Инженерный вестник Дона, №8 (2023)
ivdon.ru/ru/magazine/archive/n8y2023/8595

Повышение эффективности алгоритма Дейкстры с помощью технологий параллельных вычислений с библиотекой OpenMP

А.А.Аль-Саиди[1], И.О.Темкин[1], В.И. Алтай[1], А.Ф. Алмунтафеки[1], А.Н. Мохедхуссин[2]

[1]Университет науки и технологий «МИСИС», Москва
[2]Российский технологический университет «МИРЭА», Москва

Аннотация: Целью исследования является повышение эффективности алгоритма Дейкстры за счет использования модели разделяемой памяти с библиотекой OpenMP и работы по принципу параллельного выполнения при реализации алгоритма. Использование алгоритма Дейкстры для поиска кратчайшего пути между двумя узлами в графе довольно распространено. Однако временная сложность алгоритма возрастает с увеличением размера графа, что приводит к увеличению времени выполнения, поэтому параллельное выполнение является хорошим вариантом для решения проблемы временной сложности. В этой исследовательской работе предлагается метод параллельных вычислений для повышения эффективности алгоритма Дейкстры для больших графов. метод включает в себя разделение массива путей в алгоритме Дейкстры на указанное количество процессоров для параллельного выполнения. Мы предоставляем реализацию распараллеленного алгоритма Дейкстры и получаем доступ к его производительности, используя фактические наборы данных и с разным количеством узлов. Наши результаты показывают, что распараллеленный алгоритм Дейкстры может значительно ускорить процесс по сравнению с последовательной версией алгоритма, одновременно сокращая время выполнения и постоянно повышая эффективность процессора, что делает его полезным выбором для поиска кратчайших путей в больших графах.

Ключевые слова: Алгоритм Дейкстры, граф, кратчайшие пути, параллельный вычислений, модель общей памяти, библиотека OpenMP.

Введение

Эффективность алгоритма Дейкстры в контексте больших графов - тема, рассматриваемая в этой исследовательской статье. Одним из самых популярных алгоритмов кратчайшего пути является алгоритм Дейкстры, но при работе с большими графами его производительность может сильно пострадать [1]. В этом исследовании предлагается использовать библиотеку OpenMP и модель разделяемой памяти для повышения производительности алгоритма Дейкстры на графах больших размеров. Это исследование направлено на повышение производительности алгоритма Дейкстры при

работе с большими графами за счет использования модели общей памяти и библиотеки OpenMP.

Гипотеза данного исследования заключается в том, что алгоритм Дейкстры будет работать лучше при работе с большими графами, если использовать модель общей памяти и библиотеку OpenMP. Будет принят во внимание традиционный алгоритм Дейкстры и его модификация для модели с разделяемой памятью и библиотеки OpenMP. На реальных графиках будут проведены эксперименты, чтобы оценить, насколько хорошо работает алгоритм Дейкстры и его модификация.

Новизна работы заключается в использовании новой методологии при разработке параллельного алгоритма Дейкстры; эта методология характеризуется простотой и устранением сложности при реализации параллельных алгоритмов за счет сосредоточения внимания только на параллельном выполнении циклов. Эта методология никогда ранее не использовалась при реализации параллельного алгоритма Дейкстры.

Для оценки эффективности алгоритма Дейкстры и его модификации будут использоваться следующие критерии:

• Время выполнения алгоритмов на графиках реального мира.

• Ускорение для измерения производительности процессора и определения влияния реализации параллельного алгоритма Дейкстры на производительность процессора.

• эффективность: для измерения эффективности алгоритма Дейкстры.

Ожидается, что результаты этого исследования будут иметь практическое значение для области вычислительной математики и оптимизации алгоритмов, конкретные результаты будут описаны после проведения экспериментов и анализа полученных данных.

Последовательный алгоритм Дейкстры

Алгоритм Дейкстры - это классический алгоритм нахождения кратчайшего пути в графе. Он широко используется во многих приложениях, таких как сетевая маршрутизация, транспортное планирование и географические информационные системы [1]. Однако для больших графов время вычисления алгоритма Дейкстры может быть непомерно большим. Это связано с тем, что временная сложность алгоритма равна O(V|2), где V - количество узлов в графе. По мере увеличения размера графика время выполнения алгоритма также увеличивается, что делает его непрактичным для больших графов [2, 3].

