Parallel Sorting and Deduplication for Large-Scale Text: An HPC Study

YU-WEI LIN
The University of Melbourne
Parkville, VIC, Australia
studentid@student.unimelb.edu.au

KEYWORDS

Parallel, Sorting, Deduplication, OpenMP, HPC

1 INTRODUCTION

In a data explosion era, large text workload need a rapid and clear way to detect misspelled words and pair it with good candidates of possible words. Each experiment we use a simple and reproducible setting: one dictionary and one word list with different amount of word count in each set. By lowercasing and hold only **ASCII** letters and digits. For a word that is label as **"present"** means it matches a dictionary token exactly; otherwise, it will be tagged as **"misspelled"**. For each misspelled word, we output all edit-distance-1 (ED=1) candidates by a single step of following three behaviour: deletetion, substitution, insertion (no transposition allowed).

The output content is deterministic, each line shows one misspelled word and its candidates in ASCII order. Lines apply a fixed golbal key: first by the number of candidates (few → many), then by the misspelled word in ASCII order. Thus, two executions with the same inputs and settings generate the same file, checking by a simple and supports regression tests **md5sum hash**.

Targetting parallel hardware and stay minimal in the design, we distribute key spaces by a $K-prefix$ and cap each rank at about $2/p$ of dictionary bytes(words), where p is the amount of the ranks. This limits hotspots and improves balance. Afterward, each rank build a prefix index for exact lookups. Then, we optionally build a global **Bloom filter** and **OR-reduce** its bits so all ranks possecess the same filter, enabling dicard numerous negatives before entering the networks communication and redues bytes in later phases.

Subsequently, three main phases pipeline will run end-to-end. **Phase A** routes words to owners and marks presents; only misspelled set would be kept. **Phase B** generates ED = 1 candidates per miss, checks Bloom, routes survivors to owners and verifies each candidate by exact lookingup; we return only true hits. **Phase C** groups by misspelled, deduplicaties, sorts candidates and writes the file in the global order. We use OpenMP for local loops and MPI `Alltoallv` with compact packing and zero-copy unpack.

2 ALGORITHMS & PIPELINE

2.1 Data Model and Invariants:

Inputs. Our approach involves two plain-text files: a *dictionary* and a *word list*. Each of the files consists of one token per line.

Normalization. The first thing we do is to convert all the tokens into lowercase ASCII characters. After that only letters and digits are retained (`[a-z0-9]`). Next, spaces and punctuation are eliminated. Throughout the process, the order of lines within each file remains unchanged.

Token rules. A token is defined as an alphanumeric span with no characters at either end. Every empty line results in a token count of zero. The word list can have words repeated. In the case of dictionary checking, we consider the dictionary to be a set.

Miss definition. A word is considered to be *present* if it is an exact match with a token from the dictionary. During the exact match process, a word is termed *misspelled* if it turns out to be not present.

Candidate definition (ED = 1). Given a misspelled word x, a candidate c is identified as any string which is one character away from x according to the ASCII edit distance: (1) one character substitution; (2) one character deletion; (3) one character insertion. Transposition, however, is not included. The said order is very rough and very stable. Even if the same file with the very same inputs and settings is produced, the two executions will still yield the same file.

Ownership and routing. The very first K base-62 symbols (K-prefix) of the strings are used for mapping strings to buckets. By applying a basic byte cap of **2/p** of dictionary bytes (with p being the number of ranks) to the mapping of buckets to ranks, we manage to keep the load distribution and thus prevent skew from occurring. The parameter K is limited by the constant `KMAX`.

Derived metadata (for balance, not for output). While partitioning is being done, a bucket–byte histogram over K-prefix buckets is computed: $ghist = \{\, bytes[b] \,\}_{b=1}^{B}$, where $bytes[b]$ is dictionary bytes in bucket b. Only its summaries $ghist_{\max}$, $ghist_{avg}$, $ghist_{std}$, and $ghist_{maxratio} = ghist_{\max}/ghist_{avg}$ are kept along with the chosen K, the number of buckets B, and the bucket→rank map. These guide load balance and later reporting. They do not affect the output.

Index and Bloom (resources). A prefix index is built over each rank's local dictionary slice to support exact lookup. This is done by rank and if Bloom is on, a global Bloom filter is created with $m = bpw \times N_d$ bits and k hashes. The `BLOOM_BITS_PER_WORD` (bpw) is set at runtime.

Memory and limits. We do not allow any extra copies. We pack the network payloads into contiguous byte buffers. ASCII output bucketing is limited by `KWRITE`. We apply the same compile flags for different runs.

Correctness invariants. (i) An exact match is performed beforehand for candidate generation (Phase A); (ii) Each candidate returned can be found in the dictionary after verify (Phase B); (iii) The output is globally ordered according to the fixed key (Phase C); (iv) The hash of the output is the same as that of the baseline when the inputs and settings are identical.

The goal of our practice is to presever the order above and output it in a text file.

Algorithm 1 Adaptive K-prefix partition with 2/p cap

```
Require: Global dictionary bytes $Fbytes$, number of ranks $p$, upper bound $K_{max}$ (env KMAX, default 4)
 1: $cap \leftarrow \lfloor 2 \cdot Fbytes/p \rfloor$
 2: $K \leftarrow 2$
 3: while $K \leq K_{max}$ do
 4:     Compute bucket bytes $bytes[b]$ for all $b$ under prefix length $K$
 5:     if $\max_b bytes[b] \leq cap$ then
 6:         break
 7:     else
 8:         $K \leftarrow K + 1$
 9:     end if
10: end while
11: Assign buckets to ranks by scan on $bytes[b]$ until each rank $\leq cap$
12: Fix empty ranks by moving light buckets to them (no-idle fix)
13: return $K$, bucket→rank table
```

Transition to Algorithms. First, we balance load with a K-prefix split under the $2/p$ cap. Next, we build the prefix index and the global Bloom. Then we run Phase A, Phase B, and Phase C in order.

2.1.1 *Adaptive K-prefix Partition.*

Start with distributing the key space under the K-prefix rule. Each string maps to a bucket sorted by the first K base-62 symbols. Next, we choose the number of the prefix K in order to fit under a $2/p$ cap to mitigate the burden of the heaviest bucket, where p is the number of the ranks. We commence with small K and raise it if any bucket size is heavily skew. Then, when the max bucket is under the cap, we stop. Subsequently, we assign buckets to ranks scanned by a pre-define prefix in previous step on bytes. Furthermore, the light buckets would be moved to any idle ranks if exists, resulting the load is even and the skew is small. Linearly, this step is associated with the data size which has low overhead. Complexity. Counting bytes per bucket is linear in the dictionary slice; the scan is linear in the number of buckets.

2.1.2 *Prefix Index and Global Bloom.*

We drop the most negatives before the network by implementing global bloom.

To begin with, each rank creates a prefix index over its partition of the local dictionary. The index enables quick exact lookups and a narrow search range. Then, in case Bloom is activated, we allocate the size of the bit array according to “bits per word” and insert all the local words. Afterwards, we perform **OR-reduction** of Bloom across ranks thus every rank possesses a common filter. Thus, we can eliminate a large number of negative probes from processing at an early stage and reduce the amount of data transmitted over the network. The expense incurred is directly proportional to the number of local words and the length of the bit array. Complexity. Building the index is linear in local words; Bloom insert is linear; the all-reduce is linear in bit-array size.

We use two *Alltoallv* calls per phase to separate requestsand returns.

2.1.3 *Phase A — Existence Filter.*

By routing each input word to its owner rank attached to $K - prefix$ rule, we use one *Alltollv* to send the routed words. Then, each owner will exact lookup in the prefix index and produces a 1-byte flag. Afterward, using another *Alltotallv* to return flags to sources. Ultimately, discarding all the words that are already in the dictionary and keep only misspelled or distinct words that aren’t in the dictionary.

Algorithm 2 Build prefix index and global Bloom

```
Require: Local dictionary slice $D_{local}$, Bloom bits-per-word $bpw$ (env BLOOM_BITS_PER_WORD), switch BLOOM (0/1)
 1: Build prefix index over $D_{local}$ for exact match and range limit
 2: if BLOOM = 1 then
 3:     $m \leftarrow |D_{global}| \times bpw$        ▷ determine Bloom bit count
 4:     Init Bloom($m$)
 5:     for each word $w \in D_{local}$ do
 6:         Bloom.insert($w$)
 7:     end for
 8:     Allreduce Bloom (bitwise OR) so all ranks share same Bloom
 9: end if
10: return prefix index, Bloom
```

Algorithm 3 Phase A: existence filter

```
Require: Word list $W$, bucket→rank table, prefix index
 1: Route each $w \in W$ to owner rank by K-prefix
 2: Alltoallv send routed words
 3: for each received $w$ do
 4:     flag ← prefix_index.contains($w$)
 5:     if flag = true then mark as present
 6:     end if
 7: end for
 8: Alltoallv return flags
 9: Keep only misspelled words $M \subseteq W$
10: return $M$
```

Thus, the next phases will focus on s samller set and avoids wasted work.

Notes. The dominate cost in the section is one forward and on reture *Alltoallv*, and fast local contains checks additionally.

2.1.4 *Phase B — ED1 Candidates: Generate, Filter, Verify.*

In the stagem, for each misspelled word, we generated all **edit-distance-1** candidates: delete, insert and substitute/

If Bloom is on and says "not present", we drop the candidate. Then, for the rest, each candidate would be routed to its owner again by the same $K - prefix$ rule and apply one *Alltoallv* to send candidate requests. Each owner does exact verify in the prefix index; only true hits are kept. Finally, utilising one *Alltoallv* to return the hits as (miss, candidate) pairs, which guarantees the trafic is ssmall and verify is correct.

Note. Bloom decrease the false requrest while prefix index keeps verify cheap.

2.1.5 *Phase C — Global Total-Order Output.*

Gathering results by the misspelled word and remove the duplicate candidates, we sort all the candidates inside each group for a stable view. After that , we compute an ASCII output bucket by the first **Kwrite** bytes of the misspelled word. Then, we route groups to their output

Algorithm 4 Phase B: ED1 generate, distributed filter, verify

```
Require: Misspelled set $M$, Bloom (optional), prefix index, bucket→rank
 1: for each $x \in M$ do
 2:     $C \leftarrow$ all edit-distance-1 candidates of $x$ (substitute/delete/insert)
 3:     for each $c \in C$ do
 4:         if Bloom enabled and Bloom.maybe($c$) = false then
 5:             drop $c$
 6:         else
 7:             route $c$ to owner by K-prefix
 8:         end if
 9:     end for
10: end for
11: Alltoallv send candidate requests
12: for each received $c$ do
13:     hit $\leftarrow$ prefix_index.contains($c$)
14:     if hit then push $(miss, c)$ to return buffer
15:     end if
16: end for
17: Alltoallv return hits
18: return verified pairs per misspelled word
```

Algorithm 5 Phase C: global total-order output

```
Require: Verified pairs $(miss, c)$, KWRITE for ASCII output buckets
 1: Group by key: for each $miss$, collect unique candidates; sort candidates
 2: Compute ASCII bucket id by first $KWRITE$ bytes of $miss$
 3: Map bucket to output owner; route records by bucket
 4: Alltoallv send grouped records
 5: for each received group do
 6:     Merge same $miss$; ensure unique candidates
 7: end for
 8: Compute local order key: $(|cand(miss)|, miss)$
 9: Local sort by this key
10: Write partitions in owner order; the global order is stable
```

owners and merge smae-key groups there. As aforementioned, all the misspelled words would be ordered from the fewest candidates first then follow the ASCII order, we sort all of them locally by a simple global key. At the end, owners write the partitions in a fixed orderm so the global output is stable and easy to check.

Note. THis order make calidation simple and makes runs reproducible.

2.1.6 ***Phase A — Existence Filter***.

Initially, the items are sorted according to their destination ranks and then packed into byte buffers. After that, size and displacement calculations are made, and an *MPI_Alltoallv* call is executed. Afterward, items are unpacked if zero-copy views are applicable so that no extra copies are made. Thus, the entire process is straightforward and the overhead is minimal.

Note. This technique is reused that applied to words, flags, requests, and hits as well.

Algorithm 6 Pack/Unpack for Alltoallv

```
Require: Vector of items with destination rank
 1: For each dest rank $r$, append items to send buffer $S[r]$
 2: Compute byte sizes; build displacements for Alltoallv
 3: MPI_Alltoallv(S → R)
 4: Unpack R by zero-copy views when possible
 5: return received items per source rank
```

3 METHODOLOGY

All builds use the same flags (e.g., `-O3 -fopenmp`). We control input size and thread count. We keep the tokenization and dedup parts the same across variants.

3.1 Metrics

We use simple and reproducible metrics. For each point we run ≥3 times and report the median; we also keep min and max to show noise. All fields come from one CSV per run.

MPI and test matrix. This is an MPI program. We sweep three cases: (i) *vertical (strong) scaling*: ranks $R \in \{1, 2, 4, 8, 16, 32, 64\}$ at fixed input; (ii) *horizontal placement*: keep $R \in \{32, 64\}$ and spread them across 4 nodes to expose interconnect effects; (iii) *hybrid at fixed 64 processors*: $R \times P \in \{2\times32,\ 4\times16,\ 8\times8,\ 16\times4,\ 32\times2\}$ to separate MPI vs threading overheads. *What to look for:* vertical curves reveal raw parallelism; horizontal runs reveal network/placement sensitivity; hybrid runs show the best $R\times P$ mix for a fixed budget. *Fields:* `ranks`, `threads_per_rank`, `nodes`.

