Parallel Merging of Two Sorted Arrays

Reid Harrington: 169041378 harr1378@mylaurier.ca

Muhammad Hamza Adnan: 169023134 adna3134@mylaurier.ca

Tanner Bell: 200932300 bell2300@mylaurier.ca

Vikas Movva: 190957230 movv7230@mylaurier.ca

Mason Barney: 169035310 barn5310@mylaurier.ca

March 10, 2025

Contributions

Reid Harrington: Primary student responsible for writing code
Muhammad Hamza Adnan: Secondary student responsible for writing algorithm as well as latex encoding
Tanner Bell: Analysis of code and results and writing of the report document
Mason Barney: Secondary writer for the report and analysis of code
Vikas Movva: Tertiary writer for the report and analysis of overall program objective

Our Program

The program we created utilizes parallel programming and Message Passing Interface (MPI) to merge two sorted arrays in parallel. In summary, individual processes are assigned partitions of the two arrays, merge them independently, and then all combine together to form the final merged array.

Code

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Global variables
int *A; // A array
int *B; // B array
int n = 100; // Size of each array

int binary_search(int array[], int size, int target);

// Function for qsort, requires const void * as parameters
int compare(const void *a, const void *b)
{
    return (*(int*)a - *(int*)b);
}

void random_array(int *a, int seed)
{
    srand(seed);
    for (int i = 0; i < n; i++)
    {
        a[i] = rand() % 600; // Random number between 1 and 600
    }
    qsort(a, n, sizeof(int), compare);
}

void merge(int *A, int sizeA, int *B, int sizeB, int *C)
{
    // Indexes for A, B, and C arrays
    int i = 0, j = 0, k = 0;
    while(i < sizeA || j < sizeB) // While there are still elements left in either A or B
    {
        // Copy element from A
        if (j >= sizeB || (i < sizeA && A[i] <= B[j])) // If no elements left in B, or if we still have elements in A and the current element is less than the current element in B
        {
            C[k++] = A[i++];
        }
        // Copy element from B
        else
        {
            C[k++] = B[j++];
        }
    }
}

int main(int argc, char *argv[])
{
    // Step 1: Initiating MPI
    MPI_Init(&argc, &argv);

    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
57
58      // Allocate memory for our arrays
59      A = (int *)malloc(n * sizeof(int));
60      B = (int *)malloc(n * sizeof(int));
61      if(world_rank == 0)
62      {
63          int seedA = time(NULL);
64          int seedB = seedA + 17;
65          // Generate random numbers for arrays
66          random_array(A, seedA);
67          random_array(B, seedB);
68          printf("A array:\n");
69          for(int i = 0; i < n; i++)
70          {
71              printf("%d ", A[i]);
72          }
73          printf("\nB array:\n");
74          for(int i = 0; i < n; i++)
75          {
76              printf("%d ", B[i]);
77          }
78      }
79      MPI_Bcast(A, n, MPI_INT, 0, MPI_COMM_WORLD);
80      MPI_Bcast(B, n, MPI_INT, 0, MPI_COMM_WORLD);
81
82      int base = n / world_size; // Number of elements each process gets
83      int rem = n % world_size; // Remainder if n isn't divisible
84
85      // Step 2: Split points
86      int *split_points = NULL;
87      // Rank 0 processes the split points for B
88      if(world_rank == 0)
89      {
90          // Creating the split points array, of size processors + 1
91          split_points = (int *)malloc((world_size + 1) * sizeof(int));
92          // First split point at 0
93          split_points[0] = 0;
94
95          int base = n / world_size;
96          int current = 0;
97          // Find all split points based on A partition
98          for (int i = 0; i < world_size; i++)
99          {
100             int size_A = base + (i < rem ? 1 : 0);
101             int a_last = current + size_A - 1;
102             current += size_A;
103             split_points[i + 1] = binary_search(B, n, A[a_last]);
104         }
105         split_points[world_size] = n;
106     }
107     // Broadcast split points to all processes
108     else if (world_rank != 0)
109     {
110         split_points = (int *)malloc((world_size + 1) * sizeof(int));
111     }
112     MPI_Bcast(split_points, world_size + 1, MPI_INT, 0, MPI_COMM_WORLD);
113
114     // Step 3: Local partitioning
115     int a_start = world_rank * base + (world_rank < rem ? world_rank : rem); // Calculating
        red↪ correct partitioning if there's a remainder
116     int size_A = base + (world_rank < rem ? 1 : 0); // Calculating correct partitioning if
        red↪ there's a remainder
117
118     int b_start = split_points[world_rank];
119     int size_B = split_points[world_rank + 1] - b_start;
120
121     // Allocating memory for the C array, which holds our final merged result
122     int *local_C = (int *) malloc((n * 2) * sizeof(int));
```

