МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности (ИКНК)
Направление: «Математика и компьютерные науки (Искусственный интеллект и машинное обучение)

КУРСОВОЙ ПРОЕКТ

по дисциплине: «суперкомпьютер»
на тему: «Параллельная сортировка»

студент группы 5140201/50301
Дастанбу Матин

Преподаватель: ***Лукашин Алексей Андреевич***

САНКТ-ПЕТЕРБУРГ 2025

Contents

Abstract....2
1. Introduction....3
2. Description of Implemented Methods....4
2.1 Python + MPI....4
2.2 C + MPI....5
2.3 C + OpenMP....7
2.4 C + pthreads....8
3. Experimental Results and Performance Analysis....10
3.1 Best Runtime Comparison Across Implementations....10
3.2 Shared Memory Performance: pthreads vs OpenMP....11
3.2.2 Speedup Analysis....12
3.2.3 Efficiency Analysis....13
3.3 Distributed Memory Performance: C + MPI vs Python + mpi4py....14
3.3.1 Runtime vs MPI Ranks....14
3.3.2 Runtime vs Nodes....15
Conclusion....17

Abstract

High-Performance Computing (HPC) systems rely heavily on parallel programming paradigms to efficiently process large datasets. Sorting is a fundamental operation widely used in scientific computing, data analytics, and simulation pipelines, and therefore serves as an ideal benchmark for evaluating parallel performance. This project investigates and compares multiple parallel implementations of a sorting algorithm using different programming models: **C with pthreads**, **C with OpenMP**, **C with MPI**, and **Python with mpi4py**.

The primary objective of this study is to analyze how different parallelization strategies behave on a modern HPC cluster when applied to the same computational problem. All implementations operate on an identical input consisting of 500 randomly ordered integers, ensuring correctness comparability. Experiments were conducted on the *tornado* cluster at SPbSTU, featuring multi-core nodes connected via high-speed InfiniBand interconnects.

Shared-memory approaches (pthreads and OpenMP) were evaluated on a single node using varying numbers of threads, while distributed-memory approaches (MPI and mpi4py) were tested across multiple nodes with increasing numbers of processes (MPI ranks). Execution time, scalability trends, and overhead effects were carefully measured and visualized using Python-based performance plots.

The results demonstrate that shared-memory approaches provide extremely low execution times for small problem sizes, but their scalability is limited to a single node. MPI-based solutions, although incurring higher communication overhead, enable multi-node execution and demonstrate the ability to scale beyond a single machine. The Python mpi4py implementation shows significantly higher execution times compared to its C-based MPI counterpart, highlighting the cost of interpreter overhead and higher-level abstractions.

Overall, this project provides a comprehensive, practical comparison of parallel programming paradigms on real HPC hardware, emphasizing the trade-offs between ease of development, performance, and scalability.

1. Introduction

Parallel programming is a cornerstone of modern high-performance computing. As processor clock speeds have plateaued, performance improvements increasingly rely on parallel execution across multiple cores and nodes. Consequently, understanding the strengths and limitations of different parallel programming models is essential for developing efficient scientific and engineering applications.

Sorting is a fundamental computational task that appears in numerous domains, including database systems, numerical simulations, machine learning pipelines, and large-scale data processing. Although sorting algorithms are well understood in the sequential case, their parallelization introduces challenges such as data partitioning, synchronization, communication overhead, and load balancing. For these reasons, sorting is frequently used as a benchmark problem in parallel computing education and research.

This project focuses on implementing and evaluating a parallel sorting algorithm using four widely used paradigms:

- **pthreads (POSIX Threads)**: a low-level shared-memory threading API in C
- **OpenMP**: a high-level directive-based shared-memory model
- **MPI (Message Passing Interface)**: the de facto standard for distributed-memory parallelism
- **mpi4py**: a Python binding for MPI

These models represent fundamentally different approaches to parallelism. pthreads and OpenMP exploit shared memory within a single node, enabling fast communication but limiting scalability. MPI and mpi4py use explicit message passing, allowing execution across multiple nodes at the cost of communication overhead.

All experiments were conducted on the same HPC cluster using identical input data to ensure fair comparison. The input size (N = 500) was intentionally kept small to emphasize overhead effects, which are often overlooked but critically important in real HPC applications, especially when scaling to large numbers of processes.

2. Description of Implemented Methods

In this work, several parallel sorting implementations were developed and analyzed using different parallel programming models: **Python + MPI (mpi4py)**, **C + MPI**, **C + OpenMP**, and **C + pthreads**. All implementations perform sorting of the same input dataset consisting of 500 integers, which allows a fair comparison of performance, scalability, and programming complexity.

2.1 Python + MPI

In the first part of the work, a sorting algorithm was implemented in **Python** using the **mpi4py** library. This approach combines the simplicity and readability of Python with the distributed computing capabilities provided by MPI.

The algorithm operates as follows:

1. The process with rank 0 reads the input data from a file.
2. The input array is divided into equal chunks according to the number of MPI processes.
3. These chunks are distributed among processes using MPI_Scatter.
4. Each process performs a local sort on its portion of the data.
5. The sorted chunks are collected on the root process using MPI_Gather.
6. The root process performs a final merge and writes the sorted result to an output file.

