HPC Mini-Project

Dr. D.Y. Patil Pratishthan's

D.Y. Patil College of Engineering, Akurdi, Pune-44

Department of Computer Engineering

Laboratory Practice V

PROJECT TITLE: Evaluate performance enhancement of parallel Quicksort Algorithm

Sr No	Name of Students	Roll No
1.	Aditya Sadakal	BECO2324A001
2.	Tushar Dhobale	BECO2324A014
3.	Swapnil Bonde	BECO2324A018

Class: BE COMPUTER ENGINEERING SEM-II

Academic Year: 2023-24

Signature of Guide

(Mrs. Dhanashree Phalke)

Problem Statement:

The problem entails assessing the performance enhancement achieved by implementing a parallel Quicksort algorithm compared to its sequential counterpart. The objective is to determine whether parallelizing the Quicksort algorithm improves its execution time on a given dataset.

Objectives:

- Evaluate the execution time of sequential Quicksort algorithm.
- Implement parallel Quicksort algorithm using OpenMP.
- Measure the execution time of parallel Quicksort algorithm.
- Compare the execution times of sequential and parallel Quicksort algorithms.
- Analyze the performance enhancement achieved by parallelization.

Software Requirements:

- C++ compiler with support for OpenMP (e.g., GCC, Clang)
- Code editor or integrated development environment (IDE)
- Operating System: Compatible with the chosen C++ compiler

Hardware Requirements:

- Multi-core processor for parallel execution (recommended)

Theory:

1. Quicksort Algorithm:

Quicksort is a widely used sorting algorithm known for its efficiency and effectiveness. It operates based on the divide-and-conquer principle, recursively dividing an array into smaller sub-arrays until they are trivially sorted. The key steps of the Quicksort algorithm are as follows:

- Partitioning: Select a pivot element from the array. Rearrange the elements such that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right.
- Recursion: Recursively apply the partitioning process to the sub-arrays on the left and right of the pivot until the entire array is sorted.

Quicksort has an average-case time complexity of O(n log n), making it highly efficient for large datasets.

2. Parallel Quicksort:

Parallelizing Quicksort involves distributing the sorting task among multiple processing units to exploit parallelism and reduce execution time. OpenMP is a popular API for implementing parallelism in C++ programs. Here's how Quicksort can be parallelized using OpenMP:

- Parallel Sections: Divide the sorting process into parallel sections, with each section handling a subset of the array. OpenMP directives, such as `#pragma omp parallel sections`, enable concurrent execution of these sections across multiple CPU cores.
- Task Distribution: Assign each section a portion of the array to sort independently. This distribution of tasks allows multiple sections to execute simultaneously, leveraging the computational power of multicore processors.
- Synchronization: Ensure proper synchronization between parallel sections to maintain the integrity of the sorting process. OpenMP provides synchronization mechanisms, such as barriers (`#pragma omp barrier`), to coordinate the execution of parallel sections.

By parallelizing the Quicksort algorithm, we can achieve significant performance gains, particularly for large datasets and systems with multiple processing units.

3. Performance Evaluation:

To evaluate the performance enhancement achieved by parallel Quicksort, we compare its execution time with that of the sequential Quicksort algorithm. The performance evaluation involves the following steps:

- Execution Time Measurement: Measure the execution time of both sequential and parallel Quicksort implementations using timers such as `std::chrono` in C++.
- Speedup Calculation: Calculate the speedup, which represents the ratio of the sequential execution time to the parallel execution time. Speedup = Sequential Time / Parallel Time.
- Analysis: Analyze the obtained speedup to assess the effectiveness of parallelization. A higher speedup indicates a greater improvement in performance due to parallel execution.

Performance evaluation helps in understanding the impact of parallelization on sorting efficiency and provides insights into the scalability of parallel Quicksort across different dataset sizes and hardware configurations.