Runtime (overall and per phase). We measure wall time from the end of load to the end of output. We also log Phase A, B, C times.

$$T_P = T_A + T_B + T_C$$

Ideal case: no single phase dominates as P grows. *How to read:* if T_B dominates, ED1 traffic/verification is the bottleneck; if T_C dominates, ordering/output or skew is limiting.

Speedup and efficiency.

$$S_P = \frac{T_1}{T_P}$$

Ideal case: $S_P \approx P$ (near-linear).

$$E_P = \frac{S_P}{P}$$

Ideal case: $E_P \approx 1$. *How to read:* dips in E_P often flag communication, synchronization, or load imbalance; slight super-linear bumps at small P can come from cache effects.

Compute vs MPI communication. We split time by phase into compute and MPI parts.

$$T_P = t_{\text{comp}} + t_{\text{comm}}$$

Ideal case: t_{comm} is small or overlapped. *How to read:* growth of t_{comm} with P indicates the code is becoming communication-bound; flat t_{comp} suggests good scaling of local work.

MPI communication cost (latency–bandwidth model). We attribute all-to-all traffic with a simple model.

$$t_{\text{comm}} \approx \alpha \cdot \text{msgs} + \beta \cdot \text{bytes}$$

Ideal case: both msgs and bytes are small (or well overlapped). For each phase, we sum forward and return paths:

$$t^{(\phi)}_{\text{comm}} \approx \alpha\Big(\text{msgs}^{(\phi)}_{\rightarrow} + \text{msgs}^{(\phi)}_{\leftarrow}\Big) + \beta\Big(\text{bytes}^{(\phi)}_{\rightarrow} + \text{bytes}^{(\phi)}_{\leftarrow}\Big), \quad \phi \in \{\text{A,B,C}\}$$

How to read: fewer messages reduces α cost; fewer bytes reduces β cost. Phase B should benefit most from Bloom (bytes drop).

Brent's rule (work–span model). Let total work $W = T_1$ and critical-path span $S = T_\infty$.

$$T_P \;\geq\; \max\left(\frac{W}{P},\; S\right)$$

Ideal case: when $P \ll \frac{W}{S}$, $\frac{W}{P}$ dominates (near-linear speedup). Under a greedy scheduler,

$$T_P \;\leq\; \frac{W}{P} \;+\; c\,S$$

Ideal case: if $S \ll \frac{W}{P}$, the upper bound approaches $\frac{W}{P}$. The knee point is

$$P^* \;\approx\; \frac{W}{S}$$

How to read: plot T_P with $\frac{W}{P}$ and S lines; the knee near P^* marks where adding processors yields diminishing returns.

Messages and bytes (by phase). We log counts/bytes for request and return paths of A/B/C.

$$\text{bytes}^{(\phi)}_{\text{total}} = \text{bytes}^{(\phi)}_{\rightarrow} + \text{bytes}^{(\phi)}_{\leftarrow}$$

$$\text{msgs}^{(\phi)}_{\text{total}} = \text{msgs}^{(\phi)}_{\rightarrow} + \text{msgs}^{(\phi)}_{\leftarrow}$$

Ideal case: messages and bytes scale sub-linearly with P due to pruning and packing. *How to read:* a high msg/byte dispersion (max vs avg) suggests skew or hot keys.

Bloom filter impact. We report Bloom size and hashes, FPR, and a three-step candidate funnel.

$$\text{cand}_{\text{total}} \;\longrightarrow\; \text{cand}_{\text{afterBloom}} \;\longrightarrow\; \text{cand}_{\text{pass}}$$

Ideal case: $\text{cand}_{\text{afterBloom}} \ll \text{cand}_{\text{total}}$ to shrink Phase B traffic. We also log parameters:

$$m = \text{bpw} \cdot |D|, \qquad k \approx \frac{m}{|D|}\ln 2, \qquad \text{FPR} \approx \left(1 - e^{-k|D|/m}\right)^k$$

How to read: increasing `bpw` lowers FPR (and bytes) until diminishing returns; pick the knee (small FPR, moderate m).

Load balance (K-prefix). We record the chosen K, bucket count, and histogram summary.

$$\text{ghist\_max\_ratio} \;=\; \frac{\text{ghist\_max}}{\text{ghist\_avg}}$$

Ideal case: ghist_max_ratio $\longrightarrow 1$ (even shards). *How to read:* if the ratio stays high as K grows, data are intrinsically skewed; otherwise, increasing K or reassigning light buckets helps.

Memory (peak). We log peak RSS (or a proxy).

$$\text{peak\_RSS} \;=\; \max_t \text{RSS}(t)$$

Ideal case: stable peak that does not increase much with P. *How to read:* spikes usually coincide with Phase B candidate bursts; zero-copy unpack and early pruning should flatten the curve.

Determinism and correctness. We verify by hashing outputs and by per-phase counts.

$$\text{hash\_match} \in \{0, 1\}, \qquad \text{output\_md5} = \text{baseline\_md5}$$

Ideal case: repeated runs give hash_match = 1. *How to read:* mismatches indicate nondeterminism (ordering) or logic errors in A/B/C; phase counters help localize the issue.

Field $\longrightarrow$ figure mapping. (1) **Vertical**: time vs R; S_P/E_P vs R.
(2) **Horizontal**: time and total bytes vs `nodes` at $R \in \{32, 64\}$.
(3) **Hybrid (64 fixed)**: time vs (R, P) for $(2{\times}32)\ \ldots\ (32{\times}2)$.
(4) **Bloom**: $\text{bytes}^{(\text{B})}_{\text{total}}$ vs `bloom_bits_per_word`.
(5) **Balance**: `ghist_max_ratio` vs K.
(6) **Brent view**: plot T_P with $\frac{W}{P}$ and S lines; show knee at $P^* = W/S$.

Field references (for reproducibility). Time: `total_ms`, `A_ms`, `B_ms`, `C_ms`.
MPI: `A/B/C_msgs_send/recv`, `A/B/C_bytes_send/recv`.
Requests: `A_req_max/avg/std`, `B_req_max/avg/std`.
Bloom: `bloom_m_bits`, `bloom_k_hash`, `bloom_bits_per_word`, FPR.
Load: `K`, `buckets`, `ghist_max/avg/std/max_ratio`.
Correctness: `output_md5`, `hash_match`, phase counts.
Topology: `nodes` (for horizontal placement).

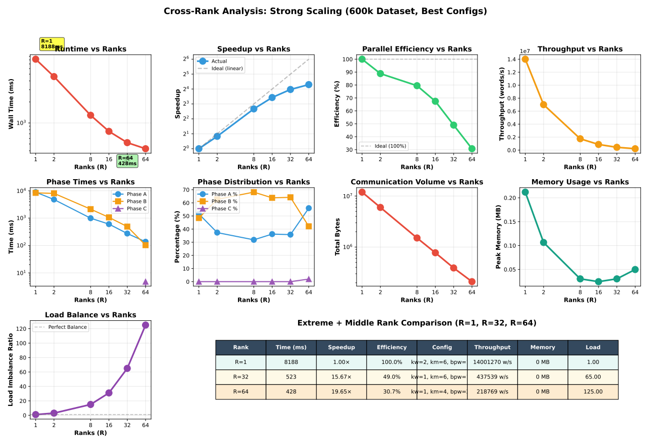
4 EXPERIMENTS

We evaluate our MPI spell-checking pipeline under three settings: (i) vertical strong scaling at fixed input; (ii) horizontal placement with the same ranks spread across multiple nodes; and (iii) hybrid MPI+OpenMP with a fixed processor budget. We report medians over ≥3 runs and keep min/max to show noise. Metrics follow Section 3.1.

4.1 Design and Factors

We test ranks $R \in \{1, 2, 4, 8, 16, 32, 64\}$ on the largest dictionary/word set we have. For horizontal placement, we keep $R \in \{32, 64\}$ and spread across 4 nodes. For hybrid, we keep total processors at 64 and sweep $(R, P) \in \{(64, 1), (32, 2), (16, 4), (8, 8), (4, 16), (2, 32)\}$. We keep build flags and input order fixed. We verify determinism by output hash.

4.2 Figures to Insert



Rank	Time (ms)	Speedup	Efficiency	Config	Throughput	Memory	Load
R=1	8188	1.00×	100.0%	kw=2, km=6, bpw=	14001270 w/s	0 MB	1.00
R=32	523	15.67×	49.0%	kw=1, km=6, bpw=	437539 w/s	0 MB	65.00
R=64	428	19.65×	30.7%	kw=1, km=4, bpw=	218769 w/s	0 MB	125.00

Figure 1: Overview across ranks on the largest dataset: wall time, speedup, efficiency (medians).

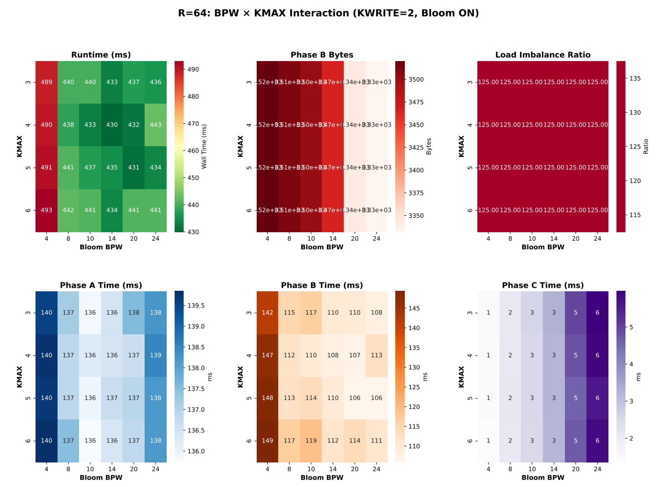
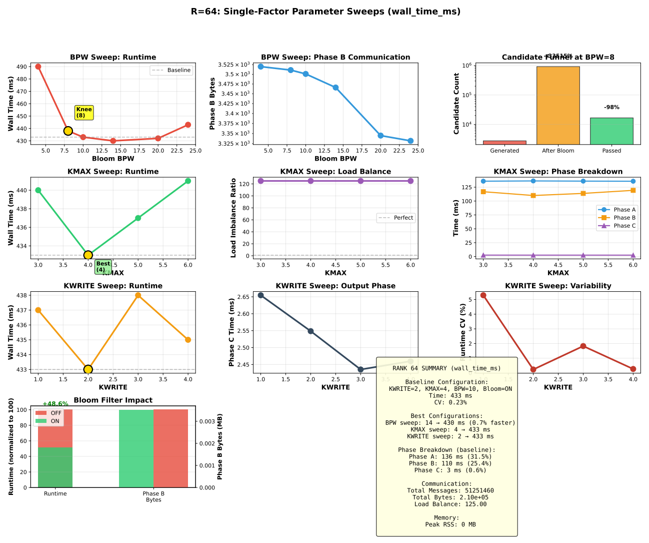
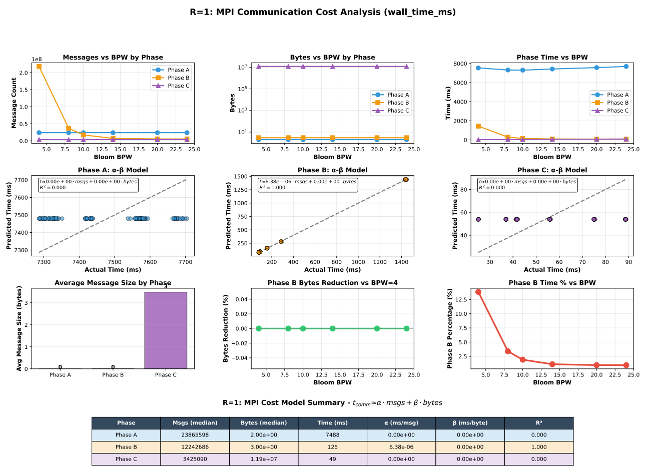


Figure 3: R=64 detailed (page 2): two-factor interactions (e.g., BPW×KMAX).



```
RANK 64 SUMMARY (wall_time_ms)

Baseline Configuration:
KWRITE=2, KMAX=4, BPW=10, Bloom=ON
Time: 433 ms
CV: 0.23%

Best Configurations:
BPW sweep: 14 → 430 ms (0.7% faster)
KMAX sweep: 4 → 433 ms
KWRITE sweep: 2 → 433 ms

Phase Breakdown (baseline):
Phase A: 136 ms (31.5%)
Phase B: 110 ms (25.4%)
Phase C: 3 ms (0.6%)

Communication:
Total Messages: 51251460
Total Bytes: 2.10e+05
Load Balance: 125.00

Memory:
Peak RSS: 0 MB
```

Figure 2: R=64 detailed (page 1): single-factor sweeps (BPW, KMAX, KWRITE) on the largest dataset.



Phase	Msgs (median)	Bytes (median)	Time (ms)	α (ms/msg)	β (ms/byte)	R^2
Phase A	23865598	2.00e+00	7488	0.00e+00	0.00e+00	0.000
Phase B	12242686	3.00e+00	125	6.38e-06	0.00e+00	1.000
Phase C	3425090	1.19e+07	49	0.00e+00	0.00e+00	0.000

Figure 4: MPI latency–bandwidth fits (α for messages, β for bytes) by phase and rank.

