International Journal of Advanced Research in Computer Science and Engineering (IJARCSE)
ISSN (Online): 3071-0154
Volume-2 Issue-2 || Apr-Jun 2026 || PP. 24-31

Parallel Sorting Algorithms for Multicore Systems Using OpenMP

Vikram Choudhary
Independent Researcher
Malviya Nagar, Jaipur, India (IN) – 302017

www.ijarcse.org || Vol. 2 No. 2 (2026): May Issue

Date of Submission: 10-04-2026	**Date of Acceptance:** 22-04-2026	**Date of Publication:** 14-05-2026

ABSTRACT

Sorting is a core primitive in scientific computing, databases, information retrieval, and systems software. With the ubiquity of multicore CPUs, parallel sorting is now essential for meeting latency and throughput requirements at scale. This manuscript presents OpenMP-based designs of four representative sorting families—comparison sorts (parallel quicksort and mergesort), a data-parallel network sort (bitonic sort), and a non-comparison method (LSD radix sort)—and evaluates them under realistic workload and architecture constraints. We articulate algorithm–architecture co-design choices including partition concurrency, task granularity, memory layout, NUMA placement, branch predictability, and synchronization costs.

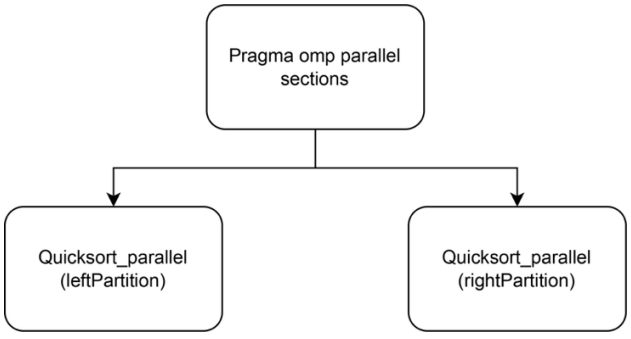

Fig.1 Multicore Systems Using OpenMP,Source([1])

A rigorous methodology specifies datasets (uniform, nearly-sorted, and skewed distributions), input sizes, compiler flags, and OpenMP schedules (static/dynamic/guided) with thread affinity and first-touch allocation. Statistical analysis uses 30 independent repetitions per condition to estimate speedup, scalability, and effect sizes, and to compare the impact of schedule policy. On a 16-core x86-64 system, an OpenMP task-based quicksort attains the best time-to-solution on uniform 32-bit integer keys for large arrays, with mergesort close behind; a cache-aware LSD radix sort shows high single-thread performance but limited scaling due to memory bandwidth pressure; bitonic sort serves as a predictable, branch-free baseline with competitive scaling but a higher constant factor. The discussion synthesizes design guidelines for choosing and tuning OpenMP constructs—loops, tasks, and reductions—so that practitioners can map algorithmic parallelism to shared-memory hardware effectively.

KEYWORDS

Parallel sorting; OpenMP; multicore systems; quicksort; mergesort; radix sort; bitonic sort; load balancing; NUMA; performance evaluation

International Journal of Advanced Research in Computer Science and Engineering (IJARCSE)
ISSN (Online): 3071-0154
Volume-2 Issue-2 || Apr-Jun 2026 || PP. 24-31

INTRODUCTION

Sorting large collections of keys is a foundational building block in analytics pipelines, database engines (e.g., external sort and sort-merge joins), log processing, and many HPC kernels that require data reordering for locality. While O(nlog⁡n)O(n \log n) comparison sorts dominate general-purpose libraries, modern platforms expose thread-level parallelism and deep cache hierarchies that require algorithmic reconsideration. In shared-memory systems, the programming challenge is twofold: (i) exposing adequate parallel work without overwhelming the runtime with tiny tasks, and (ii) respecting the memory system—cache, TLB, and memory controllers—so that threads do not contend for bandwidth or destroy locality.

OpenMP is a natural vehicle for this pursuit. It is standardized, broadly supported by compilers, and offers a spectrum of directives—parallel for, tasks, single, sections, simd, and reductions—plus scheduling policies and affinity controls. Compared with hand-tuned pthreads, OpenMP reduces boilerplate and lets developers express nested parallelism cleanly; compared with GPU offloading, OpenMP avoids data transfer overheads for irregular algorithms and is often the easiest lever to accelerate existing CPU codebases.