Для заданного исходного узла s в графе с N узлами алгоритм Дейкстры находит кратчайший путь между узлом s и каждым другим узлом. Предположим, что w [u, v] это расстояние между узлами u и v, и оно хранится в массиве(dest[]). Он может быть описан следующим образом:

Алгоритм 1:

(1) Инициализация: установите исходное расстояние dist [s] равным 0, остальные расстояния - бесконечности. Пометить статус всех узлов как непосещаемые.

(2) Цикл N-1 раз: найти узел u с минимальным расстоянием в наборе не посещённых узлов и пометить его состояние как посещенный. Это представляется как FindMinimumU(). Для каждого соседа v узла u выполните Relax(u, v). Если dist [u] + w [u, v] <dist [v], то обновите значение dist[v] на более короткое расстояние dist [u] + w [u, v]. Узел u является предыдущим узлом узла v на кратчайшем пути.

(3) Конец. Для любого узла u, dist [u] — это расстояние между s и u.

Параллельные вычисления

Методы параллельных вычислений показали свою эффективность в ускорении выполнения алгоритмов на больших наборах данных [4]. В этой

научной статье мы предлагаем метод параллельных вычислений для повышения эффективности алгоритма Дейкстры для больших графов. Наш подход предполагает разделение графа на более мелкие подграфы и распараллеливание выполнения алгоритма Дейкстры на каждом подграфе с использованием подхода распараллеливания с общей памятью. Мы представляем экспериментальное исследование, сравнивающее производительность последовательного алгоритма Дейкстры и параллельного алгоритма Дейкстры с использованием OpenMP. OpenMP — это широко используемая техника параллельных вычислений для систем с общей памятью. Существуют различные типы моделей параллельных вычислений, включая модели с общей памятью, распределенной памятью и CUDA [5] .

Существуют различные модели параллельных вычислений, которые включают в себя:

- Модель общей памяти: в этой модели несколько процессоров совместно используют общее пространство памяти и могут получать доступ к данным в этом пространстве и изменять их. Обычно он используется в многоядерных процессорах и системах симметричной многопроцессорной обработки (SMP) [5, 6].
- Модель распределенной памяти: в этой модели несколько процессоров связаны через сеть и имеют свое собственное пространство частной памяти. Связь между процессорами происходит посредством передачи сообщений, при которой данные отправляются и принимаются между процессами. Эта модель широко используется в кластерах и суперкомпьютерах [7, 8].
- Модель графического процессора: В этой модели для выполнения программы используется графический процессор (GPU), который оптимизирован для параллельной обработки. Графические процессоры

обычно имеют сотни или тысячи процессорных ядер, которые могут выполнять программные инструкции одновременно [5, 9].

- Гибридная модель: Эта модель сочетает в себе модели общей памяти и распределенной памяти, чтобы использовать преимущества обеих. Он широко используется в системах высокопроизводительных вычислений (HPC) [7].

Модель с общей памятью - это модель параллельных вычислений, в которой несколько процессоров или ядер совместно используют общее пространство памяти. В этой модели все процессоры могут получать доступ и изменять данные в общем пространстве памяти, что позволяет им совместно работать над задачей или алгоритмом. Модель общей памяти обычно используется в многоядерных процессорах, симметричных многопроцессорных системах (SMP) и некоторых библиотеках параллельного программирования, таких как OpenMP. В модели общей памяти каждый процессор имеет свой собственный кэш для хранения часто используемых данных, что сокращает время доступа к памяти и повышает производительность. Однако необходимо следить за тем, чтобы избежать ситуаций гонки, когда несколько процессоров пытаются одновременно получить доступ и изменить одни и те же данные, что приводит к неправильным результатам, на рис.1 показана архитектура модели общей памяти.

