ISSN 2307-5732
DOI 10.31891/2307-5732

НАУКОВИЙ ЖУРНАЛ

4.2023

ВІСНИК

Хмельницького національного університету

Технічні науки

Technical sciences

SCIENTIFIC JOURNAL
HERALD OF KHMELNYTSKYI NATIONAL UNIVERSITY

2023, Issue 4, Volume 323

Хмельницький

ЗМІСТ

DOI 10.31891/2307-5732-2023-323-4-95-105
УДК 004.421

ДЕНИСЮК ВАЛЕРІЙ
Вінницький національний технічний університет
ORCID ID: 0000-0003-1057-3518
e-mail: vad64@i.ua
ПОТАПОВА НАДІЯ
Донецький національний університет імені Василя Стуса м.Вінниця
ORCID ID: 0000-0003-4566-4102
e-mail: potapova.nadin@gmail.com
ЗЕЛІНСЬКА ОКСАНА
Донецький національний університет імені Василя Стуса м.Вінниця
ORCID ID: 0000-0002-9069-1428
e-mail: zeloksanavlad@gmail.com
ТАРАСЮК МИКОЛА
Вінницький національний технічний університет
e-mail: tarasyuk.m12@gmail.com

ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ АЛГОРИТМІВ ПАРАЛЕЛЬНОГО ШВИДКОГО СОРТУВАННЯ

Робота присвячена програмній реалізації та дослідженню алгоритмів паралельного швидкого сортування. А саме для випадку, коли немає можливості використати надбання технологій OpenMP або CUDA, які орієнтовані на C++. Розроблено оригінальну програму, яка реалізує паралельні алгоритми сортування у вигляді консольного додатку засобами мови C#. Розвиток розподілених систем та паралельних обчислень впливає на розвиток алгоритмів з використанням паралельних технологій. Виникає необхідність порівняння ефективності алгоритмів, що використовують технології розподілених систем та паралельних обчислень. Для програмної реалізації обрано відомі алгоритми паралельного швидкого сортування: послідовне швидке сортування; наївне паралельне сортування; оптимізоване паралельне сортування; паралельно-послідовне сортування; гіпершвидке сортування; паралельне швидке сортування шляхом регулярної вибірки. Надано матеріал по кожному алгоритму мовою C#. Зроблено порівняння швидкодії розглянутих паралельних алгоритмів сортування. Можливість порівняння ефективності алгоритмів є цікавою та пізнавальною в учбовому процесі підготовки IT-спеціалістів. Програмна реалізація дослідження алгоритмів сортування виконується в ітеративному режимі за кілька кроків: завантажити програму; у відкритому консольному вікні ввести бажану довжину масиву сортування; ввести мінімальне значення елемента масиву; ввести максимальне значення елемента масиву: із списку доступних алгоритмів сортування обрати бажаний; після вибору алгоритму сортування відбувається сортування масиву, виводиться час сортування та відсортований масив. У результаті досліджень з'ясовано, що для технологій, пов'язаних з C#, найкращим із розглянутих алгоритмів є гіпершвидке сортування, а прийнятними - оптимізоване паралельне сортування або наївне паралельне сортування.

Ключові слова: програма, алгоритм, сортування, паралельне сортування, паралельні обчислення.

DENYSIUK VALERII
Vinnytsia National Technical University, Vinnytsia, Ukraine
POTAPOVA NADIIA
Vasyl' Stus Donetsk National University, Vinnytsia, Ukraine
ZELINSKA OKSANA
Vasyl' Stus Donetsk National University, Vinnytsia, Ukraine
TARASIUK MYKOLA
Vinnytsia National Technical University, Vinnytsia, Ukraine