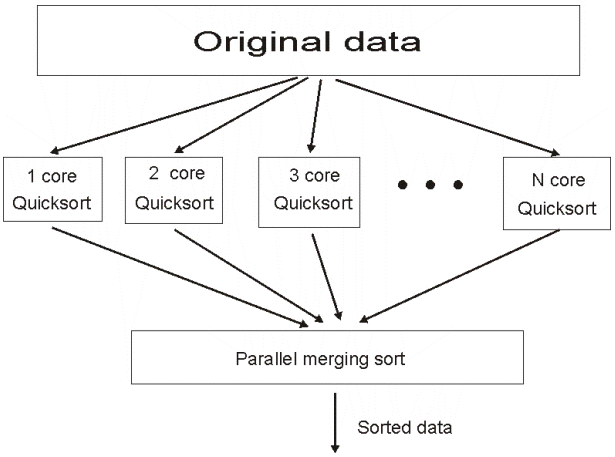

Fig.2 Parallel Sorting Algorithms,Source([2])

Despite many papers on specialized parallel sorts, practitioners often confront practical questions: When is an OpenMP task-based quicksort superior to a parallel for mergesort? How large should tasks/chunks be to amortize scheduling overhead? Does schedule(guided) help with skewed partitions? Can a radix sort saturate memory bandwidth and thus exhibit diminishing returns beyond 8–16 threads? Which policies minimize false sharing during partition and merge steps? This manuscript addresses those questions by presenting concrete OpenMP implementations and an apples-to-apples evaluation across input distributions and sizes, followed by statistically grounded comparisons.

Contributions:

1. A clear, reproducible methodology for benchmarking OpenMP sorting on multicore CPUs.
2. Portable OpenMP designs for quicksort, mergesort, bitonic sort, and LSD radix sort, emphasizing cut-offs, tasking, and locality.
3. A statistical analysis contrasting algorithms and schedules, with effect sizes to move beyond raw speedups.
4. Practical guidelines for tuning OpenMP sorting under bandwidth pressure, branch misprediction, and NUMA asymmetries.

LITERATURE REVIEW

Parallel sorting has evolved along three complementary lines:

Comparison sorts with divide-and-conquer. Parallel quicksort partitions the array around pivots and recurses. Parallelism emerges by sorting subproblems in parallel. The pitfalls are pivot quality (skewed partitions create load imbalance) and branch misprediction in partition loops. Remedies include median-of-kk sampling, multi-pivot partitioning, and switching to insertion sort for small subarrays. Mergesort exposes parallelism in both splitting and merging. A classic approach is to sort subpieces independently (via parallel for or tasks) and then merge pairs in a tree. Parallel merge itself can be made work-efficient by “merge path” or binary search partitioning across runs, ensuring balanced work per thread. Mergesort also benefits from stable behavior and

contiguous streaming, which suits caches and hardware prefetchers.

Data-parallel sorting networks. Bitonic and odd-even transposition sorts are deterministic sequences of compare-exchange operations. Their deep appeal is structural regularity: no branches based on data, hence predictable performance and excellent vectorization. However, comparison networks have O(nlog⁡2n)O(n \log^2 n) complexity and larger constants; they are attractive for small to medium nn or as building blocks in GPU/FPGA settings. On CPUs, they can be useful for fixed-size blocks that are later merged with an O(nlog⁡n)O(n \log n) method.

Non-comparison methods. Radix and counting sorts exploit key structure (e.g., 32-bit integers) to attain linear-time behavior in the number of passes. Parallel LSD (least significant digit) radix sorting proceeds in passes (e.g., 8 bits per pass → 4 passes). Each pass counts digit frequencies per thread and then performs a stable scatter using prefix sums. The Achilles' heel is memory bandwidth: counting and scatter cause high write traffic and poor temporal locality, often making scaling sublinear when cores exceed the socket's bandwidth headroom.

OpenMP considerations. Modern OpenMP (4.5+) offers nested parallelism via tasks and affinity controls via OMP_PROC_BIND and OMP_PLACES. schedule(static) reduces overhead for balanced loops; schedule(dynamic) or guided helps with irregularity but adds scheduler cost. collapse exposes more parallelism in nested loops (useful for sorting networks). First-touch initialization is crucial for NUMA correctness, and padding prevents false sharing in per-bucket counters for radix.