Для реализации модели общей памяти программу или алгоритм необходимо разделить на более мелкие задачи, которые могут выполняться параллельно несколькими процессорами. Задачи должны быть разработаны таким образом, чтобы минимизировать конфликты между процессорами, обращающимися к пространству общей памяти, а для предотвращения гоночных условий следует использовать механизмы синхронизации, такие, как блокировки или семафоры. Модель общей памяти может быть

использована для повышения производительности многих алгоритмов и приложений, включая матричное умножение, обработку изображений и параллельную сортировку. Она часто используется в сочетании с другими моделями параллельных вычислений, такими, как распределенная память или параллелизм задач, чтобы использовать их преимущества и преодолеть их недостатки [6, 7].

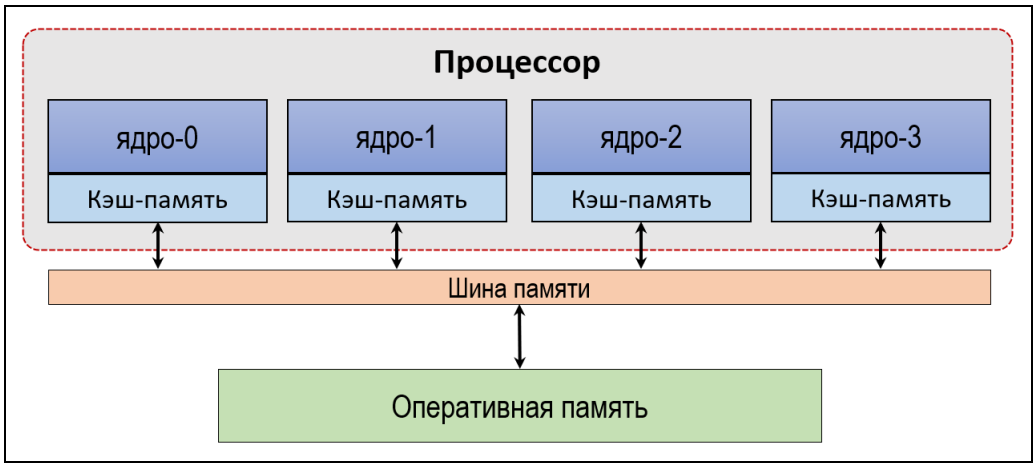

Рис. 1. – Архитектура модели общей памяти

Ускорение и эффективность - два важных показателя производительности в параллельных вычислениях. Вот как они могут быть рассчитаны:

Ускорение: Ускорение - это отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма, решающего ту же задачу. Оно показывает, насколько быстрее работает параллельный алгоритм по сравнению с последовательным. Формула для ускорения такова:

$$\text{Ускорение (S)} = T\_seq / T\_par, \quad (1)$$

где T_seq - время выполнения последовательного алгоритма; T_par - время выполнения параллельного алгоритма.

Эффективность: Эффективность измеряет, насколько эффективно используются доступные ресурсы в параллельном алгоритме. Она представляет собой отношение ускорения, достигнутого при использовании

нескольких процессоров, к количеству используемых процессоров. Формула для эффективности выглядит следующим образом:

$$\text{Эффективность (E)} = S / P , \quad (2)$$

где S - Ускорение; P - Количество процессоров.

Важно отметить, что достижение высокого ускорения не обязательно означает высокую эффективность. Параллельный алгоритм может достичь высокого ускорения, но иметь низкую эффективность, если ресурсы используются неэффективно. Аналогично, параллельный алгоритм может иметь высокую эффективность, но низкое ускорение, если размер задачи недостаточно велик, чтобы извлечь выгоду из параллелизма [10].

Библиотека OpenMP

OpenMP (Open Multi-Processing) - это библиотека для программирования мультипроцессинга с общей памятью на языках C, C++ и Fortran. Она предлагает директивы компилятора, процедуры библиотеки времени выполнения и переменные среды для параллельного программирования на архитектурах с общей памятью, таких как многоядерные процессоры и SMP-системы. OpenMP использует модель fork-join, поддерживая такие функции, как создание потоков, синхронизация, совместное использование данных, распараллеливание циклов, распараллеливание задач и распараллеливание SIMD. Он широко используется в научных вычислениях и аналитике данных для высокопроизводительной параллельной обработки, на рис. 2 показана модель Fork-Join для распараллеливания OpenMP.