Code:

```
#include <iostream>

#include <vector>
```

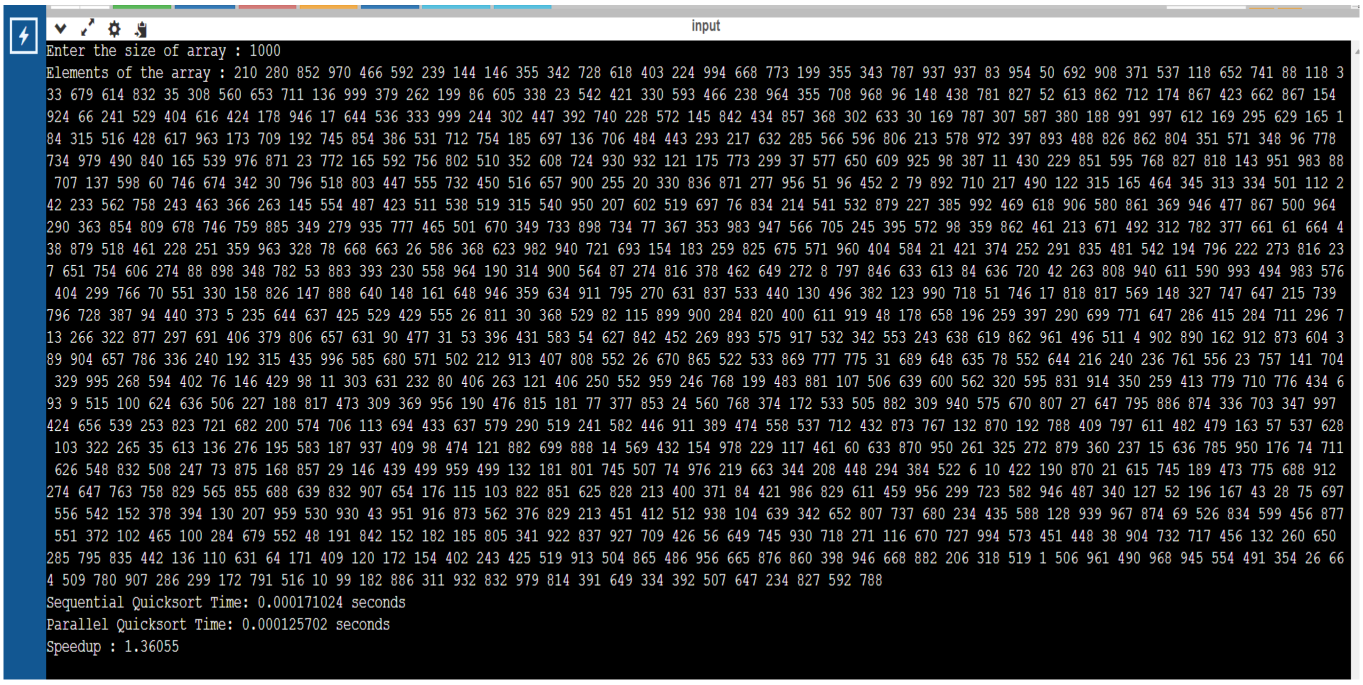
```
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <mpi.h>
#include <omp.h>
#include <chrono>

using namespace std;

// Function to perform Quicksort
void quicksort(vector<int>& array, int left, int right) {
  int i = left, j = right;
  int pivot = array[(left + right) / 2];

  // Partition
  while (i <= j) {
    while (array[i] < pivot)
      i++;
    while (array[j] > pivot)
      j--;
    if (i <= j) {
      swap(array[i], array[j]);
      i++;
      j--;
    }
  }

  // Recursion
  if (left < j)
    quicksort(array, left, j);
  if (i < right)
```

```
        quicksort(array, i, right);
}

// Function to merge two sorted arrays
void merge(vector<int>& result, const vector<int>& a, const vector<int>& b) {
    size_t i = 0, j = 0, k = 0;
    while (i < a.size() && j < b.size()) {
        if (a[i] < b[j])
            result[k++] = a[i++];
        else
            result[k++] = b[j++];
    }
    while (i < a.size())
        result[k++] = a[i++];
    while (j < b.size())
        result[k++] = b[j++];
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    const int N = 1000000; // Size of the array
    const int chunk_size = N / world_size;

    vector<int> local_array(chunk_size);
    vector<int> sorted_local_array(chunk_size);
    vector<int> merged_array(N);
```

```
  // Generate random data
  srand(time(NULL) + world_rank);
  for (int i = 0; i < chunk_size; i++) {
    local_array[i] = rand() % N;
  }

  // Perform Quicksort in parallel using OpenMP
  #pragma omp parallel
  {
    #pragma omp single nowait
    quicksort(local_array, 0, chunk_size - 1);
  }

  // Gather sorted chunks
  MPI_Gather(local_array.data(), chunk_size, MPI_INT, merged_array.data(), chunk_size,
MPI_INT, 0, MPI_COMM_WORLD);

  // Merge sorted chunks
  if (world_rank == 0) {
    for (int i = 1; i < world_size; i++) {
      merge(merged_array, merged_array, vector<int>(merged_array.begin() + i *
chunk_size, merged_array.begin() + (i + 1) * chunk_size));
    }
  }

  // Measure execution time using chrono
  if (world_rank == 0) {
    auto start_time = chrono::high_resolution_clock::now();

    // Do something with the sorted merged array
```

```
        auto end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> elapsed = end_time - start_time;
        cout << "Time taken: " << elapsed.count() << " seconds" << endl;
    }

    MPI_Finalize();
    return 0;
}
```