SOFTWARE IMPLEMENTATION AND RESEARCH OF QUICK SORTING ALGORITHMS

The article is devoted to software implementation and research of parallel quick sorting algorithms. Namely for the case when it is not possible to use the assets of OpenMP or CUDA technologies, which are oriented towards C++. An original program has been developed that implements parallel sorting algorithms in the form of a console application using the C#. The development of distributed systems and parallel computing affects the development of algorithms using parallel technologies. There is a need to compare the efficiency of algorithms using technologies of distributed systems and parallel computing. Well-known parallel quick sorting algorithms were selected for software implementation: sequential quick sorting; naive parallel quick sorting; optimized parallel quick sorting; parallel-serial sorting; hyperquick sorting; parallel quicksort by regular sampling. Material on each algorithm in C# is provided. A comparison of the speed of the considered parallel sorting algorithms was made. The possibility of comparing the efficiency of algorithms is interesting and informative in the educational process of training IT specialists. The software implementation of the research of quick sorting algorithms is performed in an iterative mode in a few steps: download the program; in the open console window, enter the desired length of the sorting array for generation; enter the minimum value of the array element; enter the maximum value of the array element: choose the desired one from the list of available sorting algorithms (enter the serial number from the list of algorithms from 1 to 6); after selecting the sorting algorithm, the array is sorted, then the sorting time in milliseconds (ms) is displayed. and a sorted array. As a result of the research, it was found that for technologies related to C#, the best of the considered algorithms is hyperfast sort, and optimized parallel sort or naive parallel sort are acceptable.

Key words: program, algorithm, sorting, parallel sorting, parallel computations

Постановка проблеми

Для структуризації та впорядкування різноманітних масивів даних і для їх подання у певному вигляді використовуються алгоритми сортування. Для сортування даних було розроблено безліч алгоритмів, що мають як свої переваги, так і недоліки [1, 2]. З розвитком розподілених систем та паралельних обчислень розвиваються і алгоритми з використанням даних технологій [3]. Такий підхід вдосконалює та підвищує ефективність самих алгоритмів з використанням паралельних технологій. Виникає необхідність порівняння ефективності алгоритмів, що використовують технології розподілених систем та паралельних обчислень. Також можливість порівняння ефективності алгоритмів є цікавою та пізнавальною в учбовому процесі підготовки IT-спеціалістів. Все це поставило задачу розробки програми, яка реалізує паралельні алгоритми сортування у вигляді консольного додатку засобами мови C# [4, 5] для випадків, коли немає можливості скористатися надбаннями технологій OpenMP або CUDA орієнтованих на C++ [6, 7].

Аналіз останніх досліджень

Існує порівняння алгоритмів сортування для паралельних обчислень [3]: послідовне швидке сортування; наївне паралельне сортування; оптимізоване паралельне сортування; паралельно-послідовне сортування; гіпершвидке сортування; паралельне швидке сортування шляхом регулярної вибірки. Найефективнішим із розглянутих названо алгоритм паралельного швидкого сортування шляхом регулярної вибірки.

Формулювання цілей

Для тестування вищеназваних алгоритмів необхідно створити програму, яка реалізує паралельні алгоритми сортування та матиме наступні можливості: генерування масиву з випадковими значеннями за розміром заданим користувачем; сортування випадково згенерованого масиву даних за допомогою обраних алгоритмів; вимірювання часу витраченого на сортування масиву з N елементів. Обрані алгоритми: послідовне швидке сортування; наївне паралельне сортування; оптимізоване паралельне сортування; паралельно-послідовне сортування; гіпершвидке сортування; паралельне швидке сортування шляхом регулярної вибірки. Практична програмна реалізація названих алгоритмів на C# дозволить порівняти їх складність та швидкодію.

Виклад основного матеріалу

Розглянемо особливості обраних алгоритмів [3] та їх реалізацію засобами C# [4, 5].

Алгоритм послідовного швидкого сортування.

1. Знайти випадковий опорний елемент ***p= list[i]***.
2. Розбити список відповідно до опорного елементу, елементи, менші за опорний, розташувати ліворуч від опорного елементу, елементи, більші за опорний - праворуч від опорного елементу, а елементи, що дорівнюють опорному елементу, посередині. ***<p = p> p*** ;виконати наступні кроки: ініціалізувати ***i*** для першого елемента в списку, а ***j*** для останнього елемента; збільшувати ***i*** до тих пір, доки ***list[i]*** не стане опорним елементом; зменшувати ***j*** до тих пір, доки ***list[j]*** не стане опорним елементом; повторювати описані вище кроки, доки ***i > j***; замінити опорний елемент на ***list[j]***
3. Рекурсивно виконати попередні кроки.
4. Коли розмір списку дорівнює ***1***, він завершується. Це діє як базовий випадок. На цьому етапі розділи відсортовані, тому вони об’єднуються, утворюючи повний відсортований список.

