Project 3 Report

COMPILATION AND EXECUTION (IN CLUSTER)

CMake Configuration: First, configure the project using CMake to generate the necessary build files. Navigate to the project directory (../project1/build/)and run the following command to create the build files:

bash **cmake ..**

This step sets up the project's build environment and prepares it for compilation.

Compilation: Next, use the **make** command to compile the project. You can specify the number of parallel jobs to speed up compilation (e.g., **-j4** for four parallel jobs):

bash **make -j4**

This command compiles the source code, linking all necessary libraries and generating the executable binary.

Cluster Submission Script:
To run our program on the cluster, use a submission script. In this example, use the **sbatch** command to submit a job to the cluster. It specifies job parameters such as the number of CPU cores, memory requirements, and the name of the executable to run.

shell **sh src/scripts/sbach.sh & sh src/scripts/sbach_bonus.sh**

The submission script is responsible for launching our program on the cluster nodes with the specified configurations.

Sorting Algorithms

Task 1: Process-Level Parallel Quick Sort with MPI:

- **Parallelization with MPI:**
  - The program utilizes MPI to parallelize the sorting process across multiple processes.
  - Each process works on a segment of the input vector, performing its own quicksort locally.

- **Key Functions:**
  - **partition**: Partitions a vector into two halves based on a chosen pivot.
  - **sequentialquickSort**: Implements the sequential quicksort algorithm recursively.
  - **quickSort**: Distributes the vector among MPI processes, performs parallel quicksort, and merges the sorted segments back on the master process.

The parallelization is achieved by dividing the vector among MPI processes, each responsible for sorting its assigned segment. The final sorted result is obtained by merging these segments.

Task 2: Process-Level Parallel Bucket Sort:

- **Parallelization with MPI:**
  - Each process performs bucket sort on its assigned data.
  - The sorted buckets are gathered and merged on the master process.

- **Bucket Sort Function:**
  - **bucketSort**: Distributes elements into local buckets based on a calculated range, sorts each local bucket using insertion sort, gathers information about bucket sizes on the master node, and finally gathers the sorted buckets to reconstruct the sorted vector.
  - Bucket sort uses insertion sort for each bucket due to its efficiency for small datasets, adaptability, simplicity, stability, and suitability for incremental data, like streaming data arrival.

The parallelization is achieved by dividing the vector into buckets, sorting each bucket independently, and then gathering the sorted buckets to reconstruct the final sorted vector.

Task 3: Process-Level Parallel Odd-Even Sort:

- **Odd-Even Sorting Process:**
  - Each process sorts its local array using sequential odd-even sort.
  - Odd and even phases are conducted using **oddEvenIteration**, which involves communication and merging with neighboring processes.

- **Odd-Even Sort Functions:**
  - **oddEvenSort:** The main function that orchestrates the parallel odd-even sort. It initializes MPI, generates a random vector, and then performs odd-even sorting using MPI communication functions.
  - **oddEvenIteration:** One iteration of the parallel odd-even transposition sort. It manages the exchange and merging of elements between neighboring processes.
  - **mergeSplitLow and mergeSplitHigh:** Functions to merge the smallest and largest elements, respectively, from local arrays and temporary buffers.
  - **sequentialOddEvenSort:** Implements the odd-even sort algorithm on a local array in a sequential manner.

The parallelization is achieved by dividing the work, independently sorting local vectors in parallel, exchanging information and elements between processes, and finally, gathering the sorted results for broadcasting. The use of MPI facilitates efficient communication and coordination among the distributed processes.

Task 5: Dynamic Thread-Level Parallel Quick Sort (Method-2: OpenMP Tasking):

- **OpenMP Parallelization:**
  - The **#pragma omp parallel** directive creates a parallel region, and **num_threads(thread_num)** sets the number of threads.
  - **#pragma omp single nowait** ensures that only one thread executes the following block. It's a task creation construct.
  - Inside the single block, the parallel tasks are spawned with **#pragma omp task** for the left and right subarrays. **shared(vec)** indicates that the tasks share the same data.
  - **#pragma omp taskwait** ensures synchronization, waiting for the completion of spawned tasks before proceeding.
  - If the array size is below the threshold, the code falls back to sequential quicksort without creating additional tasks.

This code implements parallel quicksort using OpenMP (OMP) to take advantage of multicore processors. Quick sort is a sorting algorithm that recursively divides the input array into subarrays, sorts them independently, and then combines them to achieve a sorted array. The parallelization is applied using OpenMP directives to enable concurrent execution of tasks. The overall parallelization aims to improve the performance by utilizing multiple threads for sorting subarrays concurrently, reducing the overall execution time.

Experiment Results

Workers	QuickSort (MPI)	BucketSort (MPI)	Odd-Even-Sort (MPI)	QuickSort (Threads)
1	14165	19170	37122	13358
2	11573	8305	14017	11145
4	7860	5356	3513	7853
8	6569	3059	875	3950
16	8692	2090	222	2399
32	12503	1697	66	1745

EXPERIMENT RESULTS

Task 1: Process-Level Parallel Quick Sort with MPI:
Quick Sort Sequential (Optimized with -O2)
Performance counter stats for 'sh src/sbach.sh':

```
    16,463,284      instructions:u           #    1.20  insn per cycle          (75.31%)
    13,705,813      cpu-cycles:u                                                (74.07%)
       162,706      cache-references:u                                          (71.46%)
        90,534      cache-misses:u           #   55.643 % of all cache refs     (70.73%)
     4,316,766      branch-instructions:u                                       (74.71%)
       100,699      branch-misses:u          #    2.33% of all branches         (59.19%)
       330,508      bus-cycles:u                                                (61.38%)

  29.139886185 seconds time elapsed

   0.017686000 seconds user
   0.022430000 seconds sys
```