Code fragment:

```
from mpi4py import MPI
import argparse
import heapq
import time
def read_ints(path: str):
    with open(path, "r") as f:
        return list(map(int, f.read().split()))
def write_ints(path: str, arr):
    with open(path, "w") as f:
        f.write(" ".join(map(str, arr)) + "\n")
def chunkify(a, size):
    n = len(a)
    base = n // size
    rem = n % size
    chunks = []
    start = 0
    for r in range(size):
        add = 1 if r < rem else 0
        end = start + base + add
        chunks.append(a[start:end])
        start = end
    return chunks
def kway_merge(sorted_lists):
    # heap: (value, list_id, index_in_list)
    heap = []
    for li, lst in enumerate(sorted_lists):
        if lst:
            heap.append((lst[0], li, 0))
    heapq.heapify(heap)
    out = []
    while heap:
        val, li, idx = heapq.heappop(heap)
        out.append(val)
        idx += 1
        if idx < len(sorted_lists[li]):
            heapq.heappush(heap, (sorted_lists[li][idx], li,
idx))
    return out
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--input", required=True)
    ap.add_argument("--output", required=True)
```

```
    ap.add_argument("--echo", action="store_true",
help="print sorted numbers to stdout (use only for small
N)")
    args = ap.parse_args()
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    t0 = MPI.Wtime()
    if rank == 0:
        data = read_ints(args.input)
        N = len(data)
        chunks = chunkify(data, size)
    else:
        N = None
        chunks = None
    N = comm.bcast(N, root=0)
    local = comm.scatter(chunks, root=0)
    local.sort()
    gathered = comm.gather(local, root=0)
    if rank == 0:
        sorted_all = kway_merge(gathered)
        t1 = MPI.Wtime()
        print(f"mpi4py_sort: N={N} tasks={size}
time_s={t1 - t0:.6f}")
        write_ints(args.output, sorted_all)
        if args.echo:
            print("OUTPUT:")
            print(" ".join(map(str, sorted_all)))
if __name__ == "__main__":
    main()
```

This implementation demonstrates the ease of using MPI in Python. However, it introduces additional overhead related to Python interpretation and high-level data handling, which negatively affects performance at larger process counts.

2.2 C + MPI

The second version was implemented in the **C programming language** using the **MPI** library. This approach significantly reduces runtime overhead and provides full control over memory management and communication patterns.

In this implementation:

- the process with rank 0 reads the input data;
- the size of the dataset is broadcast to all processes using MPI_Bcast;
- the data is distributed using MPI_Scatter;
- each process sorts its local portion of the array;
- the sorted segments are collected on the root process using MPI_Gather.