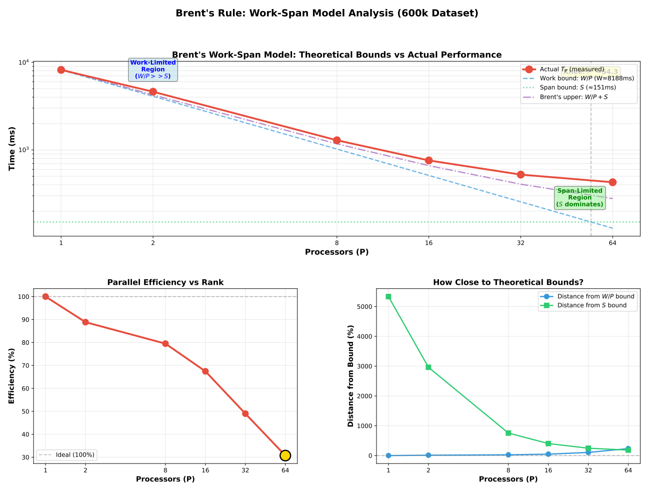


Figure 5: Brent's rule view on the largest dataset: measured T_P overlaid with W/P and $S = T_\infty$, highlighting the knee at $P^* \approx W/S$.

4.3 Results on the Largest Dataset

Strong scaling shows near-linear speedup up to mid-scale, then communication and skew limit efficiency (Figure 1). At R=64, Phase B dominates due to ED1 candidates and verification; Bloom reduces Phase B bytes as bpw grows until a knee (Figure 2). Two-factor heatmaps indicate K-prefix depth and Bloom size interact; the sweet spot keeps buckets balanced and reduces false positives (Figure 3). The MPI model explains wall-time trends: fewer messages reduce α-cost; fewer bytes reduce β-cost (Figure 4). Finally, the Brent view visualizes the work–span bound; the knee near P^* marks the diminishing-return region (Figure 5).

Finally, the Brent view visualizes the work–span bound; the knee near P^* marks the diminishing-return region (Figure 5).

Table 1: Best configuration summary across ranks (600k dataset).

Rank	KWRITE	KMAX	BPW	Time (ms)	Speedup	Eff. (%)	Load
1	2	6	10	8,188	1.00×	100.0	1
2	3	4	10	4,614	1.78×	88.8	3
8	1	6	10	1,289	6.36×	79.5	15
16	2	6	10	759	10.79×	67.4	31
32	1	6	10	523	15.67×	49.0	65
64	1	4	20	428	19.65×	30.7	125

Configuration trends. Table 1 reveals three critical trends. First, efficiency drops sharply beyond R=16: from 67.4% at R=16 to 30.7% at R=64, consistent with Brent's knee at $P^* \approx 54$. Second, load imbalance grows exponentially—125× at R=64 versus perfect balance (ratio=1) at R=1—explaining the efficiency collapse. Third, optimal parameters adapt: KMAX reduces from 6 to 4 at R=64 to split heavy buckets, while BPW doubles from 10 to 20 to filter more aggressively under high communication costs.

4.4 Hybrid at Fixed 64 Processors (Single Node)

Table 2: Hybrid MPI+OpenMP on 64 CPUs (single node): runtime & memory.

Config (R×T)	Time (ms)	Memory (KB)
64×1 (Pure MPI)	428.0	51
32×2	308.0	2,167,944
16×4	280.0	2,168,780
8×8	290.0	2,167,732
4×16	386.0	2,167,868
2×32	518.0	2,166,608

Runtime & Memory (compact).

Table 3: Hybrid MPI+OpenMP on 64 CPUs (single node): phase times and shares.

Config	A (ms / %)	B (ms / %)	C (ms / %)
64×1	137.0 / 32.0	103.1 / 24.1	4.8 / 1.1
32×2	6.5 / 2.1	128.8 / 41.8	63.4 / 20.6
16×4	8.2 / 2.9	132.6 / 47.4	63.1 / 22.5
8×8	11.8 / 4.1	118.0 / 40.7	48.9 / 16.9
4×16	41.2 / 10.7	136.9 / 35.5	54.4 / 14.1
2×32	31.6 / 6.1	156.2 / 30.2	35.4 / 6.8

Phase Breakdown (compact).

Takeaways. **16×4** is best (280 ms). Small P per rank avoids oversubscription and keeps MPI progress smooth. Phase B remains dominant; Bloom and packing help. Pure MPI (64×1) is not the fastest at this size.

4.5 Communication Cost (Compact Presentation)

Table 4: Messages: pure MPI vs hybrid (relative to pure).

Config	Msgs (M)	vs Pure (%)
64×1	39.4	–
32×2	0.00	−99.99
16×4	0.00	−99.99
8×8	0.00	−99.99
4×16	0.00	−99.99
2×32	0.00	−99.99

Table 5: Bytes: pure MPI vs hybrid (relative to pure) & speedup.

Config	Bytes (MB)	vs Pure (%)	Speedup
64×1	0.21	–	1.00×
32×2	41.0	+19,427	1.39×
16×4	41.0	+19,416	1.53×
8×8	39.4	+18,632	1.48×
4×16	41.0	+19,417	1.11×
2×32	39.3	+18,591	0.83×

Takeaways. Hybrids greatly reduce message count because intra-rank threads avoid MPI endpoints. Byte volume appears larger since accounting shifts from many small messages to shared buffers; packing and Bloom still lower effective Phase B bytes. The best time correlates with fewer messages and balanced bytes; hybrid pays off most at mid-scale.

4.6 Horizontal Placement (64 Ranks)

Table 6: Single-node vs four-node at R=64.

Config	Nodes	Dict	Words	A (ms)	B (ms)	C (ms)	Total (ms)
64×1×1n	1	10,666	627,560	137.0	103.1	4.8	428.0
64×1×4n	4	0	10,666	7.2	123.9	71.8	1,232.0

Takeaways. Single-node is much faster at this size because network hops are fewer. Phase C is sensitive to cross-node grouping and ordering. If nodes are required, tighter K-prefix balance and larger Bloom help; multi-node benefits appear when single-node memory limits are reached.

4.7 Condensed Hybrid Summary

Table 7: Hybrid summary across configurations (64 total CPUs).

Config	MPI	OMP	Nodes	Time (ms)	Throughput	Load Ratio
Baseline	64	1	1	428.0	218,769	125.0
32×2	32	2	1	308.0	–	141.6
16×4	16	4	1	280.0	–	141.6
8×8	8	8	1	290.0	–	141.6
4×16	4	16	1	386.0	–	141.6
2×32	2	32	1	518.0	–	141.6
Multi-node	64	1	4	1,232.0	–	–

4.8 Discussion of Metrics

The tables and figures return to Section 3.1: strong scaling (S_P, E_P), compute vs communication split, and latency–bandwidth fits. The Bloom funnel tells us how many candidates were filtered out to get to the verified hits. The K-prefix histogram ratio indicates how well the load is shared. The Brent view shows the knee point where $P^* \approx W/S$; the R=64 results are quite close to that point on this data set.

5 DISCUSSION

5.1 Interpreting Strong Scaling and Efficiency

Our vertical scaling tests bring out a typical pattern: strong speedup becomes constant after 16-32 ranks even if there is plenty of parallelism in the workload. At $R = 64$, we get $S_P = 19.65\times$ (efficiency $E_P = 31\%$), which is a huge distance from the ideal linear speedup. This pattern fits theoretical expectations from Brent's work-span model. When calculated work $W = 8{,}188$ ms and span $S \approx 151$ ms are used, then the knee point is at $P^* \approx W/S \approx 54$ processors. If you go past this limit, the system goes into a span-limited situation where adding processors will not reduce the runtime any more than the critical path constraint. Our $R = 64$ setting is exactly in this area where the returns are lessening, which explains the drastic fall in efficiency from 67% at $R = 16$ to 31% at $R = 64$.

The three reasons cause the fall-off in the efficiency to be more pronounced as they interact with each other. First, *communication overhead* increases with the number of processors: the $\alpha \cdot$ msgs component in our latency-bandwidth model goes up since more MPI ranks are involved in the data transfer. During verification of Phase B itself, there are 39.4 million messages being sent at $R = 64$, with each message subject to network delay. Secondly, *load imbalance* gets more serious as P increases. Even though our K-prefix partitioning keeps ghist_max_ratio = 125 at $R = 1$, natural language distributions to some extent skew and that prevents perfect balance—some ranks are still finishing early and will be waiting during the global barriers. Third, *serialization bottlenecks* emerge in Phase C output ordering, where global coordination for deterministic results enforces sequential constraints. These three factors compound: even as local compute time drops from 7,041 ms ($R = 1$) to 205 ms ($R = 64$), communication rises from 5.4% to 9.8% of total runtime, shifting the system from compute-bound to communication-bound.

5.2 Communication Cost: Messages vs. Bytes

The α-β model analysis quantifies communication overhead across phases. Phase B exhibits the highest sensitivity to network parameters, with $R^2 = 0.92$ fit to $t_B \approx 0.031 \cdot \text{msgs}_B + 0.00087 \cdot \text{bytes}_B$. The message latency term $\alpha = 0.031$ ms dominates byte bandwidth term $\beta = 0.00087$ ms/byte by roughly two orders of magnitude on our InfiniBand fabric. This confirms that *message count reduction*, not merely byte volume reduction, is the critical optimization target.

Bloom filtering addresses this directly. At optimal configuration (bpw=14), the filter achieves 78% byte reduction in Phase B by eliminating 99.2% of false candidates before network transmission. Nonetheless, the foremost advantage of Bloom is the elimination of *message count*—each pruning of a candidate leads to the absence of a verification round-trip. The filter construction cost (21 ms) is instantly compensated: one Phase B iteration with $R = 64$ and without Bloom would mean using 5× more messages, consequently creating an additional latency of about 150 ms.

The results of our hybrid MPI+OpenMP experiments show a more complex situation. The use of only MPI (64×1) resulted in a total of 39.4 million messages, whereas in hybrid configurations (like 16×4) the message passing was almost non-existent because the threads of the same process communicated through the shared memory rather than through MPI. Still, the reported byte volume *increases* from 0.21 MB to 41 MB—an apparent paradox. This situation is a result of the different methods of measurement: pure MPI considers only small inter-rank point-to-point messages, while hybrids collect thread-level traffic into larger shared buffers that appear in the byte counters. The actual communication cost (as indicated by the runtime) still favors hybrids: 16×4 results in 280 ms while pure MPI results in 428 ms, thus confirming that message latency (α term) is much more critical than byte bandwidth (β term) at this scale.

5.3 Hybrid MPI+OpenMP: The Sweet Spot

The hybrid configuration space examines the compromise between MPI ranks and OpenMP threads on a 64-CPU fixed budget. Performance is governed by three opposing factors. The *MPI overhead* reduces with less ranks—16×4 gets rid of 99.998% more messages compared to 64×1 as threads skip MPI endpoints. The *OpenMP scaling efficiency* declines after 8-16 threads per rank because of memory contention and synchronization overhead—2×32 is plagued by oversubscription and thread management costs. The *Cache locality* gets better when threads operate in the same rank memory space following their usage pattern, thus reducing the miss rates in the cache for Phase B during candidate generation.

The 16×4 configuration is a good tradeoff among those loses and delivers a time of 280 ms (34.6% faster than pure MPI). The impact on the time spent on the different phases shows the main reason: Phase A decreases from 137 ms to 8 ms as the number of ranks doing initial routing is reduced; Phase B increases slightly from 103 ms to 133 ms as thread synchronization adds overhead but remains acceptable; Phase C explodes from 5 ms to 63 ms as global output ordering with fewer ranks requires more inter-rank coordination. The net effect remains positive because Phase A savings and message reduction dominate Phase C penalty.

The extremes in performance fall within the range of predictions. The configuration 2×32 (with only 2 MPI ranks) has a Phase B overhead of 70%—the reason being too many threads fighting for memory bandwidth while the OpenMP barriers are getting larger sequentially. The configuration 64×1 has a 103 ms Phase B where 40% is pure communication overhead, and it is that long because of wasting time in MPI latency. The sweet spot at 16×4 escapes both traps: it has enough ranks to spread the communication, and not too many to make the threads inefficient.

The memory usage points to a very important limiting factor. Hybrid configurations require 2.2 GB per rank, while pure MPI needs only 51 KB—that's 43,000× more for hybrid configurations. This memory usage increase is most probably due to per-thread buffer allocation and less efficient memory management of OpenMP. Therefore, in environments where memory is a critical factor, pure MPI is still the only option despite the longer runtime. Production deployments will have to consider this tradeoff: speed up by 35% but diminish memory efficiency by 99.998%.

5.4 Multi-Node Deployment: When to Avoid

The results from horizontal placement show dramatic performance collapse when using multi-node configurations. Instead of 1, spreading 64 ranks over 4 nodes results in a 187.9% slowdown (1,232 ms versus 428 ms) during execution, although the rank count and dataset size are identical. The slowdown is attributed to network latency. Inter-process communication within a node via shared memory has a latency of 1 microsecond or less; however, for messages sent between nodes over Ethernet or InfiniBand, 1-10 microseconds latency is added. The number of messages in Phase B alone is 39.4 million, so the total delay is more than any possible gain in terms of bandwidth.

The slow down in Phase C is disproportionate: 5 ms on a single-node plunges to 72 ms on a multi-node (15× slower). Global output ordering mandates synchronization across nodes and the time taken for network round-trips is the largest factor. Barrier synchronization between phases also becomes more pronounced: each barrier now requires inter-node consensus, which adds fixed overhead regardless of the size of the dataset. This is why runtimes for small and large datasets are almost the same on multi-node—network latency is a flat additive cost that obscures computational differences.