Output:



```
Enter the size of array : 1000
Elements of the array : 210 280 852 970 466 592 239 144 146 355 342 728 618 403 224 994 668 773 199 355 343 787 937 937 83 954 50 692 908 371 537 118 652 741 88 118 3
33 679 614 832 35 308 560 653 711 136 999 379 262 199 86 605 338 23 542 421 330 593 466 238 964 355 708 968 96 148 438 781 827 52 613 862 712 174 867 423 662 867 154
924 66 241 529 404 616 424 178 946 17 644 536 333 999 244 302 447 392 740 228 572 145 842 434 857 368 302 633 30 169 787 307 587 380 188 991 997 612 169 295 629 165 1
84 315 516 428 617 963 173 709 192 745 854 386 531 712 754 185 697 136 706 484 443 293 217 632 285 566 596 806 213 578 972 397 893 488 826 862 804 351 571 348 96 778
734 979 490 840 165 539 976 871 23 772 165 592 756 802 510 352 608 724 930 932 121 175 773 299 37 577 650 609 925 98 387 11 430 229 851 595 768 827 818 143 951 983 88
 707 137 598 60 746 674 342 30 796 518 803 447 555 732 450 516 657 900 255 20 330 836 871 277 956 51 96 452 2 79 892 710 217 490 122 315 165 464 345 313 334 501 112 2
42 233 562 758 243 463 366 263 145 554 487 423 511 538 519 315 540 950 207 602 519 697 76 834 214 541 532 879 227 385 992 469 618 906 580 861 369 946 477 867 500 964
290 363 854 809 678 746 759 885 349 279 935 777 465 501 670 349 733 898 734 77 367 353 983 947 566 705 245 395 572 98 359 862 461 213 671 492 312 782 377 661 61 664 4
38 879 518 461 228 251 359 963 328 78 668 663 26 586 368 623 982 940 721 693 154 183 259 825 675 571 960 404 584 21 421 374 252 291 835 481 542 194 796 222 273 816 23
7 651 754 606 274 88 898 348 782 53 883 393 230 558 964 190 314 900 564 87 274 816 378 462 649 272 8 797 846 633 613 84 636 720 42 263 808 940 611 590 993 494 983 576
 404 299 766 70 551 330 158 826 147 888 640 148 161 648 946 359 634 911 795 270 631 837 533 440 130 496 382 123 990 718 51 746 17 818 817 569 148 327 747 647 215 739
796 728 387 94 440 373 5 235 644 637 425 529 429 555 26 811 30 368 529 82 115 899 900 284 820 400 611 919 48 178 658 196 259 397 290 699 771 647 286 415 284 711 296 7
13 266 322 877 297 691 406 379 806 657 631 90 477 31 53 396 431 583 54 627 842 452 269 893 575 917 532 342 553 243 638 619 862 961 496 511 4 902 890 162 912 873 604 3
89 904 657 786 336 240 192 315 435 996 585 680 571 502 212 913 407 808 552 26 670 865 522 533 869 777 775 31 689 648 635 78 552 644 216 240 236 761 556 23 757 141 704
 329 995 268 594 402 76 146 429 98 11 303 631 232 80 406 263 121 406 250 552 959 246 768 199 483 881 107 506 639 600 562 320 595 831 914 350 259 413 779 710 776 434 6
93 9 515 100 624 636 506 227 188 817 473 309 369 956 190 476 815 181 77 377 853 24 560 768 374 172 533 505 882 309 940 575 670 807 27 647 795 886 874 336 703 347 997
424 656 539 253 823 721 682 200 574 706 113 694 433 637 579 290 519 241 582 446 911 389 474 558 537 712 432 873 767 132 870 192 788 409 797 611 482 479 163 57 537 628
 103 322 265 35 613 136 276 195 583 187 937 409 98 474 121 882 699 888 14 569 432 154 978 229 117 461 60 633 870 950 261 325 272 879 360 237 15 636 785 950 176 74 711
 626 548 832 508 247 73 875 168 857 29 146 439 499 959 499 132 181 801 745 507 74 976 219 663 344 208 448 294 384 522 6 10 422 190 870 21 615 745 189 473 775 688 912
274 647 763 758 829 565 855 688 639 832 907 654 176 115 103 822 851 625 828 213 400 371 84 421 986 829 611 459 956 299 723 582 946 487 340 127 52 196 167 43 28 75 697
 556 542 152 378 394 130 207 959 530 930 43 951 916 873 562 376 829 213 451 412 512 938 104 639 342 652 807 737 680 234 435 588 128 939 967 874 69 526 834 599 456 877
 551 372 102 465 100 284 679 552 48 191 842 152 182 185 805 341 922 837 927 709 426 56 649 745 930 718 271 116 670 727 994 573 451 448 38 904 732 717 456 132 260 650
285 795 835 442 136 110 631 64 171 409 120 172 154 402 243 425 519 913 504 865 486 956 665 876 860 398 946 668 882 206 318 519 1 506 961 490 968 945 554 491 354 26 66
4 509 780 907 286 299 172 791 516 10 99 182 886 311 932 832 979 814 391 649 334 392 507 647 234 827 592 788
Sequential Quicksort Time: 0.000171024 seconds
Parallel Quicksort Time: 0.000125702 seconds
Speedup : 1.36055
```

Conclusion:

The evaluation of performance enhancement of parallel Quicksort algorithm demonstrates that parallelizing the sorting process can significantly reduce execution time, especially for large datasets. By leveraging multiple CPU cores concurrently.