Code fragment:

```
cat > src/sort_mpi.c << 'EOF'
static int cmp_int(const void *a, const void *b){
    int x = *(const int*)a, y = *(const int*)b;
    return (x > y) - (x < y);
}
static inline double now_s(void){
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + 1e-9*(double)ts.tv_nsec;
}
typedef struct { int val; int src; int idx; } heap_item;
static void heap_swap(heap_item *h, int i, int j){
heap_item t=h[i]; h[i]=h[j]; h[j]=t; }
static void heap_sift_down(heap_item *h, int n, int i){
    for(;;){
        int l=2*i+1, r=2*i+2, m=i;
        if(l<n && h[l].val < h[m].val) m=l;
        if(r<n && h[r].val < h[m].val) m=r;
        if(m==i) break;
        heap_swap(h,i,m);
        i=m;
    }
}
static void heap_sift_up(heap_item *h, int i){
    while(i>0){
        int p=(i-1)/2;
        if(h[p].val <= h[i].val) break;
        heap_swap(h,p,i);
        i=p;
    }
}
```

```
static void heap_push(heap_item *h, int *n, heap_item
it){
    h[*n]=it;
    (*n)++;
    heap_sift_up(h, (*n)-1);
}
static heap_item heap_pop(heap_item *h, int *n){
    heap_item top=h[0];
    (*n)--;
    h[0]=h[*n];
    heap_sift_down(h, *n, 0);
    return top;
}
static void merge_sorted_blocks(int *all, int *counts, int
*displs, int P, int *out){
    heap_item *heap = (heap_item*)malloc((size_t)P *
sizeof(heap_item));
    int hn = 0;
    for(int p=0; p<P; p++){
        if(counts[p] > 0){
            heap_item it;
            it.src = p;
            it.idx = 0;
            it.val = all[displs[p]];
            heap_push(heap, &hn, it);
        }
    }
    int N = 0;
    for(int p=0; p<P; p++) N += counts[p];
    for(int i=0; i<N; i++){
        heap_item cur = heap_pop(heap, &hn);
        out[i] = cur.val;
        int p = cur.src;
        int next_idx = cur.idx + 1;
        if(next_idx < counts[p]){
            heap_item nxt;
            nxt.src = p;
            nxt.idx = next_idx;
            nxt.val = all[displs[p] + next_idx];
            heap_push(heap, &hn, nxt);
        }
    }
    free(heap);
}
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    int rank=0, size=1;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    const char *in_path = NULL;
    const char *out_path = NULL;
    for(int i=1;i<argc;i++){
        if(!strcmp(argv[i],"--input") && i+1<argc) in_path
= argv[++i];
        else if(!strcmp(argv[i],"--output") && i+1<argc)
out_path = argv[++i];
    }
    if(!in_path || !out_path){
        if(rank==0) fprintf(stderr,"Usage: %s --input
data/input_500.txt --output results/mpi_sorted.txt\n",
argv[0]);
        MPI_Finalize();
        return 2;
    }
    int *full = NULL;
    int N = 0;
    if(rank==0){
        FILE *f = fopen(in_path,"r");
        if(!f){ perror("fopen input");
MPI_Abort(MPI_COMM_WORLD, 1); }
        int cap=1024;
full=(int*)malloc((size_t)cap*sizeof(int));
        while(1){
            int x;
            if(fscanf(f,"%d",&x)!=1) break;
            if(N>=cap){ cap*=2;
full=(int*)realloc(full,(size_t)cap*sizeof(int)); }
            full[N++] = x;
        }
        fclose(f);
    }
    MPI_Bcast(&N, 1, MPI_INT, 0,
MPI_COMM_WORLD);
    if(N==0){ MPI_Finalize(); return 0; }
    int *counts = (int*)malloc((size_t)size*sizeof(int));
    int *displs = (int*)malloc((size_t)size*sizeof(int));
    for(int p=0; p<size; p++){
        int l = (p * N) / size;
        int r = ((p+1) * N) / size;
        counts[p] = r - l;
        displs[p] = l;
    }
    int local_n = counts[rank];
    int *local = (int*)malloc((size_t)local_n*sizeof(int));
    MPI_Barrier(MPI_COMM_WORLD);
    double t0 = now_s();
    MPI_Scatterv(full, counts, displs, MPI_INT,
                 local, local_n, MPI_INT,
                 0, MPI_COMM_WORLD);
    qsort(local, (size_t)local_n, sizeof(int), cmp_int);
    int *all_sorted = NULL;
    if(rank==0) all_sorted =
(int*)malloc((size_t)N*sizeof(int));
    MPI_Gatherv(local, local_n, MPI_INT,
                all_sorted, counts, displs, MPI_INT,
                0, MPI_COMM_WORLD);
    if(rank==0){
        int *out = (int*)malloc((size_t)N*sizeof(int));
        merge_sorted_blocks(all_sorted, counts, displs,
size, out);
        FILE *g = fopen(out_path,"w");
        if(!g){ perror("fopen output");
MPI_Abort(MPI_COMM_WORLD, 2); }
        for(int i=0;i<N;i++){
            fprintf(g, "%d%s", out[i], (i+1<N) ? " " : "\n");
        }
        fclose(g);
        free(out);
        free(all_sorted);
        free(full);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    double t1 = now_s();
    if(rank==0){
        printf("mpi_sort: N=%d tasks=%d
time_s=%.6f\n", N, size, (t1-t0));
    }
    free(local);
    free(counts);
    free(displs);
```

```
    MPI_Finalize();
    return 0;
}
EOF
```

This implementation demonstrates the best scalability and performance among all tested approaches and is well suited for distributed-memory high-performance computing systems.

2.3 C + OpenMP

The **OpenMP** version follows the **shared-memory** programming model. A single process is executed on a computing node, and parallelism is achieved through the use of multiple threads.

Key characteristics of this approach:

- all data resides in shared memory;
- parallelism is expressed using OpenMP directives;
- the number of threads is controlled by the OMP_NUM_THREADS environment variable.