```
123     int size_C = size_A + size_B;
124
125     // Step 4: Merging
126     merge(&A[a_start], size_A, &B[b_start], size_B, local_C);
127
128     // Step 5: Gathering results of locally merged arrays at process 0
129     int *C = NULL;
130     int *recv_counts = NULL;
131     int *displs = NULL;
132     // Process 0 prepares to gather data
133     if (world_rank == 0)
134     {
135         C = (int *)malloc(2 * n * sizeof(int));
136         recv_counts = (int *)malloc(world_size * sizeof(int));
137         displs = (int *)malloc(world_size * sizeof(int));
138     }
139
140     MPI_Gather(&size_C, 1, MPI_INT, recv_counts, 1, MPI_INT, 0, MPI_COMM_WORLD);
141
142     if (world_rank == 0)
143     {
144         displs[0] = 0;
145         for (int i = 1; i < world_size; i++)
146         {
147             displs[i] = displs[i - 1] + recv_counts[i - 1];
148         }
149     }
150     // Gather all parts into C
151     MPI_Gatherv(local_C, size_C, MPI_INT, C, recv_counts, displs, MPI_INT, 0, MPI_COMM_WORLD
        red↪ );
152
153     // Root prints the merged array
154     if (world_rank == 0)
155     {
156         printf("\nMerged array:\n");
157         for (int i = 0; i < 2 * n; i++) {
158             printf("%d ", C[i]);
159         }
160         printf("\n");
161         free(C);
162         free(recv_counts);
163         free(displs);
164     }
165
166     MPI_Finalize();
167     return 0;
168 }
169
170 // Helper method for binary search, finds lower bound
171 int binary_search(int array[], int size, int target)
172 {
173     int left = 0;
174     int right = size;
175
176     while (left < right)
177     {
178         int middle = left + (right - left) / 2;
179
180         if (array[middle] <= target) left = middle + 1;
181
182         else right = middle;
183     }
184
185     // Target not found
186     return left;
187 }
```

Full Program Description

Our program efficiently merges two sorted arrays A and B by first initializing MPI, initializing the number of processes, and determining the rank of each process. Then we partition each array between processes; each partition is set to manage a distinct segment. We set Rank 0 to compute split points in B using binary search, which ensures a balanced workload and no overlap among each process. These split points are broadcasted to all processes. Each process utilizes these split points to define their corresponding partition of A. The next step is for each process to perform a local merge of their portion of A and B, and store the resulting temporary sequential array. After every process has performed their local merge, Rank 0 brings together all the merged subarrays using `MPI_Gatherv`. Rank 0 finally assembles the final sorted array and the final result is printed. Overall, this process reduces time complexity and significantly increases efficiency by utilizing parallel programming and MPI to divide the work among processors.

Step-by-Step Explanation

Step 1: Initialization

- **MPI Initialization**:
  - MPI protocol is initialized using `MPI_Init`. This creates the parallel environment and facilitates communication between processes.
  - `MPI_Comm_size` determines the total number of processes in the program (`world_size`).
  - `MPI_Comm_rank` assigns a unique rank identifier (ID) to each process (`world_rank`).
- **Array Initialization**:
  - Two global arrays, `Array A` and `Array B`, each has a size of 100.
  - Process 0 (rank 0) creates random numbers for `Array A` and `Array B`, sorts them using the quicksort algorithm helper method `qsort`, and prints them to the console.
  - The arrays are then broadcast to each process using the `MPI_Bcast` method, ensuring that each individual process has full access to both arrays.

Step 2: Splitting the Arrays

- **Partitioning the Arrays**:
  - The arrays are divided evenly among the processes. Each process gets a piece of `Array A` and a proportional piece of `Array B` to merge.
  - The number of elements each process gets is calculated as:

```
int base = n / world_size; // number of elements each process gets
int rem = n % world_size; // remainder if n isn't divisible
```

  - Processes with ranks less than `rem` get an extra element to take care of the remainder amount.
- **Binary Search for Split Points**:
  - Process 0 searches for split points in `Array B` using binary search. These split points ensure that each process merges non-overlapping sections of `Array A` and `Array B`.
  - The calculation for split points is based on the last element of each process's portion of `Array A` as follows:

```
split_points[i + 1] = binary_search(B, n, A[a_last]);
```

  - The split points are stored in an array and distributed to all processes using the `MPI_Bcast` method.

Step 3: Local Partitioning

- **Defining Local Portions**:
  - Each process calculates its start index and size for its part of `Array A` and `Array B`:

```
int a_start = world_rank * base + (world_rank < rem ? world_rank : rem);
int size_A = base + (world_rank < rem ? 1 : 0);
int b_start = split_points[world_rank];
int size_B = split_points[world_rank + 1] - b_start;
```

  - `a_start` and `size_A` define the part of `Array A` for the current process.
  - `b_start` and `size_B` define the part of `Array B` for the current process.

Step 4: Local Merging

- **Merging Local Portions**:
  - Each process merges its part of `Array A` and `Array B` into a temporary array `local_C` using the `merge` method.
  - The `merge` method works as follows:
    - Iterate through `Array A` and `Array B` using indices `i` and `j`.
    - Compare elements from `Array A` and `Array B` and copy the smaller element into `local_C`.

Step 5: Gather Results

- **Preparing to Gather the Data**:
  - Process 0 allocates memory for the final merged array `C` and arrays to store the counts and displacements for `MPI_Gatherv`.
- **Gathering Local Results**:
  - Each process sends the total size of its merged array (`size_C`) to process 0 using the `MPI_Gather` method.
  - Process 0 calculates the displacements for each process's parts in the final array `C`.
- **Combining Results**:
  - Process 0 gathers all `local_C` from each process using `MPI_Gatherv` and combines them into the final product, array `C`.
- **Printing the Final Merged Array**:
  - Process 0 prints the final merged array, array `C`, to the console.

Step 6: Cleanup

- **Freeing Memory**:
  - Process 0 frees all memory allocated to `C`, `recv_counts`, and `displs`.
- **Finalizing MPI**:
  - Each process calls `MPI_Finalize` to clean up the environment.

Results

For $N = 100$

A Array:

```
3 7 11 19 21 28 30 35 39 53 65 90 95 117 127 128 131 134 135 139 140
141 142 144 151 155 157 160 162 163 166 168 178 188 190 194 201 201
201 205 206 223 224 231 231 241 251 252 255 256 261 266 271 285 288
288 300 304 306 317 323 332 336 344 347 364 365 370 381 394 397 407
428 429 430 434 437 447 450 451 466 467 477 490 510 511 511 522 527
547 553 555 558 563 569 583 584 587 588 598
```

B Array:

```
6 9 12 13 13 27 32 42 57 66 68 69 72 91 102 103 116 128 145 150 170
171 172 180 186 186 190 197 225 226 240 245 250 250 260 266 285 290
294 295 295 297 299 302 308 318 327 333 334 336 337 344 346 348 355
356 359 361 371 374 386 405 406 417 418 420 423 425 425 427 437 447
448 452 473 475 475 476 481 481 495 495 498 500 502 505 509 511 516
525 539 549 554 578 578 580 589 591 596 598
```

Merged Array:

```
3 6 7 9 11 12 13 13 19 21 27 28 30 32 35 39 42 53 57 65 66 68 69 72 90
91 95 102 103 116 117 127 128 128 131 134 135 139 140 141 142 144 145
150 151 155 157 160 162 163 166 168 170 171 172 178 180 186 186 188
190 190 194 197 201 201 201 205 206 223 224 225 226 231 231 240 241
245 250 250 251 252 255 256 260 261 266 266 271 285 285 288 288 290
294 295 295 297 299 300 302 304 306 308 317 318 323 327 332 333 334
336 336 337 344 344 346 347 348 355 356 359 361 364 365 370 371 374
381 386 394 397 405 406 407 417 418 420 423 425 425 427 428 429 430
434 437 437 447 447 448 450 451 452 466 467 473 475 475 476 477 481
481 490 495 495 498 500 502 505 509 510 511 511 511 516 522 525 527
539 547 549 553 554 555 558 563 569 578 578 580 583 584 587 588 589
591 596 598 598
```

For $N = 200$

A Array:

```
2 3 3 3 5 7 7 7 13 22 22 27 29 30 36 37 39 42 43 49 54 54 58 63 63 68
72 81 83 84 85 86 86 88 92 93 100 101 105 108 109 109 112 117 118 119
120 122 123 123 125 126 130 130 131 138 140 145 145 154 155 156 156
159 159 165 172 172 177 179 187 189 191 191 201 205 210 213 217 218
235 236 241 248 249 249 256 257 258 270 272 273 277 277 283 284 288
294 296 297 305 308 308 314 315 315 327 332 333 338 339 343 350 351
358 358 359 363 364 366 366 371 374 379 383 384 387 389 390 395 396
397 397 401 402 402 404 408 409 415 416 419 420 423 433 435 443 443
445 447 449 456 460 461 466 469 480 487 490 493 495 498 501 501 507
507 511 512 512 512 513 515 519 520 520 529 530 531 532 537 547 548
549 550 551 559 559 568 569 572 573 573 575 575 579 581 582 583 592
593
```

B Array:

```
0 3 3 11 13 15 22 25 25 30 34 38 42 44 45 47 47 56 60 62 65 67 70 70
72 72 81 82 85 88 93 103 107 107 109 111 114 115 115 120 121 122 122
124 126 131 132 140 140 142 142 142 150 154 159 162 162 169 170 182
184 185 186 187 187 190 193 196 200 201 201 203 203 210 210 211 212
215 218 218 221 224 225 228 228 233 235 235 236 237 242 255 256 257
261 265 269 270 274 275 275 295 295 300 301 303 316 320 322 326 326
331 331 332 336 339 342 348 352 353 363 368 370 387 395 395 398 398
401 401 402 404 404 405 410 410 420 425 431 435 440 447 449 455 455
456 457 467 468 468 471 471 474 475 475 476 478 478 484 485 488 489
492 492 494 500 502 503 504 509 511 512 513 519 531 535 543 549 553
561 564 566 570 572 572 573 575 576 577 577 577 583 583 584 585 587
587 589 594 595
```

Merged Array:

```
0 2 3 3 3 3 3 5 7 7 7 11 13 13 15 22 22 22 25 25 27 29 30 30 34 36 37
38 39 42 42 43 44 45 47 47 49 54 54 56 58 60 62 63 63 65 67 68 70 70
72 72 72 81 81 82 83 84 85 85 86 86 88 88 92 93 93 100 101 103 105 107
107 108 109 109 109 111 112 114 115 115 117 118 119 120 120 121 122
122 122 123 123 124 125 126 126 130 130 131 131 132 138 140 140 140
142 142 142 145 145 150 154 154 155 156 156 159 159 159 162 162 165
169 170 172 172 177 179 182 184 185 186 187 187 187 189 190 191 191
193 196 200 201 201 201 203 203 205 210 210 210 211 212 213 215 217
218 218 218 221 224 225 228 228 233 235 235 235 236 236 237 241 242
248 249 249 255 256 256 257 257 258 261 265 269 270 270 272 273 274
275 275 277 277 283 284 288 294 295 295 296 297 300 301 303 305 308
308 314 315 315 316 320 322 326 326 327 331 331 332 332 333 336 338
339 339 342 343 348 350 351 352 353 358 358 359 363 363 364 366 366
368 370 371 374 379 383 384 387 387 389 390 395 395 395 396 397 397
398 398 401 401 401 402 402 402 404 404 404 405 408 409 410 410 415
416 419 420 420 423 425 431 433 435 435 440 443 443 445 447 447 449
449 455 455 456 456 457 460 461 466 467 468 468 469 471 471 474 475
475 476 478 478 480 484 485 487 488 489 490 492 492 493 494 495 498
500 501 501 502 503 504 507 507 509 511 511 512 512 512 512 513 513
515 519 519 520 520 529 530 531 531 532 535 537 543 547 548 549 549
550 551 553 559 559 561 564 566 568 569 570 572 572 572 573 573 573
575 575 575 576 577 577 577 579 581 582 583 583 583 584 585 587 587
589 592 593 594 595
```

For $N = 500$

A Array:

```
0 1 1 4 6 6 7 7 9 13 15 16 16 17 19 19 19 19 25 29 29 30 31 33 33 33
34 34 35 35 36 37 37 38 40 42 43 44 46 48 49 50 52 52 52 53 56 57 57
59 62 63 63 64 66 66 68 69 71 71 71 73 73 73 74 77 77 79 80 80 80 81
82 82 83 85 86 87 87 88 91 91 91 91 94 94 95 96 96 96 96 96 97 98 98
99 99 101 102 102 102 103 105 106 106 107 107 108 109 110 112 114 115
115 115 117 117 119 120 121 122 122 122 125 125 126 126 130 130 131
135 136 136 138 140 140 141 142 143 145 146 148 148 148 149 151 154
155 156 158 161 162 164 166 166 167 169 171 173 174 177 178 184 184
186 186 187 190 191 192 194 195 197 197 198 201 203 204 204 205 206
207 209 209 209 209 211 214 216 217 222 222 223 224 225 225 227 229
```

230 233 234 235 235 236 237 237 237 240 240 243 244 245 246 247 247
249 250 252 253 254 255 258 263 264 264 265 268 270 271 271 271 274
275 277 277 277 279 280 284 285 290 291 292 292 292 292 293 294 294
294 296 296 297 301 302 302 306 308 309 310 310 310 313 317 321 325
325 326 327 333 335 336 337 339 339 341 341 341 342 342 345 350 354
355 356 359 360 360 360 363 364 365 366 367 367 367 368 370 370 374
375 376 376 377 379 382 383 384 385 386 386 389 394 394 394 394 395
396 399 402 403 403 405 405 407 408 409 409 412 412 413 416 417 417
419 420 421 421 423 423 423 426 428 429 430 431 431 431 432 433 435
436 437 438 439 440 442 443 445 446 446 447 450 450 450 450 452 453
454 454 456 458 459 460 460 462 464 464 466 466 467 468 469 469 470
470 471 473 474 476 477 478 479 480 481 481 482 482 482 484 486 486
486 487 487 488 489 490 490 491 492 493 494 494 495 495 497 498 500
501 501 506 507 508 512 514 516 519 519 520 521 522 523 524 525 527
531 531 532 533 535 535 536 537 538 539 543 544 545 548 549 550 551
551 552 552 555 556 557 557 558 559 559 560 562 564 565 566 567 567
569 569 569 570 571 571 572 573 577 579 580 580 581 581 583 583 587
588 588 588 589 591 592 595 596 596 598 598 598 599

B Array:

3 4 8 10 11 13 13 13 15 18 20 21 23 25 25 26 26 26 30 31 32 33 33 34
34 35 36 38 38 42 45 48 49 49 49 50 50 50 52 55 56 57 57 58 58 59 59
59 59 60 61 62 62 65 65 65 66 66 69 69 70 73 74 74 77 77 78 79 80 81
83 88 88 89 89 89 90 92 96 96 96 97 97 97 98 100 104 104 104 105 107
107 107 108 111 111 113 113 113 114 114 116 117 117 119 123 123 124
124 124 125 126 127 127 129 130 130 130 133 135 137 141 142 142 143
144 145 146 147 148 149 150 151 152 153 153 154 155 155 157 157 162
163 165 165 166 166 166 168 169 171 171 176 177 180 183 184 184 185
186 188 189 190 193 194 199 200 204 205 205 210 212 215 216 216 220
221 221 222 223 223 223 223 224 225 225 226 226 226 226 228 229 229
229 229 231 232 232 233 233 233 235 237 237 238 238 238 239 243 243
243 244 246 247 248 248 253 253 254 255 255 261 261 263 264 266 268
268 269 269 269 271 272 272 275 276 278 278 281 281 281 282 283 284
285 285 286 287 287 291 291 294 296 303 305 305 307 314 314 315 316
317 319 320 320 320 321 321 328 329 330 331 331 333 336 336 340 340
340 341 342 342 342 344 344 345 346 346 347 347 348 350 351 352 352
356 358 361 365 367 369 370 371 371 374 375 377 378 380 380 383 384
384 386 390 390 392 393 394 397 400 400 403 404 405 408 408 409 410
410 410 411 411 413 413 413 416 416 417 418 420 421 422 424 427 428
429 430 432 433 435 436 436 437 437 438 438 438 440 441 442 442 443
444 445 447 450 451 453 454 454 454 455 457 457 459 459 460 462 464
464 466 466 467 468 475 475 475 477 477 478 478 478 479 479 481 488
488 489 494 494 494 495 498 498 498 498 500 500 500 501 502 502 504
505 505 506 506 508 508 510 510 512 513 513 514 516 519 519 523 523
524 524 525 527 527 528 531 531 534 534 534 539 540 542 546 546 548
549 551 551 552 555 557 557 557 557 558 558 558 558 559 559 560 561
562 562 563 564 565 565 566 568 569 569 570 571 572 572 574 574 575
578 578 579 579 580 580 581 581 581 581 584 584 589 590 590 591 593
598

Merged Array:

0 1 1 3 4 4 6 6 7 7 8 9 10 11 13 13 13 13 15 15 16 16 17 18 19 19 19
19 20 21 23 25 25 25 26 26 26 29 29 30 30 31 31 32 33 33 33 33 33 34