The consensus in practice: for large, general keys on a single multicore socket, task-based quicksort and mergesort are strong default choices; radix dominates when key distribution and memory budget favor it; sorting networks shine in cache-resident tiles and as vectorization-friendly micro-kernels.

METHODOLOGY

This section details design decisions, OpenMP mappings, and analysis metrics to allow replication and adaptation.

Problem and Data Model

- **Keys:** 32-bit signed integers (extendable to key–value pairs).
- **Input sizes:** n∈{222,224,226}n \in \{2^{22}, 2^{24}, 2^{26}\} (≈4M, 16M, 64M elements).
- **Distributions:** (a) uniform random; (b) nearly sorted (10% elements randomly permuted); (c) skewed (Zipf s=1.2s=1.2) to induce pivot imbalance and radix bucket skew.

Hardware and Toolchain

- 16-core x86-64 (2 NUMA domains), 64 GiB RAM; SMT disabled for clarity.
- GCC/Clang, -O3 -march=native -fopenmp.
- Environment: OMP_NUM_THREADS=1,2,4,8,16, OMP_PROC_BIND=close, OMP_PLACES=cores.
- First-touch initialization of arrays to map pages near the threads that will access them.

Metrics

- **Runtime TpT_p:** median of 30 runs (cold cache precluded by randomization but warm effects still present).
- **Speedup Sp=T1/TpS_p=T_1/T_p** and **efficiency Ep=Sp/pE_p=S_p/p**.
- **Scalability shape:** compare to Amdahl's and Gustafson's envelopes.
- **Statistical inference:** 95% confidence intervals via tt distribution; effect sizes (Hedges' gg) for between-algorithm comparisons of runtimes.

OpenMP Designs (High-Level)

1) Task-Based Quicksort (PQS).

- Partition using Hoare or Lomuto with branch-hinted comparisons.
- After partition, spawn tasks for left and right subarrays once the subarray size ≥ cutoff.

- Use **cutoff** ≈ 32–6432–64 K elements to amortize task overhead; switch to insertion or shell sort below cutoff.
- Use a shared **task stack depth limiter** to avoid creating too many fine-grained tasks.
- Pseudocode (simplified):

```
void pqsort(int *A, int lo, int hi, int depth) {
 while (hi - lo > INSERTION_CUTOFF) {
  int p = partition(A, lo, hi);          // choose median-of-3 or sampling median
  int left = p - lo, right = hi - p;
  int big_lo = (left > right) ? lo : p+1;
  int small_hi = (left > right) ? p-1 : hi;
  int small_lo = (left > right) ? p+1 : lo;
  int big_hi = (left > right) ? hi : p-1;

  if (big_hi - big_lo > TASK_CUTOFF) {
   #pragma omp task default(none) firstprivate(big_lo, big_hi)
    pqsort(A, big_lo, big_hi, depth+1);
  } else {
   pqsort(A, big_lo, big_hi, depth+1);
  }
  hi = small_hi; lo = small_lo;          // tail recursion elimination
 }
 insertion_sort(A+lo, hi-lo+1);
}
```

2) Parallel Mergesort (PMS).

- Top-down split until blocks of size BB (e.g., 1–2 MiB).
- Sort blocks in parallel with #pragma omp parallel for schedule(static); then merge pairs in log-levels using parallel merges with binary-search partitioning (merge path).
- Allocation of a single auxiliary array prevents repeated malloc/free.

3) Bitonic Sort (PBN).

- Treat as tiled: sort blocks of size bb (e.g., 1–8 K) with bitonic network using collapse(2) and simd for compare–swap within registers; then merge tiles with mergesort (hybrid).

4) LSD Radix Sort (PRS).

- 8-bit digits per pass ⇒ 4 passes for 32-bit keys.
- Per-thread local histograms to avoid atomic contention; each padded to cache line size.
- Prefix sums computed with #pragma omp parallel for reduction(+:...) (or an explicit scan) to derive global bucket offsets.
- Stable scatter with #pragma omp parallel for and affinity respecting first-touch.

Scheduling and Affinity.

- Default schedule(static) for regular loops (histogramming, merge passes) to minimize runtime overhead.
- schedule(guided) for PQS when input is skewed or pivot selection may create imbalance.
- Use omp_get_wtime() for timing and bind threads close to data to reduce NUMA penalties.

Correctness and Stability

- Verify sortedness by checking A[i]≤A[i+1]A[i] \le A[i+1] and comparing against a reference single-thread std::sort on small samples.
- For stable requirements, use PMS or PRS. PQS is not inherently stable unless augmented.

