COMP 6xx (Fall 2021): project 2
Name: Hossein Alishah
Student ID: 202048619
Section: Wed (7:00 pm – 9:45 pm)

Project 2. Design and implementation of a parallel sort algorithm using OpenMP:

Answer: I used ***C programing*** language:

For implementing this program, I used ***mergesort algorithm***, and for the array, I asked user to insert the size of array, then the program make an array with the random elements.

Mergesort is one of the most popular sorting techniques. It is the typical example for demonstrating the divide-and-conquer paradigm.

Mergesort has the worst-case serial growth as *O(nlogn).*

Sorting an array: A*[p .. r]* using *mergesort* involves three steps:

- *Divide Step*
- *Conquer Step*
- *Combine Step*

We can parallelize the "conquer" step where the array is recursively sorted amongst the left and right subarrays. We can 'parallelly' sort the left and the right subarrays.

In what follows, you are going to see the ***main*** implementation:

Here I ask user to put the array size, and there is some codes to show the elements of sorted array and the time of implementation for seral and parallel functions.

```c
openMP-project2.c

#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

void mergesort_parallel(int a[],int i,int j);
void merge_sorted(int a[], int p, int q, int r);
void mergesort_serial(int a[],int i,int j);

int main()
{
    printf("Enter the size of the data to be sorted (the maximum value is 100000, enter 0 to end) : ");
    int *a, num, i;
    scanf("%d",&num);

    a = (int *)malloc(sizeof(int) * num);

    for (i = 0; i < num; i++) {
        a[i] = rand() % 1000 + 1;
    }

    double start_time = omp_get_wtime();
    mergesort_parallel(a,0,num-1);
    double end_time = omp_get_wtime();
    double time_used = end_time - start_time;
    printf("\nParallel time: %f s\n\n\n", time_used);

    double start_time1 = omp_get_wtime();
    mergesort_serial(a,0,num-1);
    double end_time1 = omp_get_wtime();
    double time_used1 = end_time1 - start_time1;
    printf("\nSerial time: %f s\n\n\n", time_used1);

    printf("\nSorted array :\n");
    for(i=0;i<num;i++)
        printf("%d ",a[i]);

    return 0;
}
```

Now, the following function is the implementation of **Serial *Mergesort***:

```c
void mergesort_serial(int a[],int i,int j)
{
    int mid;

    if(i<j)
    {
        mid=(i+j)/2;
        mergesort_serial(a,i,mid);        //left recursion
        mergesort_serial(a,mid+1,j);    //right recursion
        merge_sorted(a,i, mid, j);    //merging of two sorted sub-arrays
    }
}
```

As you can see on Serial merge sort, we have another function that is called ***merge_sorted( ).*** Its task is to merge two subarrays a[i..mid] and a[mid+1..j] to create a sorted array a[i..j]. So the inputs to the function are a, i, mid and j.

The merge function works as follows:

1. Create copies of the subarrays L ← A[p..q] and M ← A[q+1..r].
2. Create three pointers i, j and k

    - i maintains current index of L, starting at 1
    - j maintains current index of M, starting at 1
    - k maintains the current index of A[p..q], starting at p.
3. Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]
4. When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

```
// Merge two subarrays L and M into arr
void merge_sorted(int a[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = a[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = a[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      a[k] = L[i];
      i++;
    } else {
      a[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    a[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    a[k] = M[j];
    j++;
    k++;
  }
}
```

Parallelizing Merge Sort Through OpenMP

The parallelizable region is the "conquer" part. We need to make sure that the left and the right sub-arrays are sorted simultaneously. We need to implement both left and right sections in parallel. This can be done in OpenMP using directive:

#pragma omp parallel sections

And each section that has to be parallelized should be enclosed with the directive:

#pragma omp section

Now, let's work on parallelizing both sections through OpenMP:

```c

void mergesort_parallel(int a[],int i,int j)
{
    int mid;

    if(i<j)
    {
        mid=(i+j)/2;

        #pragma omp parallel sections
        {

            #pragma omp section
            {
                mergesort_parallel(a,i,mid);        //left recursion
            }

            #pragma omp section
            {
                mergesort_parallel(a,mid+1,j);    //right recursion
            }
        }

       // merge(a,i,mid,mid+1,j);    //merging of two sorted sub-arrays
        merge_sorted(a,i, mid, j);
    }

}
```

At the end, I am going to run this program for some different size of array:

1. n= 50

```
Line: 41  Col: 15

Enter the size of the data to be sorted (the maximum value is 100000, enter 0 to
    end) : 50

Serial time: 0.000463 s



Parallel time: 0.000007 s



Sorted array :
43 74 92 98 100 150 158 166 170 195 229 250 268 273 279 304 328 336 337 358 394
    441 486 493 504 513 545 561 580 613 634 659 673 710 710 730 746 808 811 817
    822 827 841 879 924 931 934 968 980 988

Program ended with exit code: 0
```

2. n= 1000

```
Enter the size of the data to be sorted (the maximum value is 100000, enter 0 to
    end) : 1000

Serial time: 0.000875 s



Parallel time: 0.000229 s



Sorted array :
1 1 1 2 4 5 7 7 7 9 9 9 10 11 11 12 12 13 14 15 15 15 15 16 16 16 17 20 20 25 25
    26 27 29 29 29 30 30 32 33 34 34 35 36 36 38 39 40 41 43 43 44 45 45 46 46 47
    49 49 49 54 55 55 55 56 58 58 61 64 65 65 66 68 71 74 80 81 81 83 84 87 88 90
    90 91 92 92 93 95 95 95 96 97 98 98 99 100 101 102 103 104 104 106 107 108 108
    109 109 109 111 112 113 113 114 114 114 115 116 117 118 121 122 124 124 125
    125 126 127 128 128 129 130 130 133 134 134 138 138 138 139 141 142 142 143
    145 145 150 151 151 151 151 152 153 153 153 154 156 158 159 160 164 165 166
    166 167 168 170 170 172 174 176 177 177 178 178 179 180 181 189 190 193 194
    195 196 196 197 197 198 199 201 201 201 204 204 206 210 211 211 212 212 217
    218 218 218 219 221 221 222 224 224 225 227 227 229 229 230 231 231 232 232
    232 233 235 236 237 238 239 239 240 241 241 245 245 245 246 246 248 248 250
    250 250 251 252 253 253 254 254 255 257 258 259 259 261 261 262 263 264 267
    267 268 270 273 273 275 275 276 279 279 282 284 285 286 286 289 290 290 290
    291 294 294 295 296 296 296 298 299 299 299 300 300 300 301 302 304 304 304
    305 305 306 306 308 311 312 313 314 316 317 318 318 319 320 321 322 322 323
    325 325 328 328 329 330 332 332 333 336 336 336 337 338 339 343 343 349 351
    353 353 354 355 357 357 358 358 358 359 359 360 360 361 364 364 365 365 366
    366 367 367 369 371 371 371 371 374 378 378 378 379 380 381 382 383 385 385
    386 386 386 386 391 392 394 394 395 395 396 396 397 397 398 398 400 401 401
    404 405 407 408 409 411 411 411 411 413 413 414 415 416 416 419 419 420 420
    421 421 422 424 426 428 432 432 435 436 438 439 440 440 441 442 443 445 445
    446 448 448 449 451 451 452 454 455 455 455 456 460 460 462 463 464 466 467
    467 468 468 472 474 477 481 481 484 486 488 490 490 491 493 495 495 497 497
    499 502 502 502 502 502 503 504 504 504 504 505 506 507 507 508 510 511 512
    513 515 516 517 517 518 518 519 519 520 521 521 523 526 526 531 531 532 532
    534 534 535 535 535 537 537 538 540 541 543 544 544 544 545 545 546 547 548
    550 550 550 551 551 551 552 552 553 554 554 554 555 556 556 557 558 558 558
```