Алгоритм послідовного швидкого сортування працює крок за кроком. Перш ніж почати наступний, потрібно закінчити попередній крок. Його часова складність у середньому становить ***O(nlogn)***, просторова складність становить ***O(logn).*** На рис.1 надано лістинг класу SequentialQuickSort.

Алгоритм наївного паралельного сортування (запускає процес на кожному кроці для одночасної обробки частин).

1. Знайти опорний елемент та розділити список на дві частини, ***p < =p > p***.
2. На кожному кроці запускати ***p*** процесів, пропорційних ***n*** розділам.
3. Кожен процес знаходить опорний елемент і ділить список на основі обраного опорного елемента.
4. Значення процесів об’єднуються, повертається відсортований список.

В алгоритмі наївного паралельного сортування час для вибору опорної точки та перевпорядкування списку становить ***θ(n)***; на кожному кроці працюють ***n*** процесів; загальна часова складність становить $\boldsymbol{\theta(n^2)}$.

На рис.2 надано лістинг класу NaiveParallelQuickSort.

Алгоритм оптимізованого паралельного сортування (у алгоритмі змінюється кількість процесів, які використовуються на кожному кроці. замість подвоєння кількості процесів на кожному кроці; використовується ***n*** процесів у всьому алгоритмі для пошуку опорного елемента та зміни порядку в списку; усі процеси виконуються одночасно на кожному кроці сортування списків).

1. Запустити ***n*** процесів, які розділять список і відсортують його за допомогою обраного опорного елемента.
2. ***n*** процесів обробляються на всіх частинах від початку алгоритму до кінця сортування списку.
3. Кожен процес знаходить опорний елемент і розділяє список на основі обраного елементу.
4. Відсортований список отримуємо об’єднанням списків процесів.

На кожному кроці ***n*** процесів обробляє ***log(n)*** списків за постійний час ***O(1).*** Час паралельного виконання на ***n*** процесах дорівнює ***O(logn).*** Загальна часова складність становить ***θ(nlogn)*** , вона не змінилася порівняно з алгоритмом послідовного швидкого сортування, але оптимізоване паралельне

сортування виконується на паралельних процесорах, отже, буде виконуватися набагато швидше для більших ***n***. Складність простору дорівнює ***O(logn).*** На рис.3 надано лістинг класу OptimizedParallelQuickSort.

Алгоритм паралельно-послідовного сортування (початковий список ділиться на ***n*** менших підсписків, які явно надсилаються на ***p*** віддалених процесорів для паралельного виконання, після завершення виконання на розподіленому процесорі відсортований підсписок надсилається назад до центрального вузла обробки, який об'єднує результати процесів, створюючи повністю відсортований список).

```
namespace hyperquicksort
{   public static class SequentialQuickSort
     {   public static void Sort<T>(T[] array) where T : IComparable<T>
         {
              Sort(array, 0, array.Length - 1);
         }

         private static void Sort<T>(T[] array, int left, int right)where T :
IComparable<T>
         {   if (left < right)
              {
                   int pivotIndex = Partition(array, left, right);
                   Sort(array, left, pivotIndex - 1);
                   Sort(array, pivotIndex + 1, right);
              }
         }

         private static int Partition<T>(T[] array, int left, intright)
where T : IComparable<T>
         {    int pivotIndex = GetPivotIndex<int>(left, right);
              T pivotValue = array[pivotIndex];
              Swap(ref array[pivotIndex], ref array[right]);int
              storeIndex = left;
              for (int i = left; i < right; i++)
              {   if (array[i].CompareTo(pivotValue) < 0)
                   {
                        Swap(ref array[i], ref array[storeIndex]);
                        storeIndex++;
                   }
              }
              Swap(ref array[storeIndex], ref array[right]);return
              storeIndex;
         }

         private static int GetPivotIndex<T>(int left, int right)
         {
              return left + (right - left) / 2;
         }

         private static void Swap<T>(ref T a, ref T b)
         {
              T temp = a;
              a = b;
              b = temp;
         }
     }
}
```