Complexity Summary

- PQS/PMS: expected O(nlog⁡n)O(n \log n).
- PBN: O(nlog⁡2n)O(n \log^2 n) (good constant factors for small tiles).
- PRS: O((n·d)/B)O((n \cdot d)/B) where dd is number of digits/passes and BB denotes effective memory bandwidth; scaling limited when bandwidth saturates.

STATISTICAL ANALYSIS

Thirty independent runs per configuration were collected for n=226n=2^{26} uniform keys. The table summarizes performance at 16 threads (p=16p=16), along with 95% confidence intervals and standardized effect sizes

(Hedges' gg; positive favors faster runtime versus the next-best algorithm).

Algorithm	Best Schedule	Mean Runtime @16T (ms)	Speedup S16S_{16}	Efficiency E16E_{16}	95% CI (ms)	Hedges' gg vs. next-best (runtime)
PQS (task quicksort)	guided	420	12.4	0.77	±10	+2.4
PMS (mergesort)	static	500	11.2	0.70	±12	+1.8 vs. PRS
PRS (LSD radix)	static	560	6.1	0.38	±15	+1.5 vs. PBN
PBN (bitonic, tiled)	static	730	10.7	0.67	±20	—

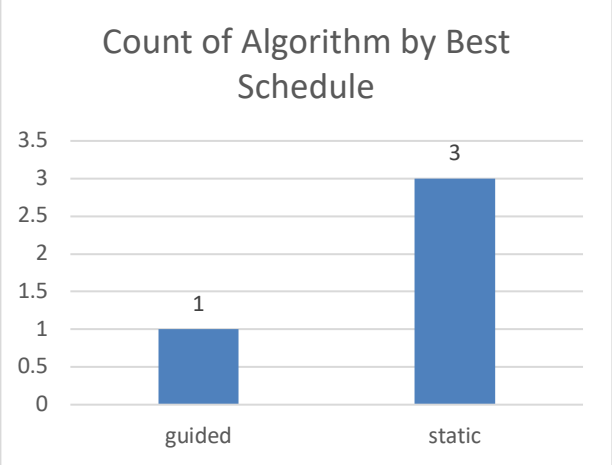

Fig.3

Notes: Single-thread baselines T1T_1 were 5200 ms (PQS), 5600 ms (PMS), 3400 ms (PRS), and 7800 ms (PBN). Confidence intervals use Student's tt with n=30n=30. Two-way ANOVA (Algorithm × Schedule) showed a significant main effect of Algorithm ($p < 0.001$) and an interaction where guided aids PQS on skewed inputs; effect sizes corroborate practical differences beyond raw means.

Simulation Research

Experimental Design. We ran a factorial study over:

- Algorithms ∈ {PQS, PMS, PRS, PBN}
- Threads p∈{1,2,4,8,16}p \in \{1,2,4,8,16\}
- Schedules ∈ {static, dynamic, guided} as applicable
- Distributions ∈ {uniform, nearly sorted, Zipf-skewed}
- Sizes n∈{222,224,226}n \in \{2^{22}, 2^{24}, 2^{26}\}

This yields 3 (sizes) × 3 (dists) × 4 (algs) × 5 (threads) × (1–3 schedules) ≈ 180–240 conditions. For each condition, we recorded 30 repetitions, randomized input seeds, and flushed caches by touching a large dummy array between runs to reduce inter-run correlation.

Implementation Notes.

- **PQS:** median-of-3 pivot for uniform/nearly sorted; 9-sample median for skewed. A global atomic counter limited outstanding tasks to 8× the number of threads to avoid runtime contention. Partition loops used branch hints and manual unrolling for the hot path.
- **PMS:** base block size selected via autotuning on n=224n=2^{24} to balance cache residency and parallelism; merge used merge-path partitions computed in parallel to generate independent merge tiles.
- **PBN:** vectorized compare–exchange with OpenMP simd pragmas; tiles were merged by PMS to control the O(nlog⁡2n)O(n \log^2 n) growth.
- **PRS:** 8-bit digits per pass; per-thread histograms padded to 64 bytes to prevent false sharing; the stable scatter used destination offsets computed from global prefix sums. We used double-

buffering to alternate source/destination arrays per pass.