34 34 34 35 35 35 36 36 37 37 38 38 38 40 42 42 43 44 45 46 48 48 49
49 49 49 50 50 50 50 52 52 52 52 53 55 56 56 57 57 57 57 58 58 59 59
59 59 59 60 61 62 62 62 63 63 64 65 65 65 66 66 66 66 68 69 69 69 70
71 71 71 73 73 73 73 74 74 74 77 77 77 77 78 79 79 80 80 80 80 81 81
82 82 83 83 85 86 87 87 88 88 88 89 89 89 90 91 91 91 91 92 94 94 95
96 96 96 96 96 96 96 96 97 97 97 97 98 98 98 99 99 100 101 102 102 102
103 104 104 104 105 105 106 106 107 107 107 107 107 108 108 109 110
111 111 112 113 113 113 114 114 114 115 115 115 116 117 117 117 117
119 119 120 121 122 122 122 123 123 124 124 124 125 125 125 126 126
126 127 127 129 130 130 130 130 130 131 133 135 135 136 136 137 138
140 140 141 141 142 142 142 143 143 144 145 145 146 146 147 148 148
148 148 149 149 150 151 151 152 153 153 154 154 155 155 155 156 157
157 158 161 162 162 163 164 165 165 166 166 166 166 166 167 168 169
169 171 171 171 173 174 176 177 177 178 180 183 184 184 184 184 185
186 186 186 187 188 189 190 190 191 192 193 194 194 195 197 197 198
199 200 201 203 204 204 204 205 205 205 206 207 209 209 209 209 210
211 212 214 215 216 216 216 217 220 221 221 222 222 222 223 223 223
223 223 224 224 225 225 225 225 226 226 226 226 227 228 229 229 229
229 229 230 231 232 232 233 233 233 233 234 235 235 235 236 237 237
237 237 237 238 238 238 239 240 240 243 243 243 243 244 244 245 246
246 247 247 247 248 248 249 250 252 253 253 253 254 254 255 255 255
258 261 261 263 263 264 264 264 265 266 268 268 268 269 269 269 270
271 271 271 271 272 272 274 275 275 276 277 277 277 278 278 279 280
281 281 281 282 283 284 284 285 285 285 286 287 287 290 291 291 291
292 292 292 292 293 294 294 294 294 296 296 296 297 301 302 302 303
305 305 306 307 308 309 310 310 310 313 314 314 315 316 317 317 319
320 320 320 321 321 321 325 325 326 327 328 329 330 331 331 333 333
335 336 336 336 337 339 339 340 340 340 341 341 341 341 342 342 342
342 342 344 344 345 345 346 346 347 347 348 350 350 351 352 352 354
355 356 356 358 359 360 360 360 361 363 364 365 365 366 367 367 367
367 368 369 370 370 370 371 371 374 374 375 375 376 376 377 377 378
379 380 380 382 383 383 384 384 384 385 386 386 386 389 390 390 392
393 394 394 394 394 394 395 396 397 399 400 400 402 403 403 403 404
405 405 405 407 408 408 408 409 409 409 410 410 410 411 411 412 412
413 413 413 413 416 416 416 417 417 417 418 419 420 420 421 421 421
422 423 423 423 424 426 427 428 428 429 429 430 430 431 431 431 432
432 433 433 435 435 436 436 436 437 437 437 438 438 438 438 439 440
440 441 442 442 442 443 443 444 445 445 446 446 447 447 450 450 450
450 450 451 452 453 453 454 454 454 454 454 455 456 457 457 458 459
459 459 460 460 460 462 462 464 464 464 464 466 466 466 466 467 467
468 468 469 469 470 470 471 473 474 475 475 475 476 477 477 477 478
478 478 478 479 479 479 480 481 481 481 482 482 482 484 486 486 486
487 487 488 488 488 489 489 490 490 491 492 493 494 494 494 494 494
495 495 495 497 498 498 498 498 498 500 500 500 500 501 501 501 502
502 504 505 505 506 506 506 507 508 508 508 510 510 512 512 513 513
514 514 516 516 519 519 519 519 520 521 522 523 523 523 524 524 524
525 525 527 527 527 528 531 531 531 531 532 533 534 534 534 535 535
536 537 538 539 539 540 542 543 544 545 546 546 548 548 549 549 550
551 551 551 551 552 552 552 555 555 556 557 557 557 557 557 557 558
558 558 558 558 559 559 559 559 560 560 561 562 562 562 563 564 564
565 565 565 566 566 567 567 568 569 569 569 569 569 570 570 571 571
571 572 572 572 573 574 574 575 577 578 578 579 579 579 580 580 580
580 581 581 581 581 581 581 583 583 584 584 587 588 588 588 589 589
590 590 591 591 592 593 595 596 596 598 598 598 598 599

Analysis

The output is consistent across a wide range of randomly generated variations of `Array A` and `Array B`. The algorithm efficiently computes the transformations on the arrays even with an increasingly larger number of processors. The results demonstrate that the parallel merging algorithm is robust and scalable, handling larger arrays without any issues.