Рис.1. Клас SequentialQuickSort

1. Розділити список розміром ***n***, щоб створити кількість підсписків, сумісних із кількістю доступних процесорів ***p***.
2. Створити p потоків відповідно до кількості доступних процесорів.
3. Призначити підсписок кожному із p потоків, аби кожний потік мав n/p послідовних елементів із вихідного списку.

```
namespace hyperquicksort
{
     public static class NaiveParallelQuickSort
     {    private const int SEQUENTIAL_THRESHOLD = 2048;
          public static void Sort<T>(T[] arr) where T : IComparable<T>
          {   Sort(arr, 0, arr.Length - 1);
          }

          private static void Sort<T>(T[] arr, int left, int right)where T :
IComparable<T>
          {  if (left < right)
            {   if (right - left < SEQUENTIAL_THRESHOLD)
            {
          }
            else
            {     SequentialQuickSort(arr, left, right);
                  Int pivotIndex = Partition(arr, left, right);
                  // Invoke two tasks to sort the left and right
                  halves of the array in parallel
                  Parallel.Invoke(
                        () => Sort(arr, left, pivotIndex - 1),
                        () => Sort(arr, pivotIndex + 1, right)
                  );
            }
      }
}
          private static void SequentialQuickSort<T>(T[] arr, int left,int right)
where T : IComparable<T>
          {
               if (left < right)
               {   int pivotIndex = Partition(arr, left, right);
                   SequentialQuickSort(arr, left, pivotIndex - 1);
                   SequentialQuickSort(arr, pivotIndex + 1, right);
               }
          }
          private static int Partition<T>(T[] arr, int left, int right)
          where T : IComparable<T>
          {  int pivotIndex = GetPivotIndex<int>(left, right);
            T pivotValue = arr[pivotIndex];
            Swap(ref arr[pivotIndex], ref arr[right]);int
            storeIndex = left;
            for (int i = left; i < right; i++)
               {    if (arr[i].CompareTo(pivotValue) < 0)
                    {    Swap(ref arr[i], ref arr[storeIndex]);storeIndex++;
                    }
               }
               Swap(ref arr[storeIndex], ref arr[right]);
               return storeIndex;
          }
          private static int GetPivotIndex<T>(int left, int right)
          {   return left + (right - left) / 2;
          }

          private static void Swap<T>(ref T a, ref T b)
          {   T temp = a;a = b; b = temp;
          }
     }
}
```

Рис.2. Клас NaiveParallelQuickSort.

4. Довільним чином обрати основний елемент та передати його усім процесам-партнерам.
5. У кожному процесі, одночасно (паралельно) в усіх процесах, поділити елементи на дві групи відповідно до обраної опорної точки, ***group1*** <= ***pivot*** <= ***group2***.

```
namespace hyperquicksort
{    public static class OptimizedParallelQuickSort
     {    private const int SEQUENTIAL_THRESHOLD = 2048;
          public static void Sort<T>(T[] array) where T : IComparable<T>
          {    Sort(array, 0, array.Length - 1);
          }

          private static void Sort<T>(T[] array, int left, int right)where T :
IComparable<T>
          {    if (left < right)
               {    if (right - left < SEQUENTIAL_THRESHOLD)
                    {    SequentialQuickSort(array, left, right);
                    }
                    else
                    {    int pivotIndex = Partition(array, left, right);
                         // Invoke two tasks to sort the left and right
                         halves of the array in parallel
                         Parallel.Invoke(
                              () => Sort(array, left, pivotIndex - 1),() =>
                              Sort(array, pivotIndex + 1, right)
                         );
                    }
               }
          }
          private static void SequentialQuickSort<T>(T[] array, intleft, int
right) where T : IComparable<T>
          {    if (left < right)
               {    int pivotIndex = Partition(array, left, right);
                    SequentialQuickSort(array, left, pivotIndex - 1);
                    SequentialQuickSort(array, pivotIndex + 1, right);
               }
          }

          private static int Partition<T>(T[] array, int left, intright)
where T : IComparable<T>
          {    int pivotIndex = GetPivotIndex<int>(left, right);
               T pivotValue =  array[pivotIndex];
               Swap(ref array[pivotIndex], ref array[right]);
               int storeIndex = left;
               for (int i = left; i < right; i++)
               {    if (array[i].CompareTo(pivotValue) < 0)
                    {    Swap(ref array[i], ref array[storeIndex]);storeIndex++;
                    }
               }
               Swap(ref array[storeIndex], ref array[right]);return
               storeIndex;
          }

          private static int GetPivotIndex<T>(int left, int right)
          {    return left + (right - left) / 2;
          }

          private static void Swap<T>(ref T a, ref T b)
          {    T temp = a;a = b; b = temp;
          }
     }
}
```