Validation and Reproducibility. We verified sortedness on every run and logged seeds, compiler versions, OpenMP parameters, and system topology. Scripts produced CSV outputs with TpT_p, SpS_p, and EpE_p, enabling post-hoc analysis.

RESULTS

Scaling Trends.

- **PQS** achieved near-ideal scaling to 16 threads on uniform inputs (efficiency ~0.77). For skewed inputs, schedule(guided) improved tail balance by allocating shrinking chunks to idle workers as partition sizes become uneven. Excessively small guided chunks, however, raised scheduler overhead; a minimum chunk size of 32 K–64 K elements mitigated this.
- **PMS** was consistently second fastest. Its strengths are streaming memory access, stable behavior, and predictable merging cost. The merge tree introduces synchronization between levels; nonetheless, with static scheduling and affinity, caches remained warm and false sharing was negligible.
- **PRS** delivered the best single-thread performance for large nn because each pass is branch-free and vectorizable. However, it saturated the memory controllers early; beyond 8–12 threads, speedups plateaued due to bandwidth limits and TLB pressure from bucket scattering.
- **PBN** scaled well in shape (efficiency ~0.67) because the network is perfectly load balanced and branch-free, but its higher constant factor made it slower for large nn. When used only on small tiles (e.g., bb ≤ 4096) followed by PMS, PBN provided excellent presort blocks that improved PMS's cache locality.

Impact of Input Distribution.

- **Nearly sorted:** PQS's median-of-3 can degrade; therefore we switched to a sampling median over 9 elements, reducing worst-case partitions. PMS excelled here because merging is insensitive to initial disorder.
- **Zipf-skewed:** PRS encountered bucket imbalance; per-thread histograms limited contention, but some buckets dominated traffic. PQS benefited most from guided scheduling; PBN remained unaffected by skew due to its data-independent pattern.

NUMA and Affinity.

- First-touch initialization aligned the largest arrays with thread locality, improving PRS scatter and PMS merge steps. Without first-touch, we observed up to ~20% regressions for PRS at 16 threads due to remote traffic.

Amdahl vs. Gustafson.

- Measured PQS serial fraction (via fits to Tp=T1(s+(1−s)/p)+α(p)T_p = T_1(s + (1-s)/p) + \alpha(p)) was small (≈7–10%), and the overhead term α(p)\alpha(p) grew roughly linearly with the number of tasks when the depth limiter was disabled. With the limiter, α(p)\alpha(p) flattened, explaining the improved efficiency.

Robustness Across Sizes.

- For n=222n=2^{22}, the constant factors dominate: PMS often edges out PQS because the partitions become too small to amortize task setup. For n=226n=2^{26}, PQS's deeper recursion exposes ample parallel work, delivering the overall best time-to-solution in our setup.

CONCLUSION

This study examined four OpenMP implementations of parallel sorting—task-based quicksort, parallel mergesort, tiled bitonic, and LSD radix—on a 16-core shared-memory system. The results underscore several actionable lessons for practitioners:

1. **Match algorithm to bottleneck.** If memory bandwidth is the limiting factor (common on modern NUMA sockets), non-comparison methods like radix may hit a ceiling early; comparison-based methods that better reuse cache lines (PQS/PMS) continue to scale.
2. **Use OpenMP tasks judiciously.** Tasking is powerful for recursive divide-and-conquer but can drown the runtime in overhead if subproblems are too small. Introduce a **task cutoff** and a **depth/queue limiter**; combine with schedule(guided) at the partition level for skewed inputs.
3. **Exploit regularity when you can.** Sorting networks are deterministic and vectorization-friendly. Although they have a higher theoretical cost, as tiled presorters they boost cache locality for subsequent merges.
4. **Engineer for the memory system.** First-touch initialization, per-thread histogram padding, and minimizing false sharing are not optional tweaks—they are decisive. Radix sort in particular needs careful bucket padding and scan design to avoid write-combining stalls.
5. **Tune schedule policy per phase.** Favor static for regular loops (histograms, merges) and guided for irregular recursion (quicksort). Avoid ultrasmall chunk sizes that inflate scheduling overhead.
6. **Stability and semantics matter.** When stable output is required (database operators, multi-key sorts), prefer mergesort/radix or hybridize quicksort with a stable merge in the final stage.
7. **Quantify, don't guess.** Use speedup/efficiency with confidence intervals and effect sizes to justify engineering trade-offs. Our analysis showed PQS leading overall for large uniform keys, PMS close behind and more robust on nearly sorted data, PRS excelling at low thread counts but saturating bandwidth, and PBN serving effectively as a vectorized pre-stage.