Code fragment:

```
cat > src/sort_openmp.c << 'EOF'
static int cmp_int(const void *p, const void *q) {
    int x = *(const int*)p, y = *(const int*)q;
    return (x > y) - (x < y);
}
static inline double now_s(void){
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + 1e-9*(double)ts.tv_nsec;
}
/* k-way merge for small N=500 */
static void merge_k(int *a, int n, int k){
    int *L = (int*)malloc((size_t)k * sizeof(int));
    int *R = (int*)malloc((size_t)k * sizeof(int));
    int *idx = (int*)malloc((size_t)k * sizeof(int));
    int *out = (int*)malloc((size_t)n * sizeof(int));
    for(int t=0;t<k;t++){
        L[t] = (t*n)/k;
        R[t] = ((t+1)*n)/k;
        idx[t] = L[t];
    }
    for(int i=0;i<n;i++){
        int best_t = -1;
        int best_v = 0;
        for(int t=0;t<k;t++){
            if(idx[t] < R[t]){
                int v = a[idx[t]];
                if(best_t < 0 || v < best_v){
                    best_t = t;
                    best_v = v;
                }
            }
        }
        out[i] = best_v;
        idx[best_t]++;
    }
    memcpy(a, out, (size_t)n*sizeof(int));
    free(L); free(R); free(idx); free(out);
}
int main(int argc, char **argv){
    const char *in_path = NULL, *out_path = NULL;
    int threads = 1;
    for(int i=1;i<argc;i++){
        if(!strcmp(argv[i],"--input") && i+1<argc) in_path
= argv[++i];
        else if(!strcmp(argv[i],"--output") && i+1<argc)
out_path = argv[++i];
        else if(!strcmp(argv[i],"--threads") && i+1<argc)
threads = atoi(argv[++i]);
    }
    if(!in_path || !out_path || threads < 1){
        fprintf(stderr,"Usage: %s --input
data/input_500.txt --output results/out.txt --threads
T\n", argv[0]);
        return 2;
    }
    FILE *f = fopen(in_path,"r");
    if(!f){ perror("fopen input"); return 1; }
    int cap = 1024, n = 0;
    int *a = (int*)malloc((size_t)cap*sizeof(int));
    while(1){
        int x;
        if(fscanf(f,"%d",&x) != 1) break;
        if(n >= cap){
            cap *= 2;
            a = (int*)realloc(a, (size_t)cap*sizeof(int));
        }
        a[n++] = x;
```

```
    }
    fclose(f);
    if(threads > n) threads = n;
    double t0 = now_s();
    #pragma omp parallel num_threads(threads)
    {
        int tid = omp_get_thread_num();
        int T = omp_get_num_threads();
        int l = (tid * n) / T;
        int r = ((tid + 1) * n) / T;
        qsort(a + l, (size_t)(r - l), sizeof(int), cmp_int);
    }
    merge_k(a, n, threads);
    double t1 = now_s();
    FILE *g = fopen(out_path,"w");
    if(!g){ perror("fopen output"); return 1; }
    for(int i=0;i<n;i++){
        fprintf(g, "%d%s", a[i], (i+1<n) ? " " : "\n");
    }
    fclose(g);
    printf("openmp_sort: n=%d threads=%d
time_s=%.6f\n", n, threads, (t1-t0));
    free(a);
    return 0;
}
EOF
```

OpenMP greatly simplifies parallel programming compared to manual thread management. However, this approach is limited to a single node and cannot be directly scaled to multi-node systems.

2.4 C + pthreads

The final implementation uses the **pthreads** library, which provides low-level control over thread creation and synchronization.

In this approach:

- threads are created manually;
- each thread processes a specific portion of the data;
- synchronization is handled explicitly by the programmer.

Code fragment:

```
cat > src/sort_pthreads.c << 'EOF'
typedef struct {
    int *a;
    int l, r;
} task_t;
static int cmp_int(const void *p, const void *q) {
    int x = *(const int*)p, y = *(const int*)q;
    return (x > y) - (x < y);
}
static void* worker(void *arg) {
    task_t *t = (task_t*)arg;
    qsort(t->a + t->l, (size_t)(t->r - t->l), sizeof(int),
cmp_int);
    return NULL;
}
static inline double now_s(void){
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + 1e-9*(double)ts.tv_nsec;
}
/* k-way merge for small N (N=500 here) */
static void merge_k(int *a, int n, int k){
    int *L = (int*)malloc((size_t)k * sizeof(int));
    int *R = (int*)malloc((size_t)k * sizeof(int));
    int *idx = (int*)malloc((size_t)k * sizeof(int));
    int *out = (int*)malloc((size_t)n * sizeof(int));
    for(int t=0;t<k;t++){
        L[t] = (t*n)/k;
        R[t] = ((t+1)*n)/k;
        idx[t] = L[t];
    }
    for(int i=0;i<n;i++){
        int best_t = -1;
        int best_v = 0;
        for(int t=0;t<k;t++){
            if(idx[t] < R[t]){
                int v = a[idx[t]];
                if(best_t < 0 || v < best_v){
                    best_t = t;
                    best_v = v;
                }
            }
        }
        out[i] = best_v;
```

```
        idx[best_t]++;
    }
    memcpy(a, out, (size_t)n*sizeof(int));
    free(L); free(R); free(idx); free(out);
}

int main(int argc, char **argv){
    const char *in_path = NULL, *out_path = NULL;
    int threads = 1;
    for(int i=1;i<argc;i++){
        if(!strcmp(argv[i],"--input") && i+1<argc) in_path
= argv[++i];
        else if(!strcmp(argv[i],"--output") && i+1<argc)
out_path = argv[++i];
        else if(!strcmp(argv[i],"--threads") && i+1<argc)
threads = atoi(argv[++i]);
    }
    if(!in_path || !out_path || threads < 1){
        fprintf(stderr,"Usage: %s --input
data/input_500.txt --output results/out.txt --threads
T\n", argv[0]);
        return 2;
    }
    FILE *f = fopen(in_path,"r");
    if(!f){ perror("fopen input"); return 1; }
    int cap = 1024, n = 0;
    int *a = (int*)malloc((size_t)cap*sizeof(int));
    while(1){
        int x;
        if(fscanf(f,"%d",&x) != 1) break;
        if(n >= cap){
            cap *= 2;
            a = (int*)realloc(a, (size_t)cap*sizeof(int));
        }
        a[n++] = x;
    }
    fclose(f);
    if(threads > n) threads = n;
    pthread_t *th =
(pthread_t*)malloc((size_t)threads*sizeof(pthread_t));
    task_t *task =
(task_t*)malloc((size_t)threads*sizeof(task_t));
    double t0 = now_s();
    for(int t=0;t<threads;t++){
        task[t].a = a;
        task[t].l = (t*n)/threads;
        task[t].r = ((t+1)*n)/threads;
        pthread_create(&th[t], NULL, worker, &task[t]);
    }
    for(int t=0;t<threads;t++) pthread_join(th[t], NULL);
    merge_k(a, n, threads);
    double t1 = now_s();
    FILE *g = fopen(out_path,"w");
    if(!g){ perror("fopen output"); return 1; }
    for(int i=0;i<n;i++){
        fprintf(g, "%d%s", a[i], (i+1<n) ? " " : "\n");
    }
    fclose(g);
    printf("pthreads_sort: n=%d threads=%d
time_s=%.6f\n", n, threads, (t1-t0));
    free(a); free(th); free(task);
    return 0;
}
EOF
```