3. n= 50,000

```
Enter the size of the data to be sorted (the maximum value is 100000, enter 0 to
    end) : 50000

Serial time: 0.016637 s



Parallel time: 0.006736 s



Sorted array :
1 1 1 2 4 5 7 7 7 9 9 9 10 11 11 12 12 13 14 15 15 15 15 16 16 16 17 20 20 25 25
    26 27 29 29 29 30 30 32 33 34 34 35 36 36 38 39 40 41 43 43 44 45 45 46 46 47
    49 49 49 54 55 55 55 56 58 58 61 64 65 65 66 68 71 74 80 81 81 83 84 87 88 90
    90 91 92 92 93 95 95 95 96 97 98 98 99 100 101 102 103 104 104 106 107 108 108
    109 109 109 111 112 113 113 114 114 114 115 116 117 118 121 122 124 124 125
    125 126 127 128 128 129 130 130 133 134 134 138 138 138 139 141 142 142 143
    145 145 150 151 151 151 151 152 153 153 153 154 156 158 159 160 164 165 166
    166 167 168 170 170 172 174 176 177 177 178 178 179 180 181 189 190 193 194
    195 196 196 197 197 198 199 201 201 201 204 204 206 210 211 211 212 212 217
    218 218 218 219 221 221 222 224 224 225 227 227 229 229 230 231 231 232 232
    232 233 235 236 237 238 239 239 240 241 241 245 245 245 246 246 248 248 250
    250 250 251 252 253 253 254 254 255 257 258 259 259 261 261 262 263 264 267
    267 268 270 273 273 275 275 276 279 279 282 284 285 286 286 289 290 290 290
    291 294 294 295 296 296 296 298 299 299 299 300 300 300 301 302 304 304 304
    305 305 306 306 308 311 312 313 314 316 317 318 318 319 320 321 322 322 323
    325 325 328 328 329 330 332 332 333 336 336 336 337 338 339 343 343 349 351
    353 353 354 355 357 357 358 358 358 359 359 360 360 361 364 364 365 365 366
    366 367 367 369 371 371 371 371 374 378 378 378 379 380 381 382 383 385 385
    386 386 386 386 391 392 394 394 395 395 396 396 397 397 398 398 400 401 401
    404 405 407 408 409 411 411 411 411 413 413 414 415 416 416 419 419 420 420
    421 421 422 424 426 428 432 432 435 436 438 439 440 440 441 442 443 445 445
    446 448 448 449 451 451 452 454 455 455 455 456 460 460 462 463 464 466 467
    467 468 468 472 474 477 481 481 484 486 488 490 490 491 493 495 495 497 497
    499 502 502 502 502 502 503 504 504 504 504 505 506 507 507 508 510 511 512
    513 515 516 517 517 518 518 519 519 520 521 521 523 526 526 531 531 532 532
    534 534 535 535 535 537 537 538 540 541 543 544 544 544 545 545 546 547 548
    550 550 550 551 551 551 552 552 553 554 554 554 555 556 556 557 558 558 558
    561 561 561 562 563 564 564 565 566 567 567 568 571 572 575 576 577 577 580
    580 581 582 582 585 586 588 589 590 590 591 591 592 593 594 594 595 595 597
    598 599 602 604 606 608 610 611 613 613 615 616 617 617 622 622 623 624 624
    624 624 626 626 626 628 630 631 631 632 634 634 636 636 639 640 640 641 642
    644 644 647 649 650 651 652 652 653 655 657 658 659 659 662 662 662 663 663
    664 667 667 667 668 669 670 670 670 670 671 672 672 672 672 673 676 676 677
    677 680 680 680 681 684 685 685 690 691 693 694 696 697 699 699 699 700 700
```