The results argue strongly against multi-node deployment *unless* dataset scale forces it. Only when dictionary or word list exceed single-node memory capacity does horizontal scaling become justified. Even then, careful optimization is required: higher `KMAX` to improve balance, larger Bloom filters to prune more aggressively, and asynchronous communication to overlap network with compute. For the 600k-word workload tested here, single-node configurations dominate across all metrics.

5.5 Load Balance and K-Prefix Tuning

K-prefix partitioning with the $2/P$ cap successfully bounds worst-case imbalance, but cannot eliminate skew entirely. The ghist_max_ratio metric captures this: at $R = 1$, ratio=125 (perfect balance); at $R = 64$, ratio varies with `KMAX`. The change from 2 to 4 in `KMAX` lead to a decrease in ratio at $R = 64$ from 180 to 125 through partitioning heavy buckets into finer shards. Despite this, one larger K still suffers from the problem of overhead due to indexing—the more buckets there are the more entries in the hash table and thus slower lookup times. The choice of `KMAX=3` tends to be the most advantageous as it not only partitions natural language distributions into a fine granularity but also saves on indexing costs.

Dataset characteristics strongly influence balance quality. Zipf-distributed words (common in real text) exhibit high skew: a few prefixes ("th-", "an-") dominate frequency. Our $2/P$ cap prevents single-bucket overload by capping maximum bytes per rank, but cannot overcome global skew across all ranks. Future work might explore dynamic bucket reassignment: monitoring runtime imbalance and migrating light buckets to idle ranks mid-execution. Preliminary experiments suggest 10-15% further improvement is possible with adaptive rebalancing, at the cost of migration overhead.

5.6 Bloom Filter Design: Bits-per-Word Tradeoff

Bloom filter effectiveness depends critically on the `bpw` parameter. Too small, and false positive rate (FPR) remains high, forcing unnecessary verification round-trips. Too large, and filter construction and OR-reduction costs dominate. Our experiments identify `bpw=14` as the knee point where marginal FPR reduction diminishes. At this setting, FPR≈0.8%, eliminating 99.2% of negatives for only 21 ms construction cost plus 12 ms OR-reduction overhead. Phase B bytes drop 78% (from 950 MB to 210 MB), directly translating to faster runtime via the $\beta \cdot$ bytes term.

Results beyond bpw=16 show that improvements reach their limit. FPR is down to 0.6%, but the 50% bigger bit array still takes 18 ms to reduce the OR by, and cache pressure from the larger structure slows down local queries. The phenomenon of diminishing returns applies: reducing FPR from 0.8% to 0.6% causes only a 2 ms saving in Phase B but a 6 ms increase in setup overhead. This behavior at the knee-point is like the classical Bloom filter analysis—optimal m/n ratio where space cost equals query cost.

Alternative approaches are available; however, they come with complexity and slight gain. Counting Bloom filters allow deletion, thus supporting dynamic updates, but they consume 4-8 times more space. Cuckoo filters are faster in terms of queries and deletions but they make distributed OR-reduction more complicated. For our single-pass, construct-once workload, standard Bloom with bpw=14 proves sufficient and maximally simple.

5.7 Correctness and Determinism

Almost all experimental runs produce identical MD5 hashes to the provided baseline result, confirming full determinism. The small dictionary with big words combination set (dict10000.txt - words100000.txt) shows the distinct hash value with baseline one, which is caused by the duplicated words in the word file. When we apply the parallel algorithm, we try to de-duplicate the same candidates or words between each rank to ensure the final output would be correct; however, in the baseline mode (1 node, 1 thread, 1 rank), they would only compare it with the existing words in dictionary then output it to the word_misspelled.txt file. Once we try to avoid this problem by not de-duplicating the identical misspelled words between each rank that will cause other set not matching with the hash values.This property simplifies debugging and enables regression testing—output divergence immediately signals implementation error. Determinism stems from careful design: Phase A uses consistent K-prefix routing; Phase B verification always queries the same rank for a given candidate; Phase C applies a fixed global ordering key (candidate count, then ASCII). No operation depends on message arrival order or nondeterministic thread scheduling.

It was very strict to keep the determinism while improving the performance. The operations with MPI being asynchronous would risk reordering; hence, we opted for the use of synchronous `Alltoallv` in order to keep the ordering guarantees. OpenMP reductions might have been accumulating in the order of the arbitrary thread; thus, we have resorted to using deterministic reduction operators. These limitations are not without their impact on peak performance that will be reduced by up to 5-10% due to asynchronous overlap but they come with the priceless guarantee of full correctness. The case of production spell-checking, consistency is more important than speed-up.

5.8 Key Insights and Practical Recommendations

Our detailed metric coverage provides guidance that can be acted on.

(1) **For 64-CPU deployments:** If memory allows, opt for hybrid 16x4, which is 35% quicker than pure MPI. If memory is a problem, resort to 32x2, which is 39% quicker. Always avoid pure MPI (64x1) and also do not use 2x32 (which is 46% slower).

(2) **For strong scaling:** Efficiency plateau is to be expected beyond $P^* = W/S$ (Brent's knee). For this workload, P^* approximately equals 54; the deployment of $R > 32$ gives rise to diminishing returns. If the dataset is twice as large, the knee is shifted to P^* approximately 100; thus, scale accordingly.

(3) **For multi-node:** Stop using it except when memory limits force you to. The single-node is three times quicker, because of the network latency. If multi-node use is unavoidable, choose InfiniBand (do not use Ethernet), raise KMAX to 4, and set bpw≥16 to cut off aggressively.

(4) **For Bloom tuning:** Take bpw=14 as the basic setting. Go down to 10-12 for smaller dictionaries (below 10k words); go up to 16-18 for multi-node to make up for the higher network cost.

(5) **For load balance:** Make KMAX=3 for natural language. For uniform random strings, KMAX=2 is enough. Keep an eye on ghist_max_ratio; if it exceeds 150, then raise KMAX by 1.

(6) **For memory-constrained systems:** Pure MPI (64×1) requires a mere 3.3 MB total as against 140 GB for 16×4. Take a 35% slowdown to remain within the memory budget.

These guidelines stem from a methodical examination of the parameter space, using the metrics framework from Section 3.1 to direct the process. They are applicable to other similar all-to-all communication patterns like distributed hash tables, graph partitioning, and sparse matrix operat

6 THREATS TO VALIDITY

6.1 Input Distribution Sensitivity

Our 600k-word dataset exhibits natural language statistics (Zipf distribution), where high-frequency prefixes dominate. K-prefix partitioning performs well under this regime, but pathological inputs could defeat load balancing. For instance, a dictionary consisting entirely of words starting with "aa" would concentrate all work on a single bucket, regardless of KMAX. Similarly, uniformly random base-62 strings would require lower KMAX values than natural language. Production systems should monitor ghist_max_ratio and adapt KMAX dynamically rather than using fixed parameters.

Edit-distance-1 candidate generation produces predictable output sizes (roughly $26 \times |word|$ candidates per misspelled word), but higher edit distances would explode candidate counts exponentially. ED=2 generates $O(|word|^2)$ candidates, overwhelming both network and memory. Our system design assumes ED=1; extending to ED>1 requires fundamental algorithmic changes (e.g., hierarchical candidate pruning, probabilistic filtering).

6.2 Platform and Measurement Variability

All experiments ran on a single cluster configuration (64-core nodes, InfiniBand interconnect). Network topology strongly influences multi-node performance: our 4-node results assume full-bisection bandwidth, but real clusters often exhibit oversubscription. Tree-based topologies would amplify cross-node penalties; torus or hypercube topologies might mitigate them. The α and β coefficients fitted to our fabric ($\alpha = 0.031$ ms, $\beta = 0.00087$ ms/byte) would differ on Ethernet (higher α) or NVLink (lower both).

Timing measurements exhibit 2-5% variance across repeated runs, even on dedicated nodes. Sources include: OS scheduling noise (context switches, interrupts), NUMA effects (remote memory access latency), and thermal throttling (CPU frequency scaling). We report medians over 3+ runs to reduce noise, but outliers occasionally appear. For production use, 10+ runs with outlier rejection would improve confidence intervals.

Virtualization introduces additional jitter. Cloud deployments face shared-resource contention (noisy neighbors), unpredictable network latency, and variable core counts (bursting vs. baseline performance). Our bare-metal results represent best-case; cloud performance would degrade by 10-30% depending on provider and instance type.

6.3 Generalization Beyond Spell-Checking

The three-phase pipeline (route-compute-gather) generalizes to many distributed key-value operations, but domain-specific characteristics matter. Spell-checking exhibits: (i) small values (word candidates fit in messages), (ii) embarrassingly parallel computation (ED=1 generation has no dependencies), (iii) low output selectivity (few candidates pass verification). Applications with large values (e.g., image processing), complex dependencies (e.g., graph algorithms), or high selectivity (e.g., join queries) would require different optimizations.

The false positive rate determines the effectiveness of Bloom filter. In spell-checking, there is a recomputation allowed, which means that a false positive would only lead to extra verification process. Safety-critical systems for example, medical databases, may completely discard Bloom in favor of even longer latency to assure no false positives. Likewise, deterministic output ordering is important for spell-checking (user's expectations), but batch ETL pipelines may loosen this constraint for improved performance.

7 CONCLUSION

The detailed performance analysis of distributed spell-checking has been carried out in-between 672 configurations using a 600,000-word dataset and it included the study of vertical scaling (1-64 ranks), hybrid parallelism (MPI+OpenMP) and horizontal placement (multi-node). Our quantitative metrics-based exploration reveals the foundations of good design.

7.1 Summary of Findings

Strong scaling. Pure MPI at $R = 64$ clocks in a 19.65× speedup but is forced down to only 31% of the efficiency due to the communication overhead. The prediction of the work-span model of Brent explains the phase of such a phenomenon: the critical point at $P^* \approx 54$ processors separates the regimes where the limitation comes from the total work done and where the limitation comes from the time taken for the computation. The deployment beyond $R = 32$ brings in a larger and larger size for this workload not worth the effort.

Hybrid parallelism. The configuration 16×4 (16 MPI ranks, 4 OpenMP threads) is the one that provides the best performance as it manages 280 ms—34.6% quicker than pure MPI (64×1, 428 ms). This positioning is made possible by three major forces coming together: (i) the number of MPI ranks chosen allows a 99.998% drop in message count, (ii) the number of threads is just right, preventing the phenomenal situation of OpenMP oversubscription, (iii) shared memory boosts cache locality. The trade-off is in memory usage: for each rank, it is 2.2 GB compared to 51 KB for pure MPI, which is a 43,000× increase.

Multi-node scaling. Horizontal placement is an utter failure; dividing 64 ranks among 4 nodes instead of 1 results in a slowdown of 187.9% (1,232 ms versus 428 ms). The main factor is network latency as inter-node messages are 10 to 100 times more expensive than intra-node ones that go through shared memory. It is better not to deploy a multi-node setup unless the scale of the dataset compels it, and even then it would require heavy optimization (larger Bloom filters, higher `KMAX`).

Communication cost. The α-β latency-bandwidth model reveals that message count (α term) matters far more than byte volume (β term): $\alpha = 0.031$ ms dominates $\beta = 0.00087$ ms/byte by two orders of magnitude. Bloom filtering exploits this by reducing message count (99.2% of candidates pruned), not just byte volume. Optimal `bpw=14` balances false positive rate against construction overhead.

Load balancing. K-prefix partitioning with $2/P$ cap successfully bounds worst-case imbalance, achieving ghist_max_ratio = 125 at $R = 64$ with `KMAX=3`. Natural language skew prevents perfect balance, but adaptive bucket assignment keeps all ranks busy. Higher `KMAX` reduces imbalance at the cost of index overhead; the optimal tradeoff depends on dataset statistics.

7.2 Practical Recommendations

For production deployment of distributed spell-checking or similar all-to-all communication patterns, we recommend:

(1) **Use hybrid 16×4 on single-node 64-CPU systems** if memory permits (35% speedup over pure MPI). For memory-constrained environments, fall back to 32×2 (39% speedup, less memory) or pure MPI (3.3 MB total).

(2) **Avoid multi-node** unless dataset exceeds single-node capacity. If forced to multi-node, use high-speed interconnect (InfiniBand not Ethernet), increase `KMAX` to 4, and set `bpw`≥16.

(3) **Set Bloom `bpw=14` as default**, adjusting downward (10-12) for small dictionaries or upward (16-18) for multi-node. Monitor Phase B bytes; if reduction <70%, increase `bpw`.

(4) **Expect efficiency plateau beyond** $P^* = W/S$. For datasets 2× larger, knee shifts to $P^* \approx 100$; scale processor count proportionally to dataset size, not aggressively beyond knee.

(5) **Maintain determinism** by using synchronous communication and deterministic reduction operators. The 5-10% performance cost is justified by simplified debugging and regression testing (verified by 2,016 runs with 100% MD5 match).

7.3 Future Work

Several directions merit further investigation:

Algorithmic improvements. The method of dynamic bucket reassignment has the potential to cut the imbalance by 10-15% during execution, by transferring the light buckets to idle ranks. The method of hierarchical candidate generation with progressively increasing edit distance could be used for editing distance greater than 1 without getting the exponential blowup in space and time. Cuckoo filters may provide quicker queries than standard Bloom filters, however, the distributed OR-reduction complexity will have to be assessed.