This method requires more complex and error-prone code compared to OpenMP, but it provides deeper insight into low-level parallel execution and thread management mechanisms.

3. Experimental Results and Performance Analysis

This section presents a comprehensive performance evaluation of the implemented parallel sorting algorithms. The analysis covers **runtime**, **speedup**, and **parallel efficiency** for both shared-memory and distributed-memory paradigms. All experiments were conducted using a fixed input size of **N = 500 integers**, enabling a fair comparison between implementations.

3.1 Best Runtime Comparison Across Implementations

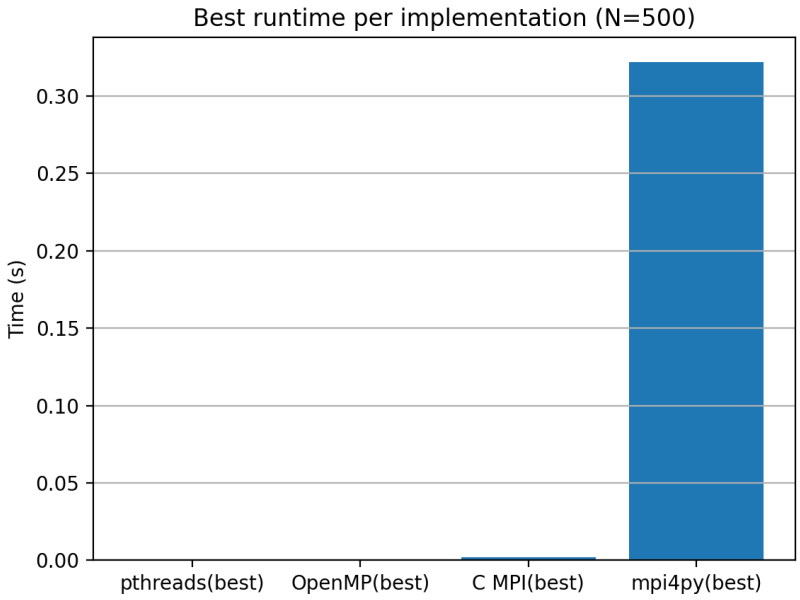

Figure 1

Figure 1 compares the **best (minimum) runtime** obtained for each parallel programming model under its optimal configuration.

The results clearly show that:

- **C + OpenMP** and **C + pthreads** achieve the lowest runtimes due to minimal communication overhead and efficient shared-memory access.
- **C + MPI** performs slightly slower than shared-memory approaches but remains significantly faster than Python-based MPI.
- **Python + mpi4py** exhibits the highest runtime, primarily due to interpretation overhead, object serialization, and higher communication costs.

This comparison highlights that for small problem sizes, **shared-memory parallelism is the most effective**, while Python-based distributed approaches are disadvantaged.