Библиотека OpenMP включает множество функций, которые могут быть использованы для управления параллелизмом и синхронизацией в программе.

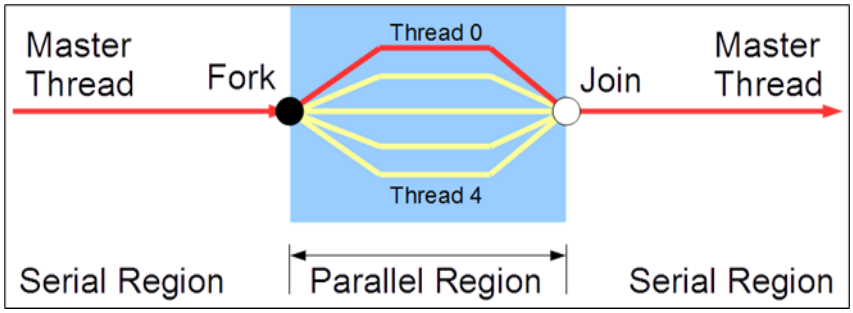

Рис. 2. – Модель Fork-Join для распараллеливания OpenMP

Некоторые из наиболее часто используемых функций включают:

omp_get_num_threads(): Возвращает количество потоков, выполняющихся в настоящее время в параллельном регионе.

omp_get_thread_num(): Возвращает номер потока вызывающего потока в параллельном регионе.

omp_set_num_threads(): Устанавливает количество потоков, которые будут использоваться в последующих параллельных регионах.

omp_get_wtime(): Возвращает время настенных часов в секундах.

omp_barrier(): Синхронизирует все потоки в параллельном регионе, чтобы они достигли одной и той же точки в коде перед продолжением работы.

omp_parallel(): Помечает участок кода как параллельную область, указывая, что заключенный в ней код должен выполняться параллельно несколькими потоками.

omp_for(): Распределяет итерации цикла между несколькими потоками, позволяя каждому потоку выполнять подмножество итераций.

Эти и подобные им функции предоставляют мощный набор инструментов для разработки параллельных приложений с использованием OpenMP. Используя возможности библиотеки OpenMP, разработчики могут создавать эффективные и масштабируемые параллельные программы, которые могут использовать всю мощь современных многоядерных процессоров [11, 12].

Реализация параллельного алгоритма Дейкстры

Параллельный алгоритм Дейкстры фокусируется на параллельном выполнении циклов, где циклы обработки матрицы путей графа делятся на количество выбранных потоков или ядер, как описано в Алгоритме 2:

Алгоритм 2:

```
Parallel Algorithm Dijkstra (Graph, source, NUM_THREADS):
// Начало первой параллельной области
NUM_PROCS = omp_get_num_procs()
NUM_THREADS = NUM_PROCS
#Pragma omp parallel for num_threads(NUM_THREADS) {
  // Инициализируем расстояния до всех узлов как бесконечность, кроме исходного узла.
  distances = map infinity to all nodes
  distances = 0
} // Конец первой параллельной области
  // Инициализируем пустой набор посещенных узлов и очередь приоритетов для отслеживания узлов,
   //которые необходимо посетить.
  visited = empty set
  queue = new PriorityQueue()
  queue.enqueue(source, 0)
// Начало второй параллельной области
#Pragma omp parallel for num_threads(NUM_THREADS) {
// Зацикливаемся, пока все узлы не будут посещены.
  while queue is not empty:
     // Удаление узла с наименьшим расстоянием из приоритетной очереди.
     current = queue.dequeue()
     // Если узел уже был посещен, пропустите его.
     if current in visited:
        continue
     // Пометить узел как посещенный.
     visited.add(current)
     // Проверьте все соседние узлы на предмет необходимости обновления их расстояний.
     for neighbor in Graph.neighbors(current):
        // Вычислить предварительное расстояние до соседа через текущий узел.
        tentative_distance = distances[current] + Graph.distance(current, neighbor)
       // Если предварительное расстояние меньше, чем текущее расстояние до соседа, обновите
       //расстояние.
        if tentative_distance < distances[neighbor]:
           distances[neighbor] = tentative_distance
            queue.enqueue(neighbor, distances[neighbor])
} // Конец второй параллельной области
  // Возвращаем вычисленные расстояния от источника до всех остальных узлов графа.
 return distances
```