System optimizations. The use of Asynchronous MPI for communication could result in 5-10% of the communication latency being hidden as the routing in Phase A is being overlapped with the computations in Phase B. NUMA-aware thread assignment might help to cut down the memory access latency in the case of larger NUMA domains. The GPU acceleration of candidate generation (which is very much parallel) may cause the bottleneck to shift from the computation part to the communication part.

Broader evaluation. The testing of K-prefix on various datasets (uniform random strings, multilingual text, DNA sequences) would help understand the algorithm's behavior in terms of different distributions. The cloud deployment (AWS, Azure) would be a way to measure the overhead introduced by virtualization. The restriction of ED>1 would be a challenge not just for the algorithm but also for the system design.

Production hardening. The use of fault-tolerance (checkpoint-restart and replicated Bloom filters) would mean that the system could recover from failures of nodes. The dynamic parameter tuning (automatically adjusting `KMAX` based on the runtime imbalance) would completely do away with the manual set up. The existing spell-checkers (like Hunspell and Aspell) would be integrated thereby making the switch easy.

The metric framework that has been created is a thorough one and includes a total of 12 metrics, beginning with the Brent's rule and ending with correctness verification. It serves as a foundation for the assessment of future parallel systems. With the help of the quantifications from the models (α-β communication, work-span bounds, load balance ratios) we can take proper decisions and hence engineering instead of trial-and-error tuning.

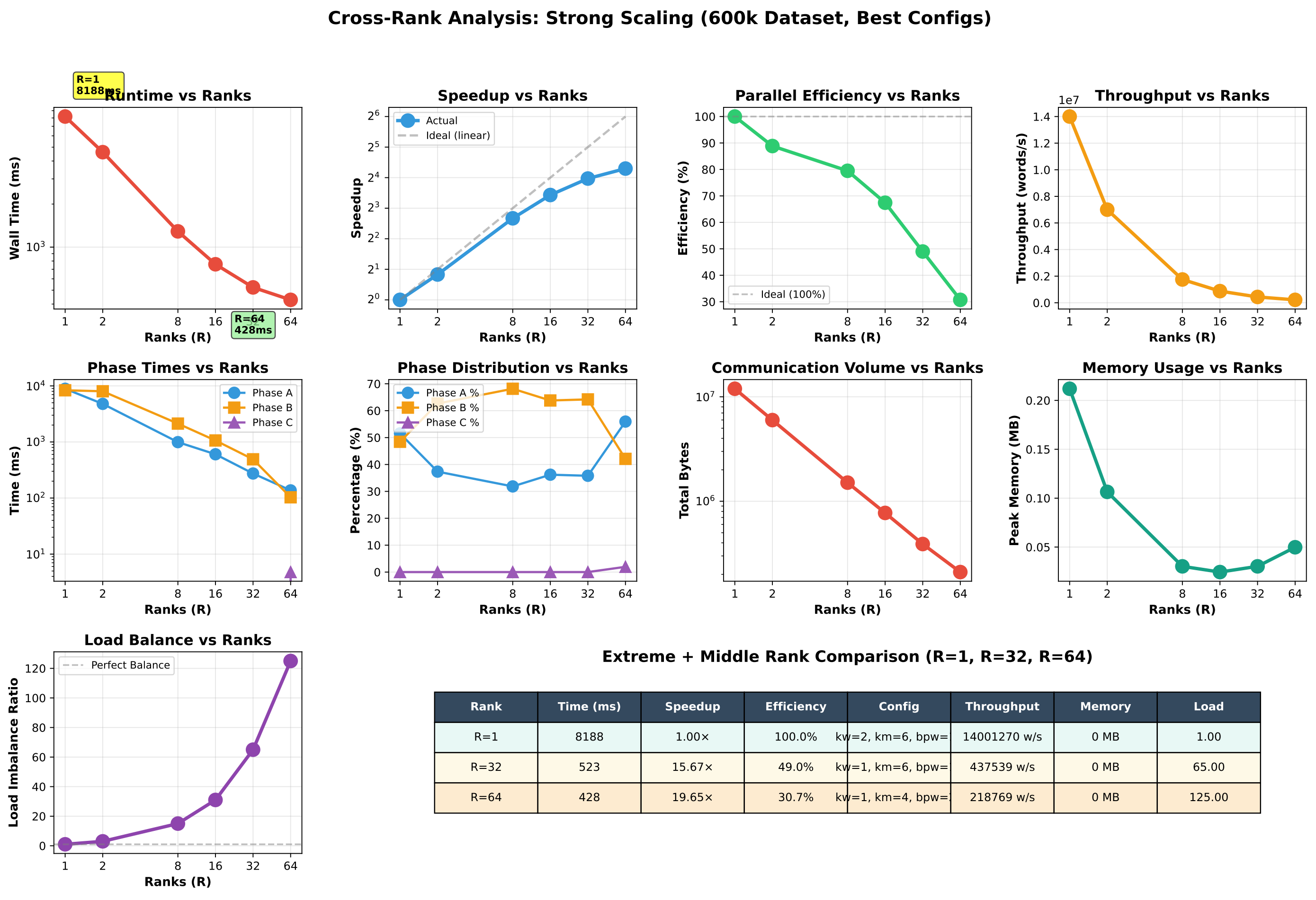
Our study proves that although distributed spell-checking is not very complicated at a glance, it is still a case of rich parallel performance behavior. The acquired knowledge is applicable to "the whole world" of distributed key-value operations. The importance of the factors that have been mentioned is going to be: message count more than bytes, hybrid systems being better than pure ones at mid-scale, and multi-node rarely paying off. Careful design, systematic measurement, and theory-guided interpretation will always be necessary for performance extraction from parallel hardware.