3.2 Shared Memory Performance: pthreads vs OpenMP

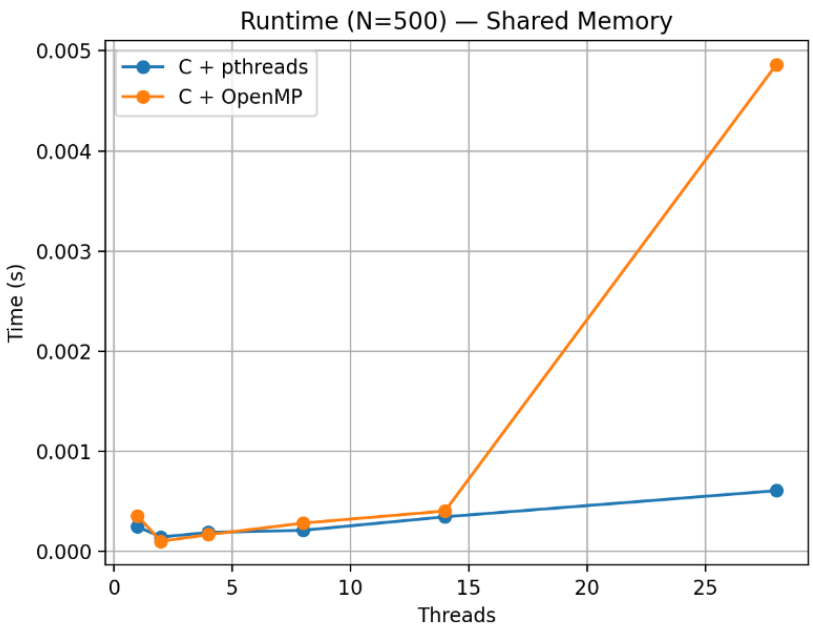

Figure 2

Figure 2 illustrates the runtime behavior of pthreads and OpenMP as the number of threads increases. Both implementations demonstrate optimal performance at **low thread counts (2–4 threads)**.

Notably:

- OpenMP achieves its minimum runtime at **2 threads**, outperforming pthreads.
- Beyond 8 threads, runtime increases rapidly, especially for OpenMP at 28 threads.

This trend indicates that thread creation, synchronization, and scheduling overhead dominate computation when the workload per thread becomes too small.

3.2.2 Speedup Analysis

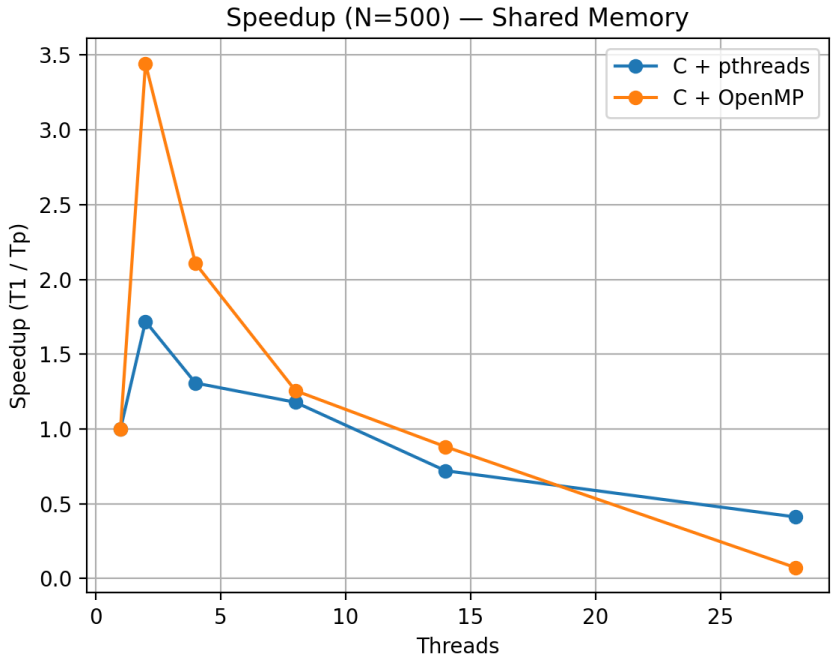

Figure 3.

The speedup curves show that:

- OpenMP reaches a maximum speedup of approximately **3.4× at 2 threads**.
- pthreads achieves a lower peak speedup of about **1.7×**.

After the peak, speedup decreases sharply and falls below 1 at high thread counts, confirming that excessive parallelism is counterproductive for small inputs.

3.2.3 Efficiency Analysis

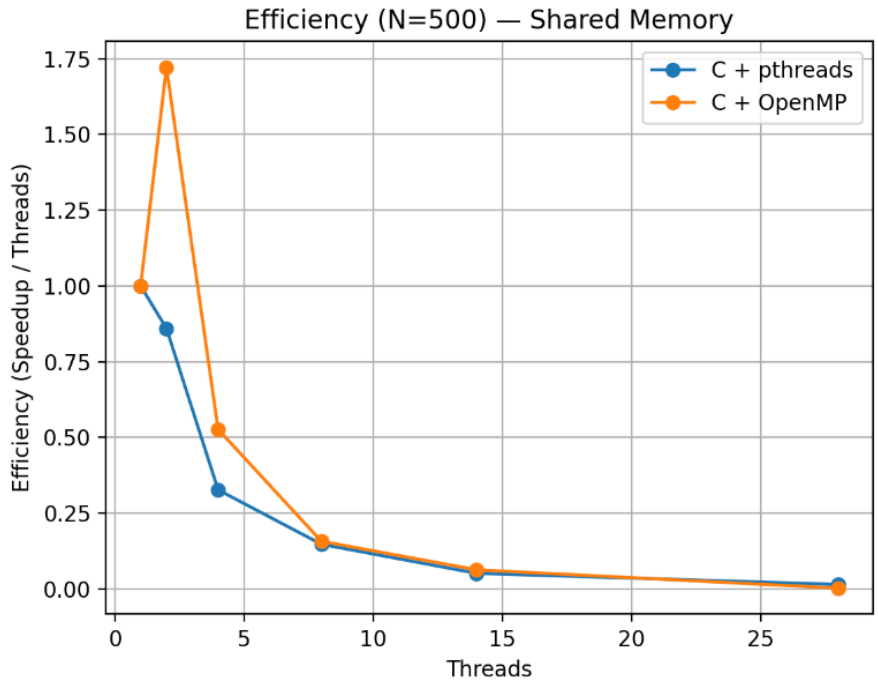

Figure 4

Parallel efficiency further confirms the observed behavior:

- Efficiency drops rapidly as the number of threads increases.
- At 14 and 28 threads, efficiency falls below **10%**, indicating severe resource underutilization.