Рис.3. Клас OptimizedParallelQuickSort

6. Кожен процес у верхній половині списку процесів надсилає свій «нижчий список» процесу-партнеру в нижній половині списку процесів, а той у відповідь отримує «старший список».
7. Нижня половина матиме значення, менші за опорні, а елементи верхньої половини матимуть значення, більші за опорні.
8. Після ***logP*** рекурсій кожен процес має невідсортований підсписок, який не перетинається зі значеннями інших процесів. Найбільше значення процесу ***i*** буде менше найменшого значення процесу ***i+1.*** На цьому етапі підсписок має досить малий розмір, кожен процес сортує свої значення послідовно, а основний процес об'єднує відсортовані результати кожного процесу.

В алгоритмі паралельно-послідовного сортування ***p*** процесів працює з ***n*** елементами списку, кожний ***p*** процес виконує ***(logn)*** кроків з ***n/p*** елементами підсписку.

Загальна часова складність дорівнює $O(\frac{n}{p} * logn)$. Просторова складність становить ***O(logn)***.

Алгоритм погано справляється з балансуванням навантаження, необхідно обрати відповідне середнє значення як опорний елемент для алгоритму, щоб розділити список принаймні на однакові частини та зберегти баланс. Пошук середнього значення є дорогою операцією на паралельному процесорі. Тому краще знайти середнє значення, яке є близьким до дійсного медіани. Об'єднуються блоки процесів за порядком процесів - визначається початок і кінець кожного блоку та приєднується його кінець до початку блоку наступного процесу. Наприклад, машина має ***64*** потоки, обробку списку розміром ***n*** можна розподілити на ***n/64*** підсписки; обробка підсписків відбувається паралельно; кожний підсписок послідовно сортується та об'єднується з іншими для отримання результату; при збільшенні ***n*** збільшується розмір підсписку, який обробляє кожен потік. Кожен потік обробляє список за постійний час O(1) за logn кроків. Для потоків, які працюють паралельно, часова та просторова складність становить O(logn). На рис.4 надано лістинг класу ParallelSequentialQuickSort.

Алгоритм гіпершвидкого сортування (є вдосконаленням алгоритму паралельно-послідовного сортування, частково розв'язує проблему балансування навантаження, покращує шанси знайти справжню медіану шляхом послідовного сортування підсписків за допомогою однієї опорної точки, яка транслюється всім процесам на початку алгоритму).

1. Список розміром n розділено між n процесами. Наприклад, є список розміром 64 елементи, розпаралелити можна на ***8*** процесів, тоді кожен процес оброблятиме ***8*** елементів списку.
2. Процес серед ***8*** елементів знаходить опорний елемент і передає його всім процесам, які сортують свої підсписки послідовно за допомогою широкомовного опорного елемента. Це покращить шанси знайти опорні точки, близькі до справжньої медіани.
3. Обрати опорний елемент та передати його усім процесам-партнерам (вибір Pivot і його трансляція).
4. У кожному процесі, одночасно (паралельно) в усіх процесах, поділити елементи на дві групи відповідно до обраної опорної точки, ***group1 <= pivot <= group2*** (поділ підсписку низьких і високих значень).
5. Кожен процес у верхній половині списку процесів надсилає свій «нижчий список» процесу-партнеру в нижній половині списку процесів, а той у відповідь отримує «старший список» (обмін значеннями між партнерськими процесами).
6. Залишок верхньої половини від одного партнерського процесу та отримана верхня половина від іншого партнерського процесу об'єднуються в локальний підсписок для кожного процесу.
7. Використовується рекурсія для верхньої та нижньої половини кожного підпроцесу, щоб створити відсортований список.
8. Для отримання повністю відсортованого списку об'єднуються процеси.