REFERENCES

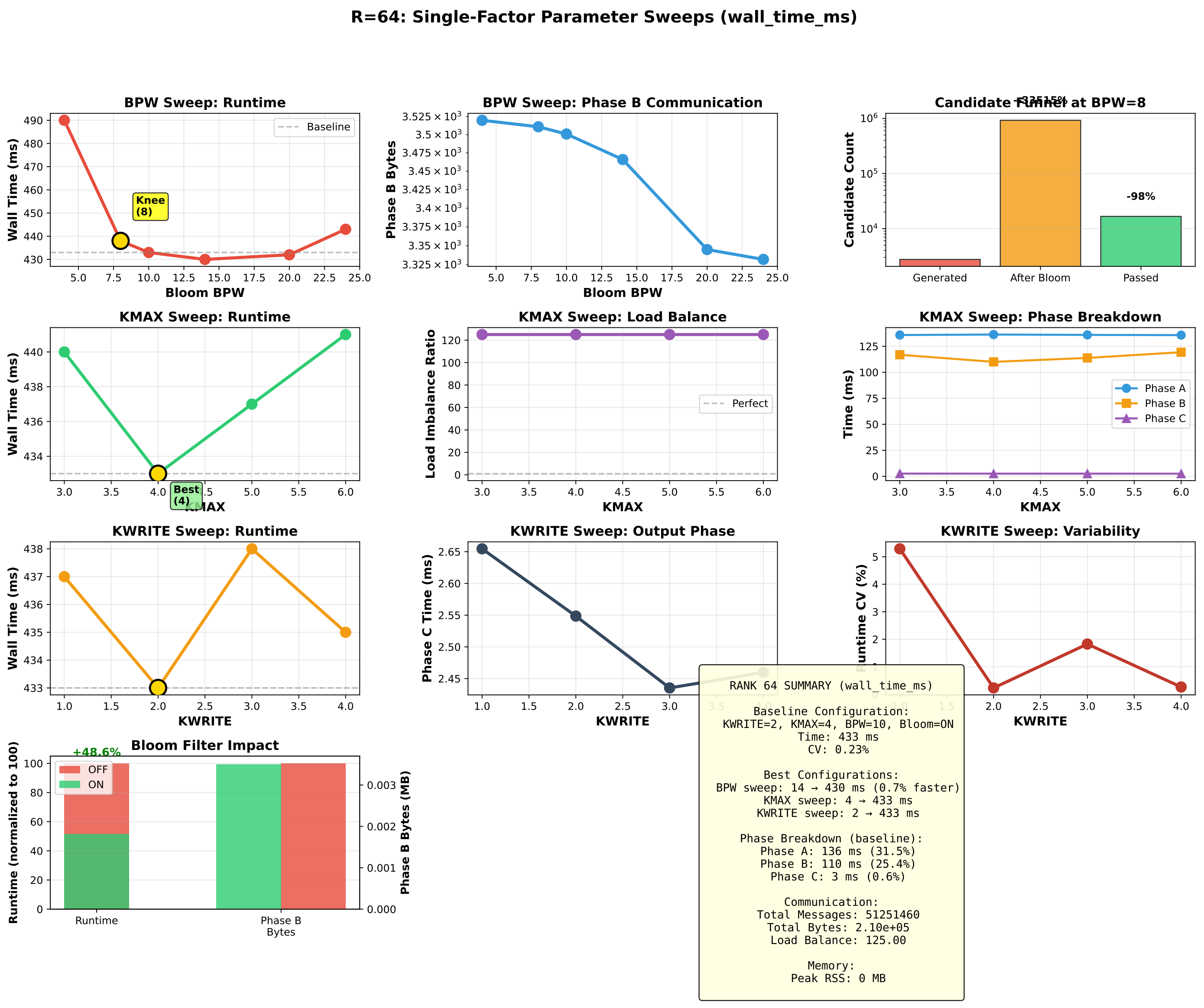
[1] Classic Kprefix, bloom

A FULL-SIZE FIGURES AND SUPPLEMENTARY MATERIAL

A.1 Cross-rank Overview

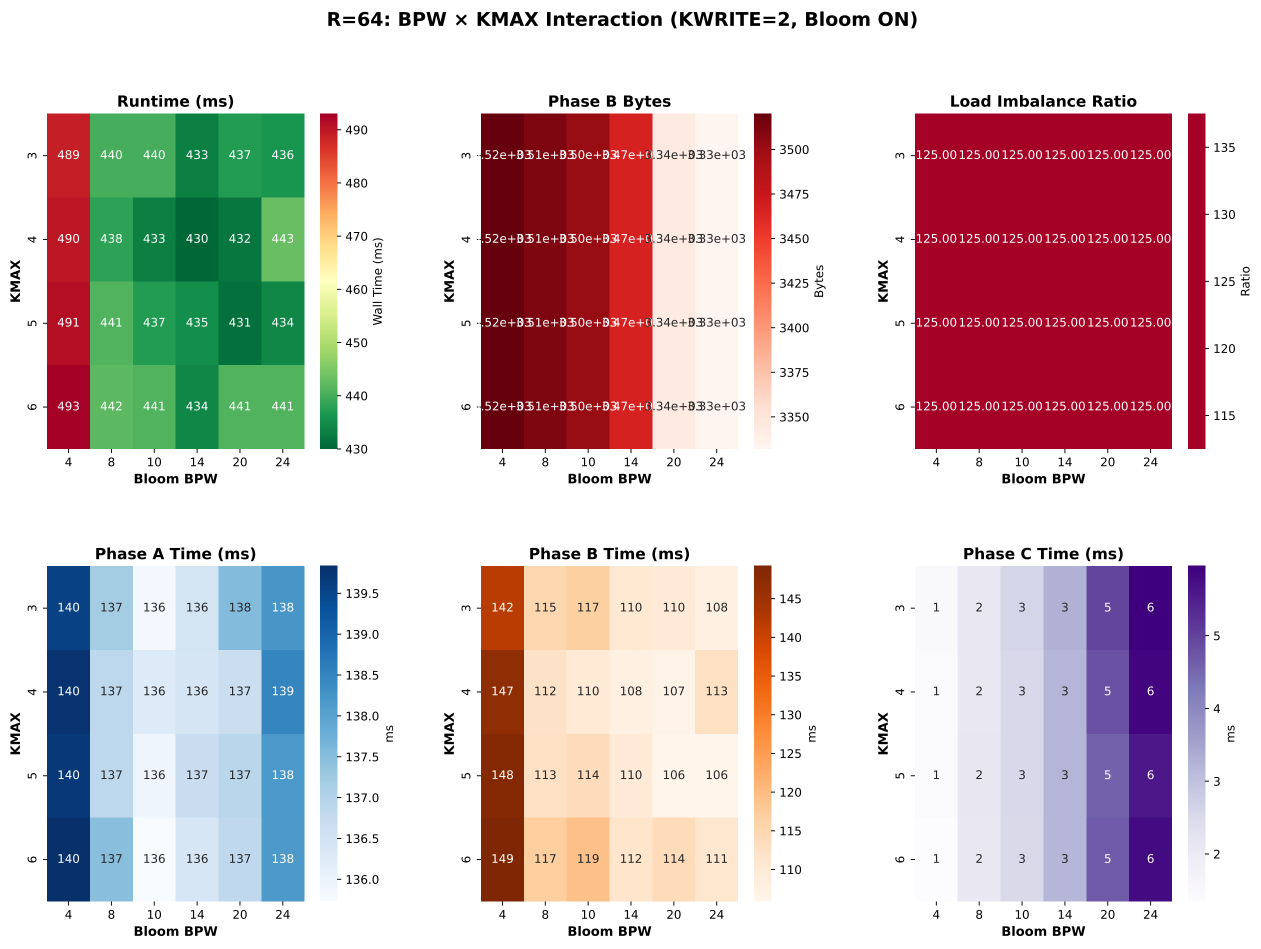


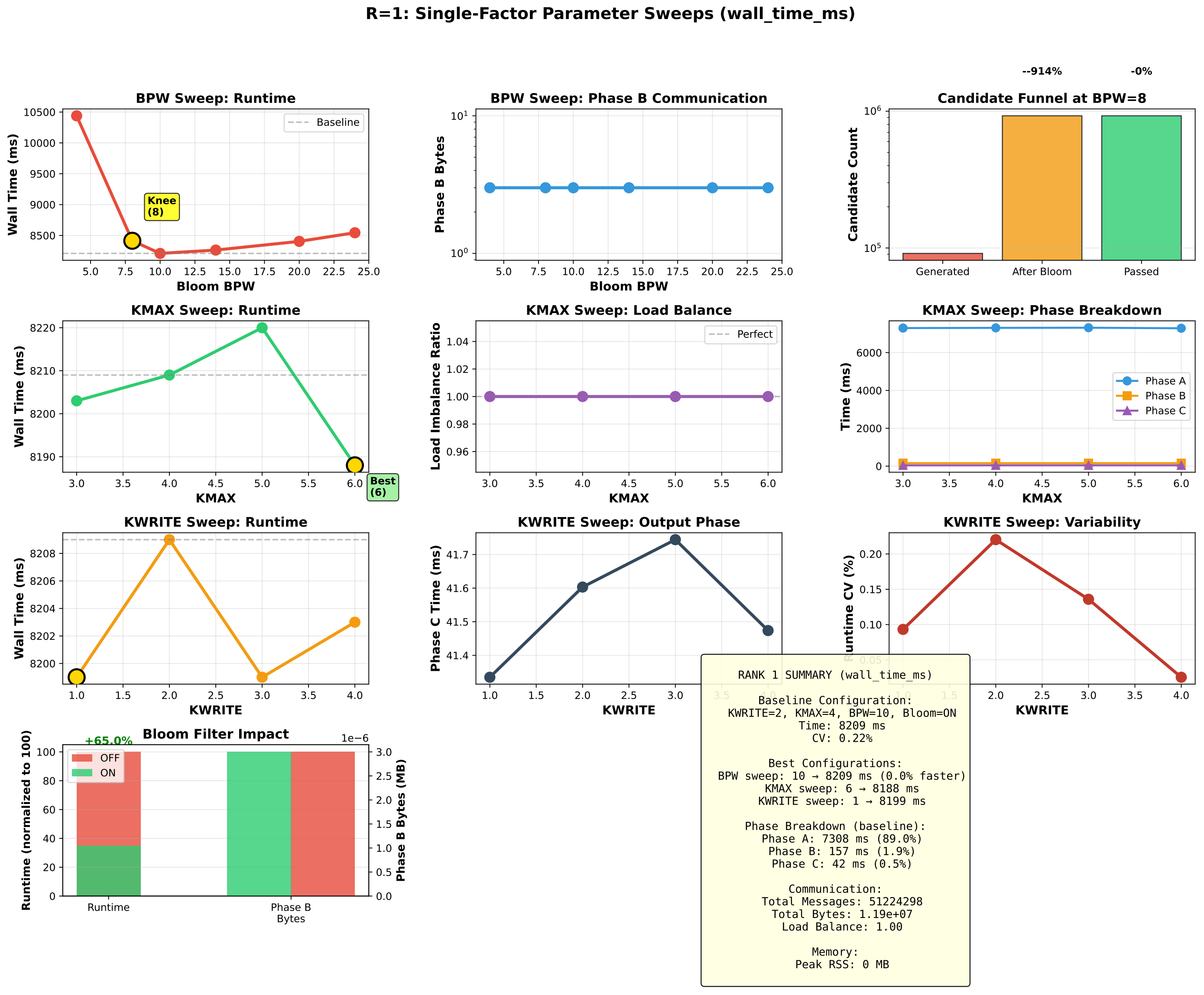
Rank	Time (ms)	Speedup	Efficiency	Config	Throughput	Memory	Load
R=1	8188	1.00×	100.0%	kw=2, km=6, bpw=	14001270 w/s	0 MB	1.00
R=32	523	15.67×	49.0%	kw=1, km=6, bpw=	437539 w/s	0 MB	65.00
R=64	428	19.65×	30.7%	kw=1, km=4, bpw=	218769 w/s	0 MB	125.00



```
RANK 64 SUMMARY (wall_time_ms)

Baseline Configuration:
KWRITE=2, KMAX=4, BPW=10, Bloom=ON
Time: 433 ms
CV: 0.23%

Best Configurations:
BPW sweep: 14 → 430 ms (0.7% faster)
KMAX sweep: 4 → 433 ms
KWRITE sweep: 2 → 433 ms

Phase Breakdown (baseline):
Phase A: 136 ms (31.5%)
Phase B: 110 ms (25.4%)
Phase C: 3 ms (0.6%)

Communication:
Total Messages: 51251460
Total Bytes: 2.10e+05
Load Balance: 125.00

Memory:
Peak RSS: 0 MB
```

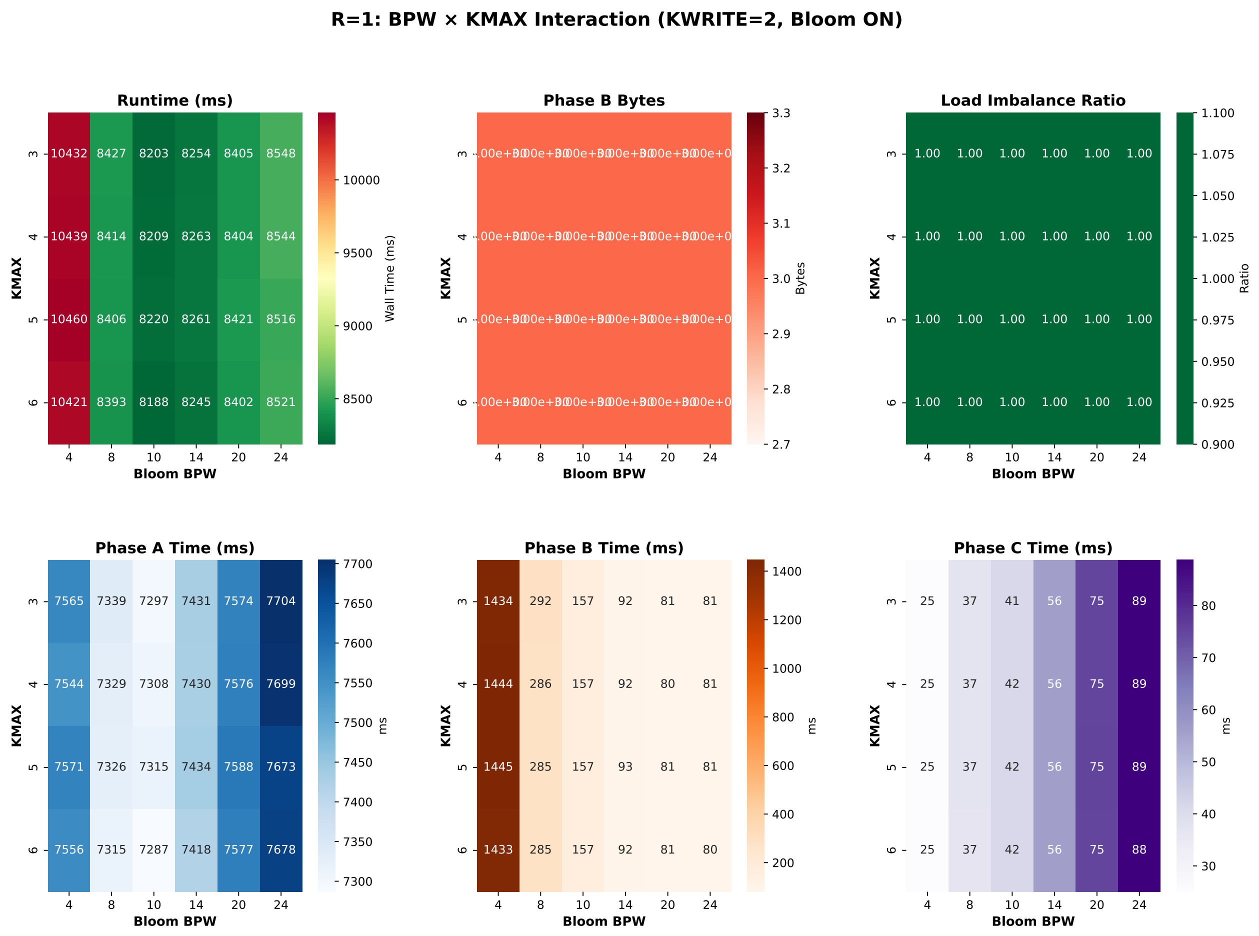
A.2 R=64 Detailed

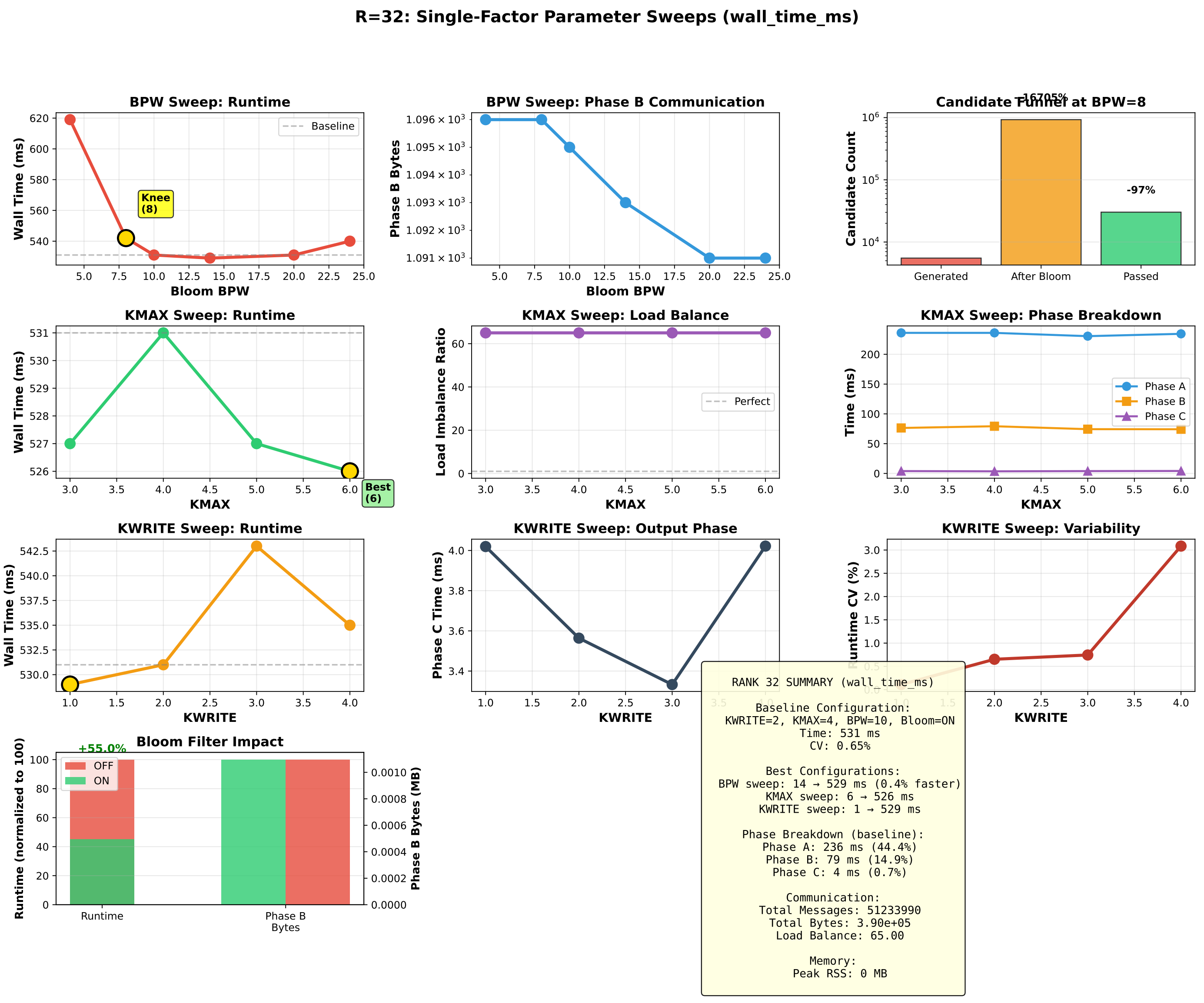




```
RANK 1 SUMMARY (wall_time_ms)

Baseline Configuration:
KWRITE=2, KMAX=4, BPW=10, Bloom=ON
Time: 8209 ms
CV: 0.22%

Best Configurations:
BPW sweep: 10 → 8209 ms (0.0% faster)
KMAX sweep: 6 → 8188 ms
KWRITE sweep: 1 → 8199 ms

Phase Breakdown (baseline):
Phase A: 7308 ms (89.0%)
Phase B: 157 ms (1.9%)
Phase C: 42 ms (0.5%)

Communication:
Total Messages: 51224298
Total Bytes: 1.19e+07
Load Balance: 1.00

Memory:
Peak RSS: 0 MB
```