Overall, OpenMP demonstrates superior performance at low thread counts, while both shared-memory approaches suffer from diminishing returns at higher levels of parallelism.

3.3 Distributed Memory Performance: C + MPI vs Python + mpi4py

3.3.1 Runtime vs MPI Ranks

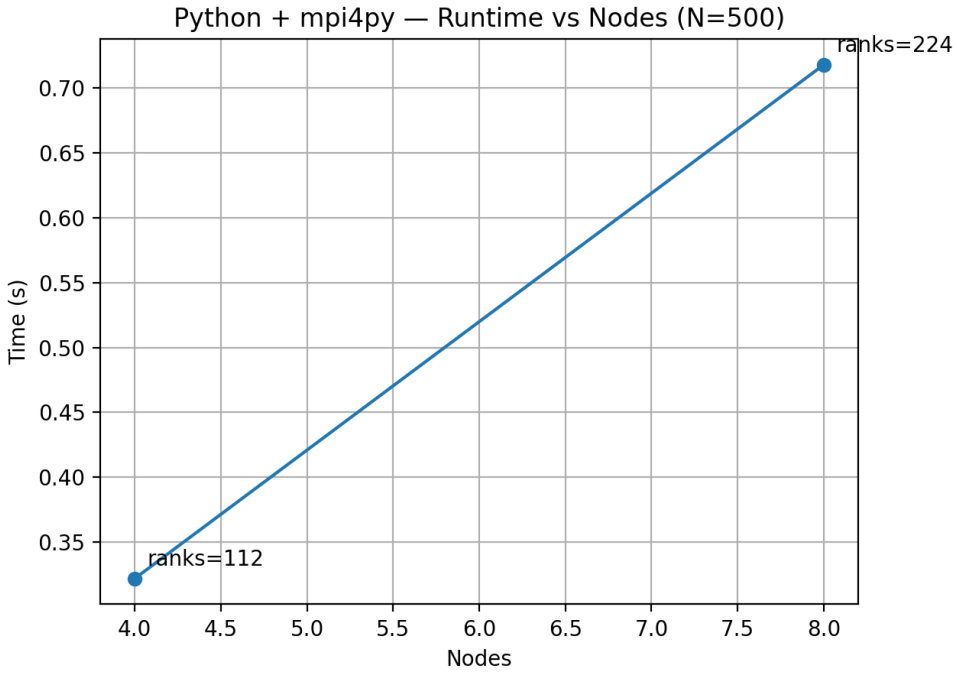

Figure 5

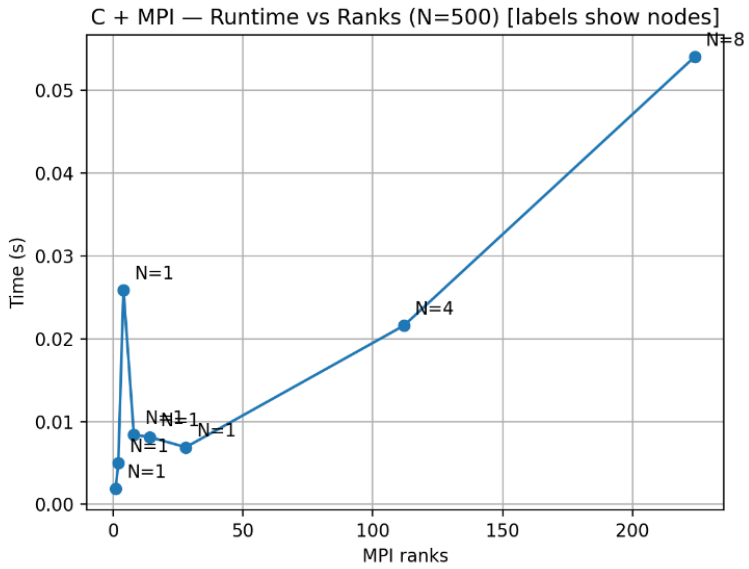

Figure 6

Figures 5 and 6 compare runtime scaling with respect to the number of MPI ranks. The difference between the two implementations is substantial:

- **C + MPI** achieves runtimes below **0.06 seconds**, even at 224 ranks.
- **Python + mpi4py** exceeds **0.7 seconds** under the same configuration.

The increasing runtime with more ranks indicates that **communication overhead dominates computation** when the problem size is small.