В цьому алгоритмі існують накладні витрати на зв'язок для передачі значень між процесами-партнерами. Може виникати дисбаланс навантаження, але алгоритм кращий порівняно з алгоритмом паралельно-послідовного сортування, який набагато гірше балансує навантаження.В алгоритмі гіпершвидкого сортування ***logn*** кроків та ***n*** процесів, загальна часова складність становить ***θ(nlogn).*** Просторова складність дорівнює ***O(logn).*** На рис.5 надано лістинг класу HyperQuickSort.

Алгоритм гіпершвидкого сортування серед обраних алгоритмів є паралельним за визначенням, оскільки в його основі лежить вдосконалений метод паралельно-послідовного сортування. В свою чергу цей алгоритм складається із паралельного і послідовного методу сортування, відповідно до назви.

```
namespace hyperquicksort
{    public static class ParallelSequentialQuickSort
     {    private const int SEQUENTIAL_THRESHOLD = 2048;
          public static void Sort<T>(T[] array) where T : IComparable<T>
          {    Sort(array, 0, array.Length - 1);
          }

          private static void Sort<T>(T[] array, int left, int right)
          where T : IComparable<T>
          {    if (left < right)
               {    if (right - left < SEQUENTIAL_THRESHOLD)
                    {    SequentialQuickSort(array, left, right);
                    }
                    else
                    {
                    int pivotIndex = Partition(array, left, right);
                    // Invoke two tasks to sort the left and right
                     halves of the array in parallel
                         Parallel.Invoke(
                              () => Sort(array, left, pivotIndex - 1),() =>
                              Sort(array, pivotIndex + 1, right)
                         );
                    }

                }
           }

          private static void SequentialQuickSort<T>(T[] array, intleft, int
right) where T : IComparable<T>
          {    if (left < right)
               {    int pivotIndex = Partition(array, left, right);
                    SequentialQuickSort(array, left, pivotIndex - 1);
                    SequentialQuickSort(array, pivotIndex + 1, right);
               }
          }

          private static int Partition<T>(T[] array, int left, intright)
where T : IComparable<T>
          {    int pivotIndex = GetPivotIndex<int>(left, right);
               T pivotValue = array[pivotIndex];
               Swap(ref array[pivotIndex], ref array[right]);
               int storeIndex = left;
               for (int i = left; i < right; i++)
               {    if (array[i].CompareTo(pivotValue) < 0)
                    {    Swap(ref array[i], ref array[storeIndex]);storeIndex++;
                    }
               }
               Swap(ref array[storeIndex], ref array[right]);return storeIndex;
          }

          private static int GetPivotIndex<T>(int left, int right)
          {    return left + (right - left) / 2;
          }

          private static void Swap<T>(ref T a, ref T b)
          {    T temp = a;a = b; b = temp;
          }
     }
}
```

Рис.4. Клас ParallelSequentialQuickSort.

```
using System;
namespace hyperquicksort
{
       public class HyperQuickSort
       {
          public static void Sort(int[] data, int left, int right)
          {
               // Check if the data should be sorted in parallelif (right
               - left > 1000)
               {
                    // Invoke two tasks to sort the left and right halvesof the
array in parallel
                    Parallel.Invoke(
                         () => Sort(data, left, (left + right) / 2),
                         () => Sort(data, (left + right) / 2 + 1, right)
                         );
               }
               else
               {
                   // Use a traditional quicksort algorithm
                   to sort the  data Quicksort(data, left, right);
                }
          }
          static void Quicksort(int[] data, int left, int right)
          {
               // Check if the data is already sortedif
               (left >= right)
               {
                    return;
               }

               // Choose a pivot value and partition the data around it
               int pivot = data[(left + right) / 2];
               int i = left;
               int j = right;
               while (i < j)
               {    while (data[i] < pivot)
                    {    i++;
                    }
                    while (data[j] > pivot)
                    {    j--;
                    }
                    if (i <= j)
                    {    int temp = data[i];data[i] = data[j]; data[j] = temp; i++;
                         j--;
                    }
               }

               // Recursively sort the left and right halves of the data
               Quicksort(data, left, j);
               Quicksort(data, i, right);
          }
     }
}
```