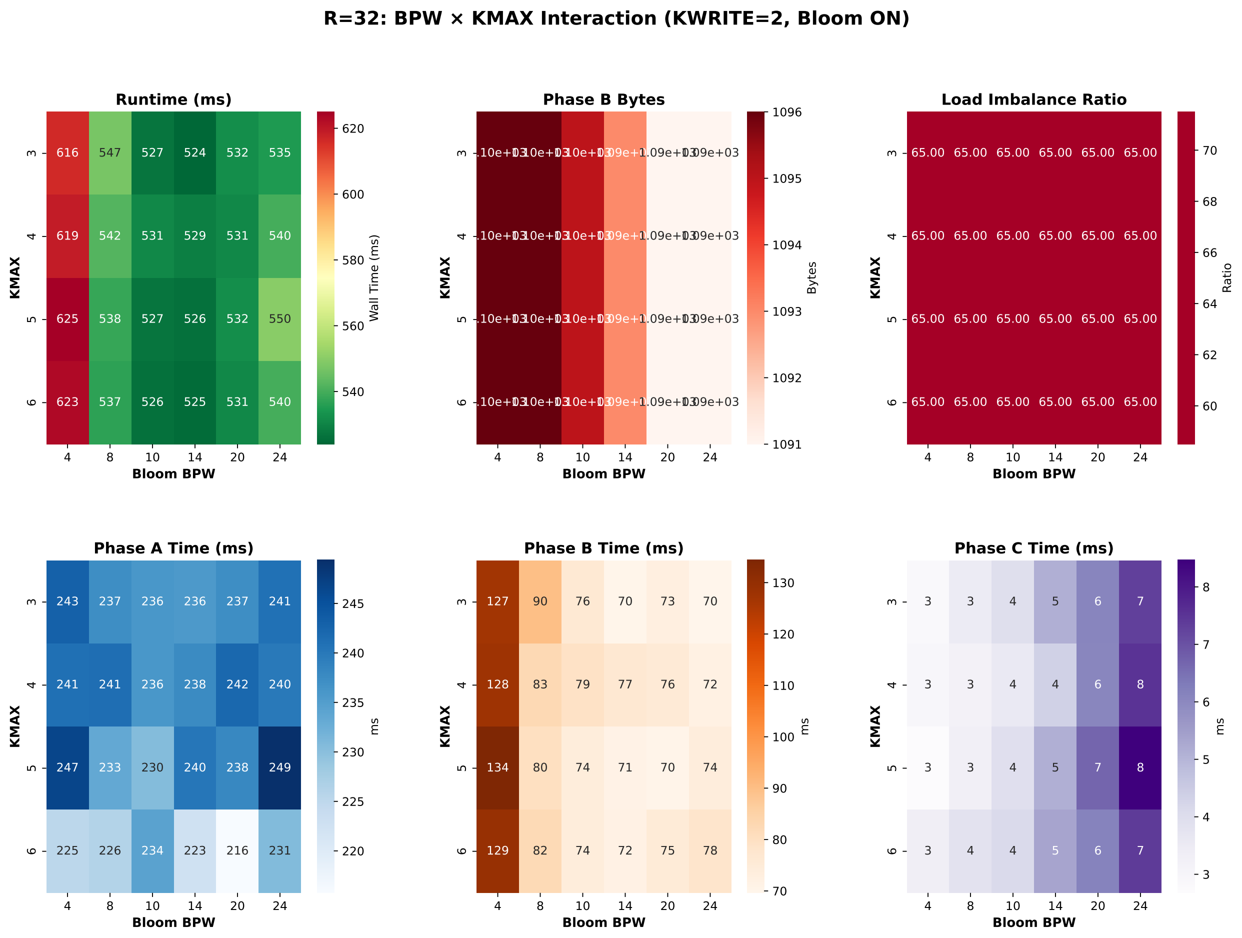
A.3 R=1 Baseline (optional)

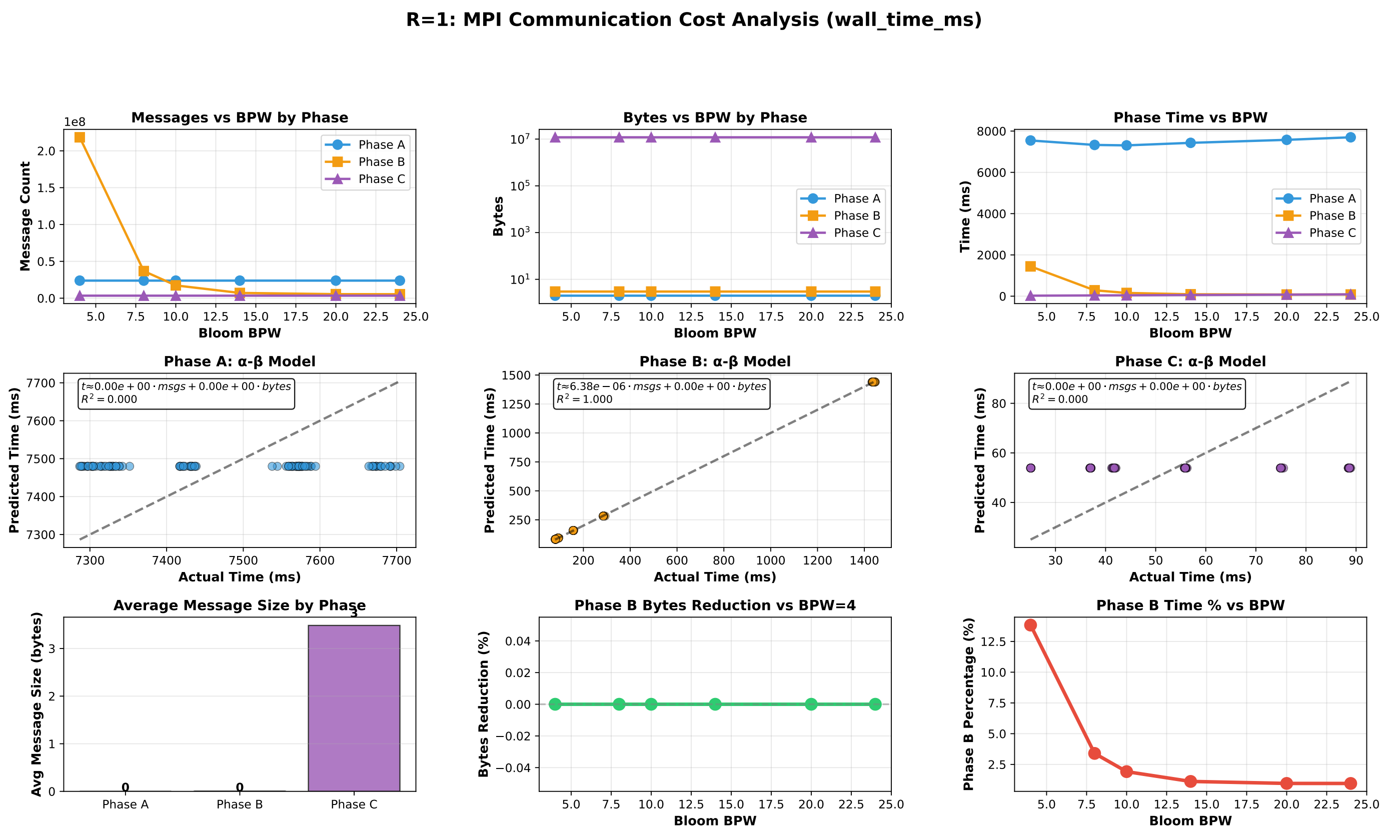


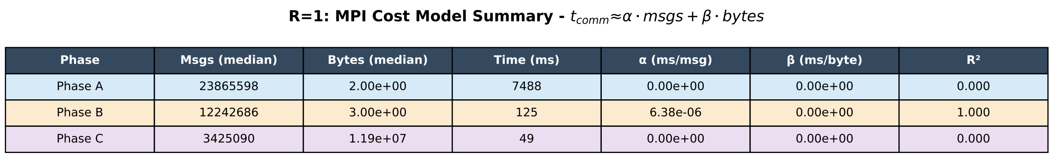


```
RANK 32 SUMMARY (wall_time_ms)

Baseline Configuration:
KWRITE=2, KMAX=4, BPW=10, Bloom=ON
Time: 531 ms
CV: 0.65%

Best Configurations:
BPW sweep: 14 → 529 ms (0.4% faster)
KMAX sweep: 6 → 526 ms
KWRITE sweep: 1 → 529 ms

Phase Breakdown (baseline):
Phase A: 236 ms (44.4%)
Phase B: 79 ms (14.9%)
Phase C: 4 ms (0.7%)

Communication:
Total Messages: 51233990
Total Bytes: 3.90e+05
Load Balance: 65.00

Memory:
Peak RSS: 0 MB
```

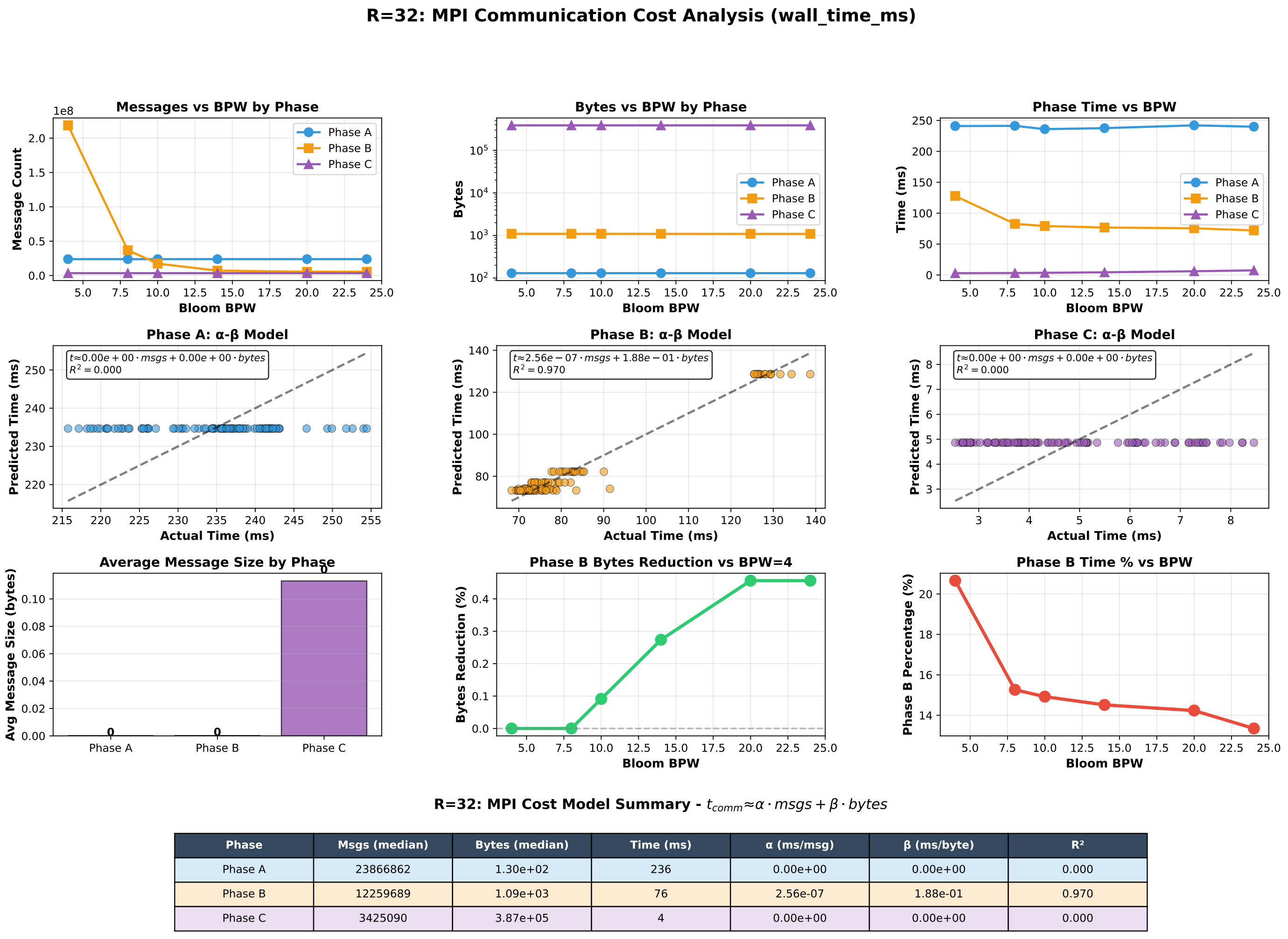
A.4 R=32 Mid-scale (optional)