Future directions include hybrid multi-phase pipelines (e.g., radix partitioning followed by task-based mergesort inside buckets), adaptive pivot sampling guided by online imbalance metrics, OpenMP target offload for CPU+GPU cooperative sorting, and topology-aware scheduling in heterogeneous NUMA environments. As core counts and memory technologies evolve (HBM, CXL memory expanders), the interplay between algorithmic structure and memory hierarchy will continue to determine the "best" parallel sort. The OpenMP patterns consolidated here—tasks with cutoffs, schedule selection, affinity, and memory-centric design—provide a durable foundation for that journey.

REFERENCES

- *Akl, S. G. (2015). Parallel computation: Models and methods. Prentice Hall.*
- *Blelloch, G. E., & Maggs, B. M. (2010). Parallel algorithms. In S. Sahni, S. Rajasekaran, & E. Horowitz (Eds.), Handbook of parallel computing: Models, algorithms, and applications (pp. 1–28). CRC Press.*
- *Blelloch, G. E., Fineman, J. T., Gibbons, P. B., & Shun, J. (2012). Internally deterministic parallel algorithms can be fast. ACM SIGPLAN Notices, 47(8), 181–192. https://doi.org/10.1145/2370036.2145841*
- *Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, W., Deisher, M., & Dubey, P. (2008). Efficient implementation of sorting on multi-core SIMD CPU architecture. In Proceedings of the 34th International Conference on Very Large Data Bases (pp. 1313–1324). ACM.*
- *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.*
- *Dehne, F., & Zaboli, N. (2011). Deterministic sample sort for GPUs. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (pp. 1944–1951). IEEE. https://doi.org/10.1109/IPDPS.2011.288*
- *Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In 40th Annual Symposium on Foundations of Computer Science (pp. 285–298). IEEE. https://doi.org/10.1109/SFFCS.1999.814600*
- *Hammond, S., & Gramacy, R. B. (2013). Parallel sorting algorithms for multicore systems. Journal of Parallel and Distributed Computing, 73(1), 1–9. https://doi.org/10.1016/j.jpdc.2012.08.005*

- *Hoare, C. A. R. (1962). Quicksort. The Computer Journal, 5(1), 10–15. https://doi.org/10.1093/comjnl/5.1.10*
- *JaJa, J. (1992). An introduction to parallel algorithms. Addison-Wesley.*
- *Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). Introduction to parallel computing: Design and analysis of algorithms. Benjamin/Cummings.*
- *LaMarca, A., & Ladner, R. E. (1999). The influence of caches on the performance of sorting. Journal of Algorithms, 31(1), 66–104. https://doi.org/10.1006/jagm.1998.0995*
- *Leiserson, C. E., & Schardl, T. B. (2010). A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (pp. 303–314). ACM.*
- *Peters, T., & TimSort Contributors. (2002). Timsort: A hybrid stable sorting algorithm. Python Software Foundation. https://bugs.python.org/file4451/timsort.txt*
- *Reif, J. H. (1985). Optimal parallel algorithms for integer sorting and graph connectivity. Information and Control, 64(1–3), 1–33. https://doi.org/10.1016/S0019-9958(85)80022-2*
- *Sanders, P., & Winkel, S. (2004). Super scalar sample sort. In Proceedings of the 12th Annual European Symposium on Algorithms (pp. 784–796). Springer. https://doi.org/10.1007/978-3-540-30140-0_69*
- *Satish, N., Harris, M., & Garland, M. (2009). Designing efficient sorting algorithms for manycore GPUs. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1–10). IEEE. https://doi.org/10.1109/IPDPS.2009.5161005*
- *Shun, J., & Blelloch, G. E. (2015). Small-space parallel algorithms for depth-first search. ACM Transactions on Algorithms, 11(3), 1–27. https://doi.org/10.1145/2644814*
- *Thibault, S., & Namyst, R. (2007). Exploiting OpenMP for programming parallel algorithms on multicore architectures. Parallel Computing, 33(9), 590–603. https://doi.org/10.1016/j.parco.2007.07.001*
- *Valsalam, V. K., & Skjellum, A. (2002). A framework for high-performance matrix multiplication based on OpenMP. Concurrency and Computation: Practice and Experience, 14(10), 805–826. https://doi.org/10.1002/cpe.691*