Рис.5. Клас HyperQuickSort

Також існує ще більш інноваційна технологія паралельного сортування з регулярною вибіркою, ефективність кожного із цих алгоритмів буде залежати від багатьох факторів, а саме: розмір масиву, його наповнення, потужність процесора, кількість ядер та потоків тощо.

Алгоритм паралельного швидкого сортування шляхом регулярної вибірки (алгоритм на початку послідовно сортує список, а потім обирає діапазон зразків, які будуть використовуватися для подальшого розділення та заміни елементів у наступних процесах).

1. Оригінальний список поділяється на ***n*** процесів.
2. Кожен процес сортує свій підсписок за допомогою послідовного швидкого сортування.
3. Кожен процес обирає регулярні зразки зі свого відсортованого підсписку.
4. Єдиний процес збирає зразки, сортує їх і транслює обрані опорні дані іншим процесам.

5. Усі процеси використовують обрані опорні точки, щоб одночасно розділити свої підсписки на розділи відповідно до вибраних опорних елементів.
6. Процеси обмінюються відсортованими значеннями з іншими процесами-партнерами.
7. Для отримання повністю відсортованого списку об'єднуються відсортовані значення процесів.

Алгоритм паралельного швидкого сортування шляхом регулярної вибірки досягає кращий баланс навантаження, але не ідеальний. Уникає повторної заміни однакових значень, жодних накладних витрат. Кількість процесів не обов'язково має бути степенем ***2***, під час роботи алгоритму деякі процеси можуть бути звільнені залежно від порядку вибраних елементів і центрів. Початкове швидке сортування відбувається за час $\theta(\frac{n}{p} * log\frac{n}{p})$. Сортування зразків - $\theta(p^2 * logp)$. Об'єднання підмасивів відбувається за $\theta(\frac{n}{p} * logp)$. Загальна складність за часом становить ***O(nlogn).*** Складність простору дорівнює ***O(logn).*** На рис.6 надано лістинг класу ParallelQuickSortByRegularSampling.

Було створено клас Program, як програмну реалізацію дослідження алгоритмів швидкого сортування. Програма виконується в ітеративному режимі за кілька кроків: завантажити програму; у відкритому консольному вікні ввести бажану довжину масиву сортування для генерації; ввести мінімальне значення елемента масиву; ввести максимальне значення елемента масиву: із списку доступних алгоритмів сортування обрати бажаний (ввести порядкового номеру їз списку алгоритмів від 1 до 6); після вибору алгоритму сортування відбувається сортування масиву, згодом виводиться час сортування у мілісекундах (ms). та відсортований масив.

Із розглянутих паралельних алгоритмів швидкого сортування кращими повинні бути такі три алгоритми [3]: паралельне швидке сортування, гіпершвидке сортування, паралельне швидке сортування шляхом регулярної вибірки. Якщо використовувати технології OpenMP або CUDA [3, 6, 7] (з орієнтацією на C++), то найкращим вважається алгоритм паралельного швидкого сортування шляхом регулярної вибірки, прийнятним – алгоритм гіпершвидкого сортування, поганим – алгоритм паралельного швидкого сортування.

Проведено тестування алгоритмів для двох випадків сортування масивів (табл.1): малих - із ***16*** елементів, кожний із яких змінюється в межах від ***1*** до ***9;*** великих - із ***1 000 000*** елементів, кожний із яких змінюється в межах від ***1*** до ***999.***

Таблиця 1

Результати тестування алгоритмів сортування

Алгоритм сортування	Масив із ***16*** елементів, кожний із яких змінюється в межах від ***1*** до ***9*** , мс.	Масив із ***1 000 000*** елементів, кожний із яких змінюється в межах від ***1*** до ***999*** , мс.
Послідовне швидке сортування	*2*	*1278*
Наївне паралельне сортування	*8*	*950*
Оптимізоване паралельне сортування	*8*	*924*
Паралельно-послідовне сортування	*7*	*1310*
Гіпершвидке сортування	*6*	*262*
Паралельне швидке сортування шляхом регулярної вибірки	*84*	*30931*