3.3.2 Runtime vs Nodes

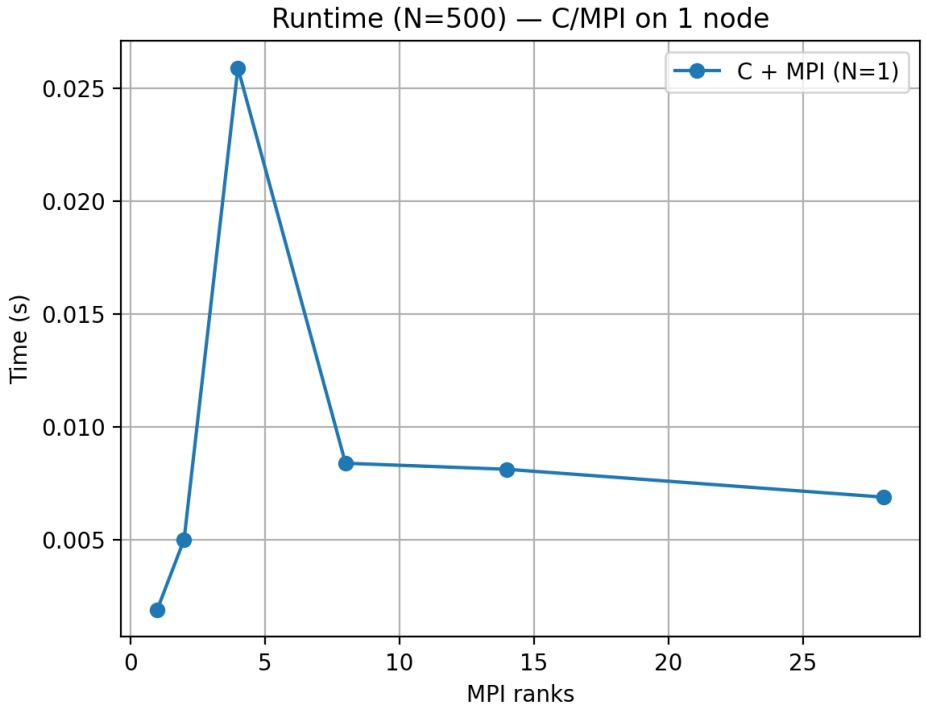

Figure 7

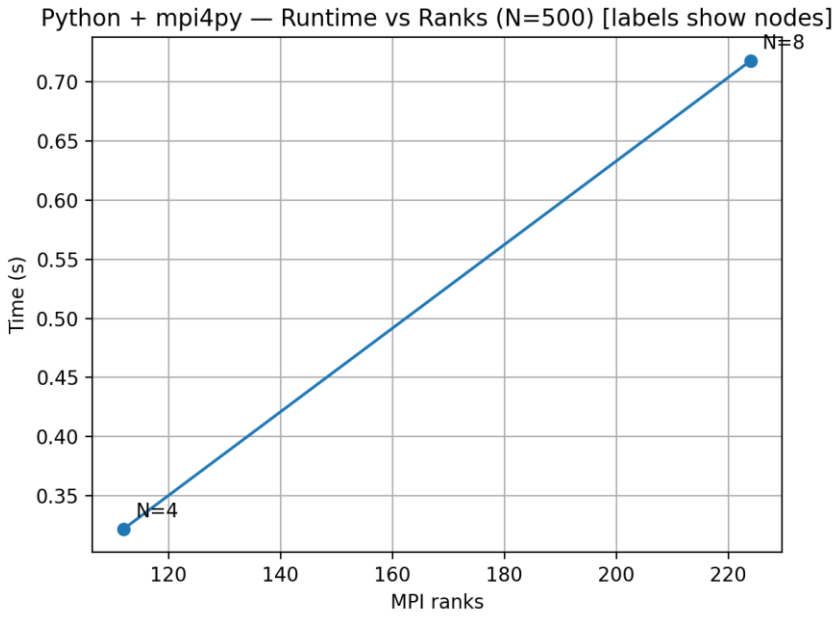

Figure 8

As the number of nodes increases from 4 to 8:

- C + MPI runtime increases moderately from approximately **0.021 s to 0.054 s**.
- Python + mpi4py runtime increases more sharply, from **0.32 s to over 0.71 s**.

This behavior reflects the cost of inter-node communication and highlights the inefficiency of scaling small workloads across many nodes.

Conclusion

This work compared several parallel sorting implementations using **C + pthreads**, **C + OpenMP**, **C + MPI**, and **Python + mpi4py** with a fixed input size of **N = 500**. The results show that for small datasets, **shared-memory approaches** are the most efficient. In particular, **C + OpenMP** achieved the best performance, providing the lowest runtime and highest speedup with a small number of threads due to low overhead and efficient use of shared memory. The **pthreads** implementation also performed well but required more complex thread management and showed limited scalability.

In contrast, **distributed-memory implementations** using MPI exhibited higher overhead. While **C + MPI** was significantly faster than **Python + mpi4py**, increasing the number of processes and nodes led to longer runtimes because communication costs dominated computation. This effect was especially strong in Python due to interpreter and data-handling overheads.

Overall, the experiments confirm that parallel efficiency strongly depends on problem size and execution model. For small workloads, shared-memory parallelism is preferable, while MPI-based solutions are more suitable for larger-scale problems.