Рассчитать временную сложность параллельного алгоритма Дейкстры. Для получения таблицы маршрутизации нам потребуется O(V) раундов итераций (пока все вершины не будут включены в кластер). В каждом раунде

мы будем обновлять значение для O(V) вершин, используя P ядер, работающих независимо, и использовать параллельный префикс для выбора глобальной ближайшей вершины, поэтому время работы в каждом раунде составляет O(V/P) + O(log(P)). Таким образом, общее время работы составляет:

$$T_P = O\left(\frac{V^2}{P} + V.\ \log(P)\right), \quad (3)$$

где P - количество используемых ядер; V - количество вершин.

Материалы и методы

Мы оцениваем эффективность параллельного алгоритма Дейкстры для нахождения всех коротких путей от исходного узла ко всем узлам, сравнивая параллельное выполнение с последовательным выполнением алгоритма. В данной работе реализован параллельный алгоритм Дейкстры для нахождения всех кратчайших путей между исходным узлом и всеми узлами. Мы реализовали параллельный алгоритм Дейкстры на различном количестве потоков от 1 потока до 8 потоков для выполнения на равном количестве процессоров или ядер. Мы протестировали нашу реализацию на настольном компьютере с процессором Intel (R) Core (TM) i7-8550U с частотой 1,80 ГГц, 8 ядрами, 8 потоками и 16 ГБ оперативной памяти. В качестве языка программирования использовался C++ с библиотекой OpenMP под Microsoft Visual Basic 2022 в среде операционной системы Windows 11 с 64-битной архитектурой. Для оценки производительности алгоритмов. Принцип работы модели общей памяти заключается в том, что алгоритм начинает выполняться последовательно с основным потоком, называемым ведущим потоком, а когда выполнение достигает параллельной области, начинается параллельное выполнение с определенным количеством потоков с помощью переменной (num_threads). Идея последовательного выполнения параллельного алгоритма заключается в том, чтобы сделать значение

переменной (num_threads) равным единице, и таким образом параллельная область выполняется с использованием только одного потока. Параллельный выполняется с количеством потоков в диапазоне от 2 до 8. Число 8 потоков - это максимальное число потоков, равное количеству используемых ядер процессора, где каждый поток выполняется на одном ядре одновременно, после выполнения кода мы можем получить результат, который представляет собой продолжительность в секундах.

Результаты и обсуждение

В Таблице № 1 содержится наилучшее время для последовательного выполнения алгоритма Дейкстры (один поток) для каждого количества потоков и параллельно с библиотеками OpenMP, отмечено, что результаты были выбраны в несколько раз лучше, чем выполнение кода. Код выполняется в одной и той же системной среде, а источник входных данных генерируется функцией Random. В данной работе для входной матрицы смежности используется в общей сложности три набора данных. То есть, для графа используются 500 вершин, 750 вершин и 1000 вершин. После выполнения кода, мы можем получить результат, который представляет собой продолжительность в секундах. Для параллельных вычислений OpenMP для выполнения кода используется 1, 2, 3,4,5,6,7 and 8 процессоров (количество вершин должно быть больше количества процессоров).

Таблица № 1

Показывает время выполнения в зависимости от количества потоков/ядер

Количество вершин	Количество потоков /ядер							
	1	2	3	4	5	6	7	8
500	165,5	146,4	91,4	74,0	62,6	52,2	51,7	50,2
750	380,2	251,7	212,4	159,2	132,2	115,0	112,9	110,4
1000	1510,3	1340,7	866,56	654,0	544,7	461,9	421,4	375,5