Аналіз отриманих результатів (табл.1) для технологій з C# дає такі алгоритми: гіпершвидке сортування - кращє, оптимізоване паралельне сортування або наївне паралельне сортування - прийнятні.

```
namespace hyperquicksort
{
    public class ParallelQuickSortByRegularSampling
    {
        public void Sort(int[] arr)
        {
            Quicksort(arr, 0, arr.Length - 1);
        }

        private void Quicksort(int[] arr, int left, int right)
        {
            if (left >= right)
            {
                return;
            }

            int pivotIndex = (left + right) / 2;
            int pivotValue = arr[pivotIndex];
            int i = left;
            int j =
            right;
            while (i <= j)
            {
                while (arr[i] < pivotValue)
                {
                    i++;
                }
                while (arr[j] > pivotValue)
                {
                    j--;
                }
                if (i <= j)
                {
                    int temp =
                    arr[i];arr[i] =
                    arr[j]; arr[j] =
                    temp; i++;
                    j--;
                }
            }

            if (left < j)
            {
                    Parallel.Invoke(
                    () => Quicksort(arr, left, j),()
                    => Quicksort(arr, i, right)
                     );
             }
             else
             {
                Quicksort(arr, i, right);

             }
        }
    }
}
```

Рис.6. Клас ParallelQuickSortByRegularSampling.

Висновки

За результатами досліджень з’ясовано, що без використання технології OpenMP або CUDA алгоритм паралельного швидкого сортування шляхом регулярної вибірки не відповідає хорошим часовим показникам та програє багатьом із розглянутих алгоритмів.

Для технологій, що пов’язані з C#, найкращим із розглянутих алгоритмів є гіпершвидке сортування, а прийнятними - оптимізоване паралельне сортування або наївне паралельне сортування.

У перспективних дослідження варто розглянути окремий випадок сортування багатовимірних масивів даних. У цьому випадку до алгоритму сортування додається ще й алгоритм поділу масиву на

підсписки. Для тестування алгоритмів сортування в таких масивах є сенс створити програму з наступними можливостями: генерування багатовимірного масиву з випадковими значеннями за розміром, що задається користувачем; поділ даного масиву на підмасиви; сортування підмасивів послідовним і паралельним алгоритмом сортування; вимірювання часу на сортування N-вимірного масиву.

Також цікавим є питання, коли у деяких випадках немає змоги зберігати дані, які обробляються, у пам'яті. Тоді потрібен ефективний зовнішній алгоритм сортування з розпаралелюванням.

Література

1. D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (3rd. ed.), Addison Wesley Longman Publishing Co., Inc., 1998. - 812 p

2. С. В. Коляденко, В. О. Денисюк, Н. П. Юрчук. Дискретний аналіз. Частина1. Навчальний посібник. - Вінниця: ВНАУ, 2019. - 161с.

3. Parallel Quick Sort. URL: https://iq.opengenus.org/parallel-quicksort/

4. C# documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/

5. Task Class. URL: https://learn.microsoft.com/en-us/dotnet/ api/system.threading. tasks.task?

6. OpenMP. URL: https://www.openmp.org/

7. NVIDIA CUDA. URL: https://docs.nvidia.com/cuda/doc/index.html

References

1. D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (3rd. ed.), Addison Wesley Longman Publishing Co., Inc., 1998. - 812 p

2. S. V. Koliadenko, V. O. Denysiuk, N. P. Yurchuk. Dyskretnyi analiz. Chastyna1. Navchalnyi posibnyk. - Vinnytsia: VNAU, 2019. - 161 p.

3. Parallel Quick Sort. URL: https://iq.opengenus.org/parallel-quicksort/

4. C# documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/

5. Task Class. URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view =net-6.0

6. OpenMP. URL: https://www.openmp.org/

7. NVIDIA CUDA. URL: https://docs.nvidia.com/cuda/doc/index.html