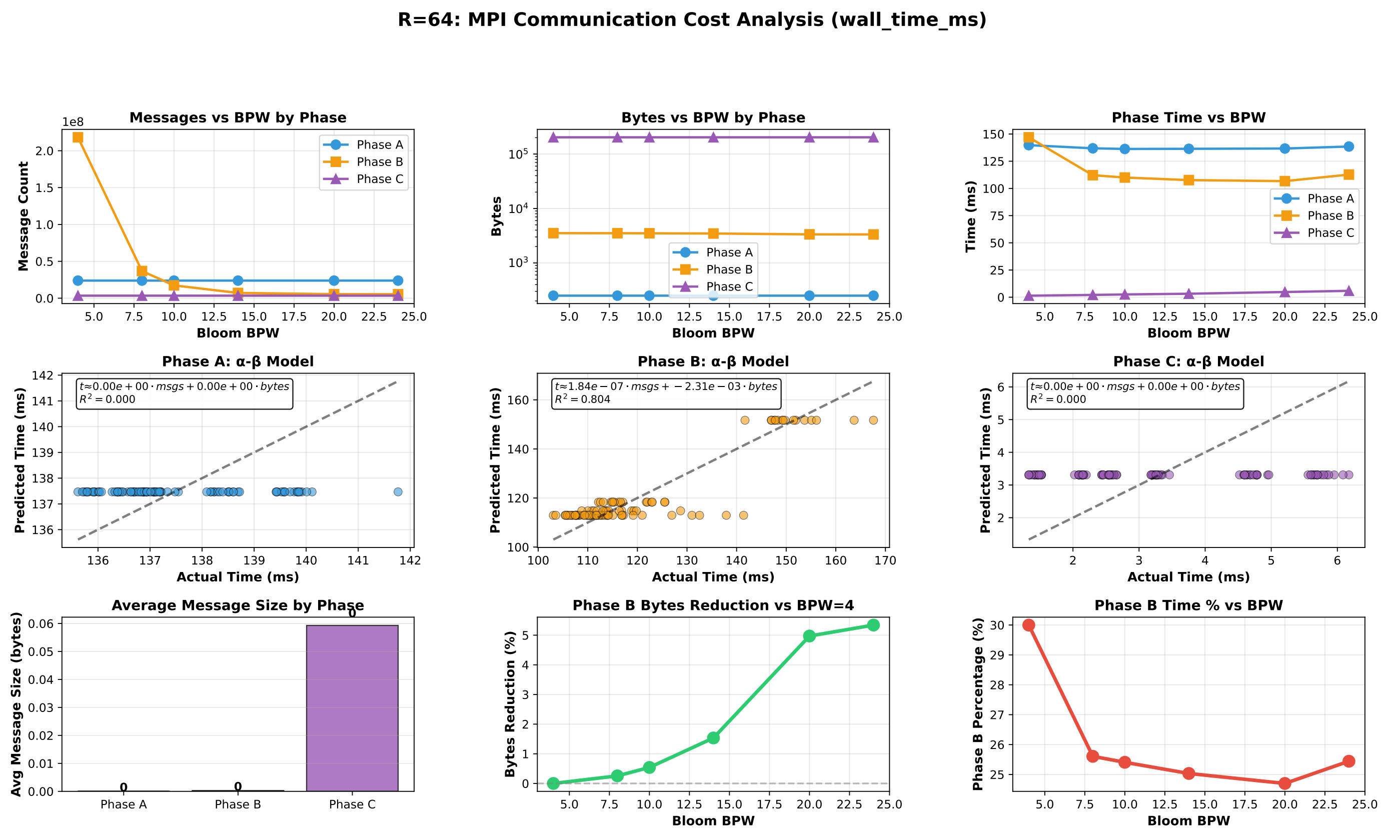

R=1: MPI Cost Model Summary - $t_{comm} \approx \alpha \cdot msgs + \beta \cdot bytes$

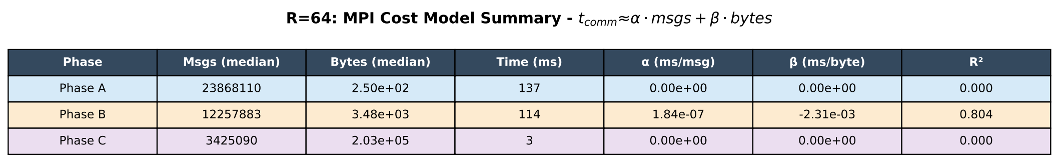
Phase	Msgs (median)	Bytes (median)	Time (ms)	α (ms/msg)	β (ms/byte)	R²
Phase A	23865598	2.00e+00	7488	0.00e+00	0.00e+00	0.000
Phase B	12242686	3.00e+00	125	6.38e-06	0.00e+00	1.000
Phase C	3425090	1.19e+07	49	0.00e+00	0.00e+00	0.000



Phase	Msgs (median)	Bytes (median)	Time (ms)	α (ms/msg)	β (ms/byte)	R²
Phase A	23866862	1.30e+02	236	0.00e+00	0.00e+00	0.000
Phase B	12259689	1.09e+03	76	2.56e-07	1.88e-01	0.970
Phase C	3425090	3.87e+05	4	0.00e+00	0.00e+00	0.000

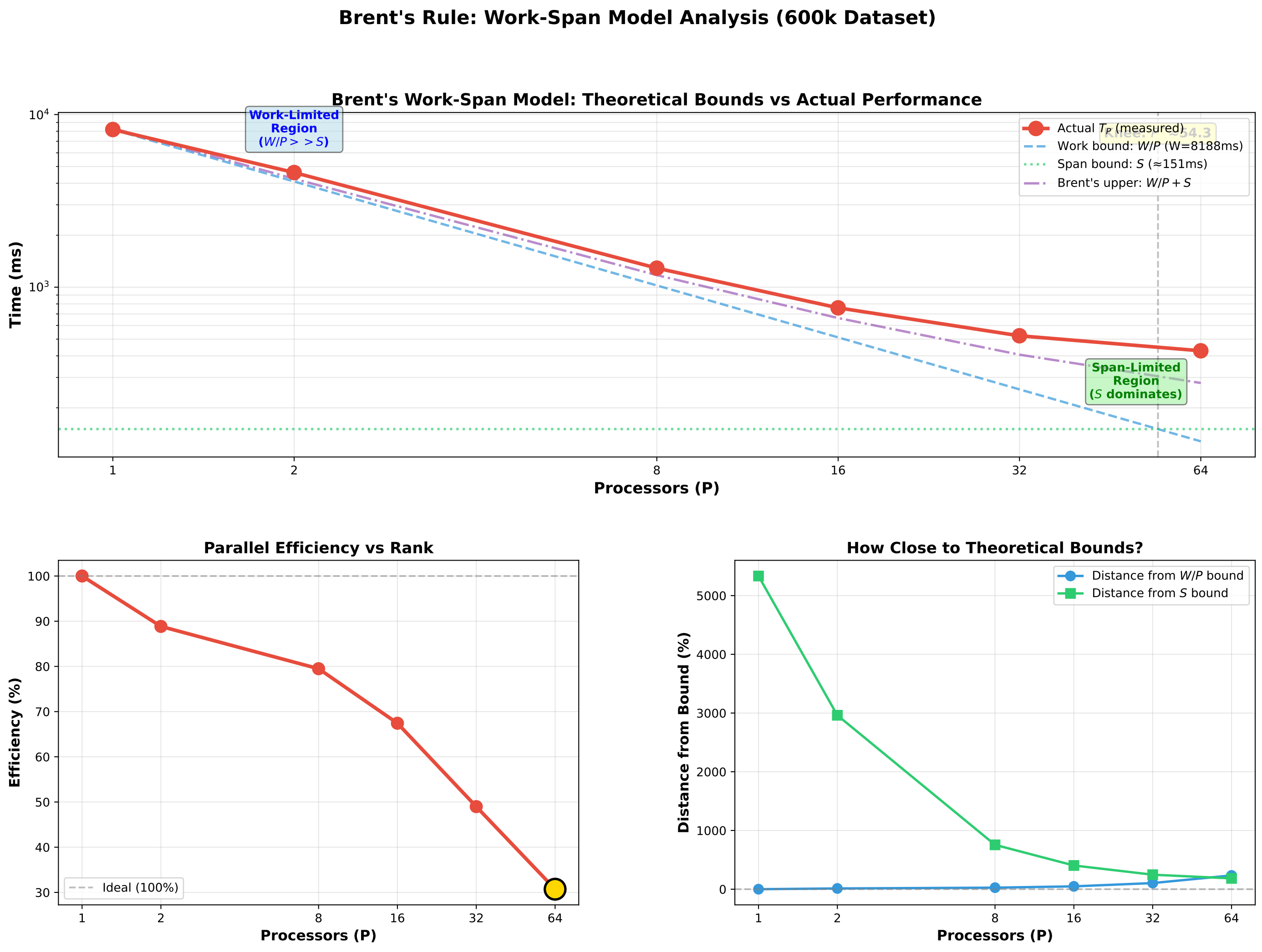
A.5 MPI Cost Analysis



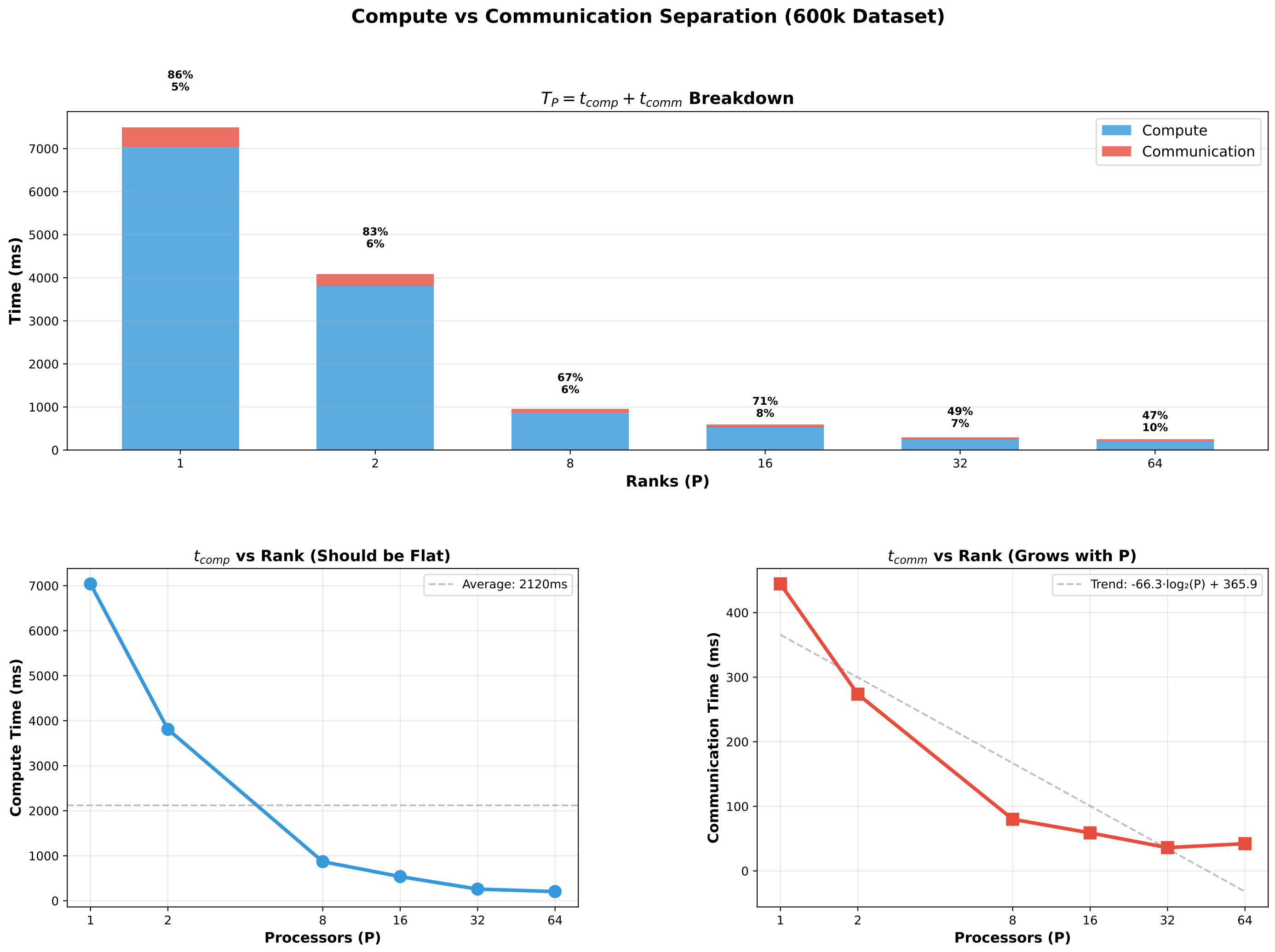


R=64: MPI Cost Model Summary - $t_{comm} \approx \alpha \cdot msgs + \beta \cdot bytes$

Phase	Msgs (median)	Bytes (median)	Time (ms)	α (ms/msg)	β (ms/byte)	R²
Phase A	23868110	2.50e+02	137	0.00e+00	0.00e+00	0.000
Phase B	12257883	3.48e+03	114	1.84e-07	-2.31e-03	0.804
Phase C	3425090	2.03e+05	3	0.00e+00	0.00e+00	0.000


Brent's Rule: Work-Span Model Analysis (600k Dataset)
Brent's Work-Span Model: Theoretical Bounds vs Actual Performance
Work-Limited Region ($W/P >> S$)
Span-Limited Region (S dominates)
Knee: $P^* \approx 54.3$
Actual T_P (measured)
Work bound: W/P (W=8188ms)
Span bound: S (≈151ms)
Brent's upper: $W/P + S$
Time (ms)
Processors (P)
Parallel Efficiency vs Rank
Efficiency (%)
Ideal (100%)
How Close to Theoretical Bounds?
Distance from W/P bound
Distance from S bound
Distance from Bound (%)

A.6 Supplementary Metrics (includes Brent's rule view)


Compute vs Communication Separation (600k Dataset)
$T_P = t_{comp} + t_{comm}$ Breakdown
Compute
Communication
Time (ms)
Ranks (P)
t_{comp} vs Rank (Should be Flat)
Average: 2120ms
Compute Time (ms)
Processors (P)
t_{comm} vs Rank (Grows with P)
Trend: -66.3·log₂(P) + 365.9
Communication Time (ms)
Processors (P)