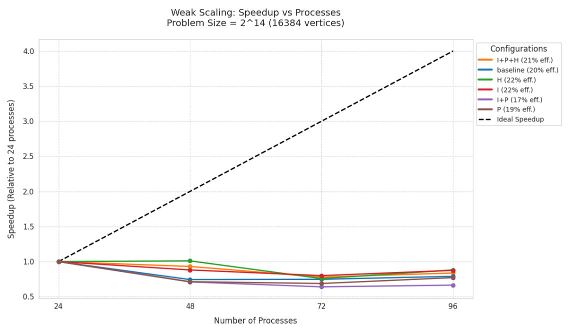
Final Configuration

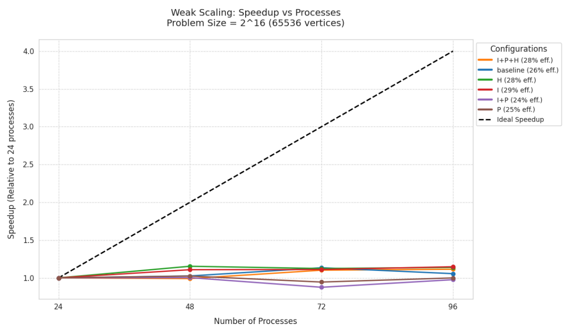
Benchmarking revealed the optimal configuration as all optimizations enabled with $\Delta = 20$ and $\tau = 0.90$. The configuration can also be manually changed via the `Makefile`.

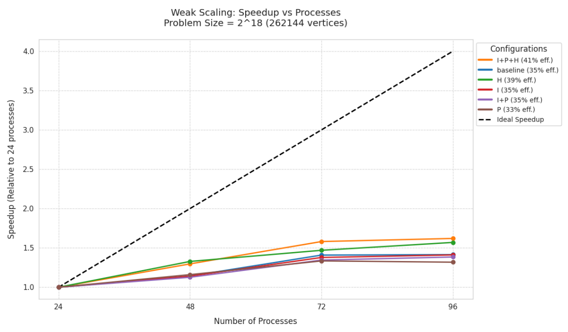
Weak (Gustafson) Scaling

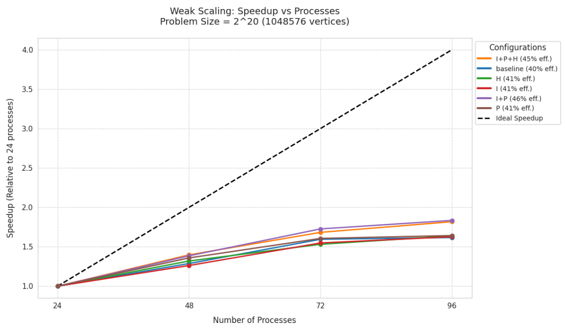
The weak scaling results for each configuration are shown below:

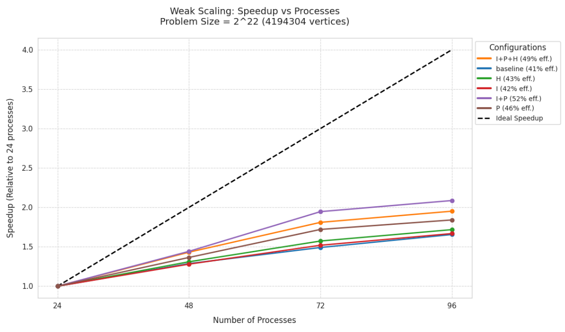












The speedup is sublinear, particularly for smaller graphs. However, scaling improves with larger graph sizes, suggesting approximately 50% efficiency could be achieved for graphs of sizes around 2^{25} vertices.