Quick Sort MPI (Optimized with -O2) num_cores = 32
Performance counter stats for 'sh src/sbach.sh':

```
    18,429,411      instructions:u           #    1.23  insn per cycle          (72.48%)
    14,974,699      cpu-cycles:u                                                (69.62%)
       164,902      cache-references:u                                          (71.68%)
        86,991      cache-misses:u           #   52.753 % of all cache refs     (75.01%)
     4,394,573      branch-instructions:u                                       (76.04%)
       117,159      branch-misses:u          #    2.67% of all branches         (63.89%)
       390,305      bus-cycles:u                                                (58.53%)

  34.489679325 seconds time elapsed

   0.020338000 seconds user
   0.025181000 seconds sys
```

Sequential Quick Sort: Efficient for a single core, with relatively higher cache misses.
Quick Sort MPI (32 cores): Shows potential benefits in cache misses but has more bus cycles, indicating potential communication overhead.
Not as scalable for 32 cores.

Task 2: Process-Level Parallel Bucket Sort:
Bucket Sort Sequential (Optimized with -O2)
Performance counter stats for 'sh src/sbach.sh':

```
     17,064,624      instructions:u           #    1.18  insn per cycle          (72.58%)
     14,447,491      cpu-cycles:u                                          (70.54%)
        177,046      cache-references:u                                      (74.58%)
         76,129      cache-misses:u           #   43.000 % of all cache refs      (76.00%)
      4,326,095      branch-instructions:u                                    (76.18%)
        104,212      branch-misses:u          #    2.41% of all branches         (57.90%)
        323,825      bus-cycles:u                                          (56.69%)

   34.990662257 seconds time elapsed

    0.018944000 seconds user
    0.021360000 seconds sys
```

Bucket Sort MPI (Optimized with -O2) num_cores = 32
Performance counter stats for 'sh src/sbach.sh':

```
     17,806,143      instructions:u           #    1.13  insn per cycle          (74.63%)
     15,789,763      cpu-cycles:u                                          (76.77%)
        144,176      cache-references:u                                      (68.26%)
         68,721      cache-misses:u           #   47.665 % of all cache refs      (70.36%)
      4,224,531      branch-instructions:u                                    (78.17%)
        122,736      branch-misses:u          #    2.91% of all branches         (61.15%)
        402,536      bus-cycles:u                                          (58.23%)

   24.149516623 seconds time elapsed

    0.014235000 seconds user
    0.031545000 seconds sys
```

Sequential Bucket Sort: Shows good instruction efficiency and lower cache misses.
Bucket Sort MPI (32 cores): Slightly lower instruction efficiency and increased cache misses, indicating potential communication overhead.

Task 3: Process-Level Parallel Odd-Even Sort:
Odd-Even Sort Sequential (Optimized with -O2)
Performance counter stats for 'sh src/sbach.sh':

```
      17,942,596      instructions:u           #    1.35  insn per cycle         (73.42%)
      13,263,993      cpu-cycles:u                                              (74.68%)
         169,695      cache-references:u                                        (75.82%)
          97,254      cache-misses:u           #   57.311 % of all cache refs    (73.72%)
       4,859,235      branch-instructions:u                                     (69.96%)
         112,724      branch-misses:u          #    2.32% of all branches        (58.16%)
         375,657      bus-cycles:u                                              (57.89%)

    41.668702996 seconds time elapsed

     0.020179000 seconds user
     0.022691000 seconds sys
```

Odd-Even Sort MPI (Optimized with -O2) num_cores = 32
Performance counter stats for 'sh src/sbach.sh':

```
      19,255,500      instructions:u           #    1.25  insn per cycle         (75.08%)
      15,401,702      cpu-cycles:u                                              (71.09%)
         177,601      cache-references:u                                        (71.38%)
          97,900      cache-misses:u           #   55.124 % of all cache refs    (71.13%)
       4,773,301      branch-instructions:u                                     (73.01%)
         117,169      branch-misses:u          #    2.45% of all branches        (64.20%)
         407,719      bus-cycles:u                                              (63.19%)

     0.540197784 seconds time elapsed

     0.024356000 seconds user
     0.023865000 seconds sys
```

Sequential Odd-Even Sort: Shows good instruction efficiency and moderate cache misses.
Odd-Even Sort MPI (32 cores): Comparable cache misses, but slightly lower instruction efficiency compared to the sequential version.

Task 5: Dynamic Thread-Level Parallel Quick Sort (Method-2: OpenMP Tasking):
Performance counter stats for 'sh src/sbach_bonus.sh':

```
    18,450,005      instructions:u           #    1.35  insn per cycle          (72.64%)
    13,655,867      cpu-cycles:u                                            (71.48%)
       142,366      cache-references:u                                        (72.24%)
        81,223      cache-misses:u           #   57.052 % of all cache refs      (69.48%)
     4,514,661      branch-instructions:u                                       (73.42%)
       111,419      branch-misses:u          #    2.47% of all branches          (63.13%)
       375,828      bus-cycles:u                                            (58.62%)

   17.956732238 seconds time elapsed

    0.019261000 seconds user
    0.022908000 seconds sys
```

The OpenMP tasking version shows performance comparable to the sequential version. Good instruction efficiency and cache performance indicate effective parallelization with OpenMP.