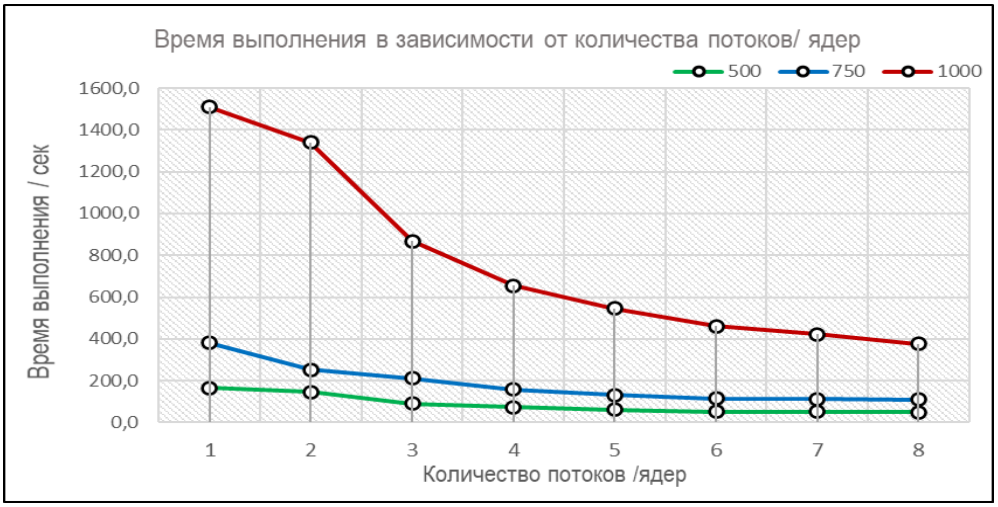

Рис. 3. – Время выполнения в зависимости от количества потоков /ядер

Мы можем вычислить ускорение и эффективность в каждом условии. В таблице № 2 показано значение ускорения для каждого случая, а ускорение для всех случаев можно увидеть на рис. 4.

Таблица № 2

Показывает ускорение для каждого количества потоков или ядер

Количество вершин	Количество потоков /ядер							
	1	2	3	4	5	6	7	8
500	1,00	1,13	1,81	2,24	2,64	3,17	3,20	3,30
750	1,00	1,51	1,79	2,39	2,88	3,31	3,37	3,44
1000	1,00	1,13	1,74	2,31	2,77	3,27	3,58	4,02

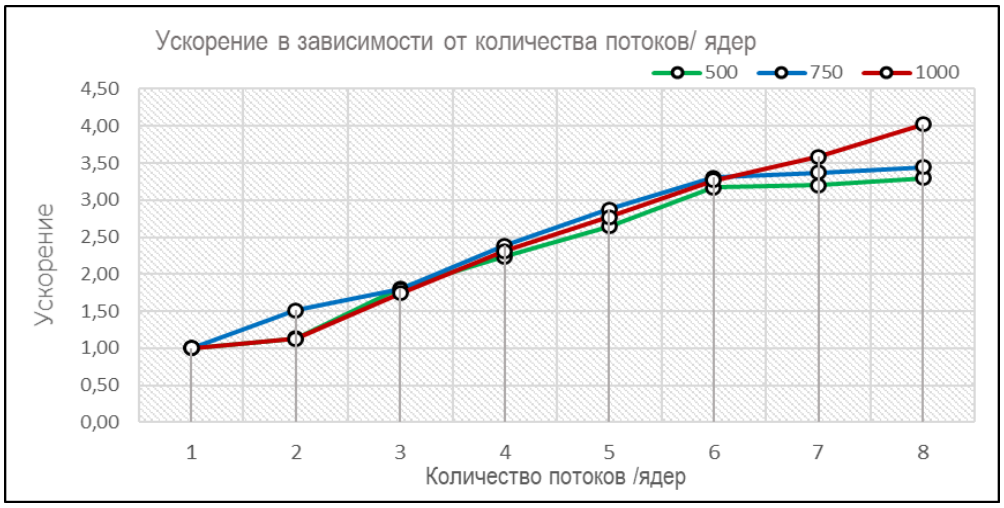

Рис. 4. – Показывает ускорение в зависимости от количества потоков /ядер

Таблица № 3

Показывает эффективность для каждого количества потоков или ядер

Количество вершин	Количество потоков /ядер							
	1	2	3	4	5	6	7	8
500	1,00	0,57	0,60	0,56	0,53	0,53	0,46	0,41
750	1,00	0,76	0,60	0,60	0,58	0,55	0,48	0,43
1000	1,00	0,56	0,58	0,58	0,55	0,54	0,51	0,50

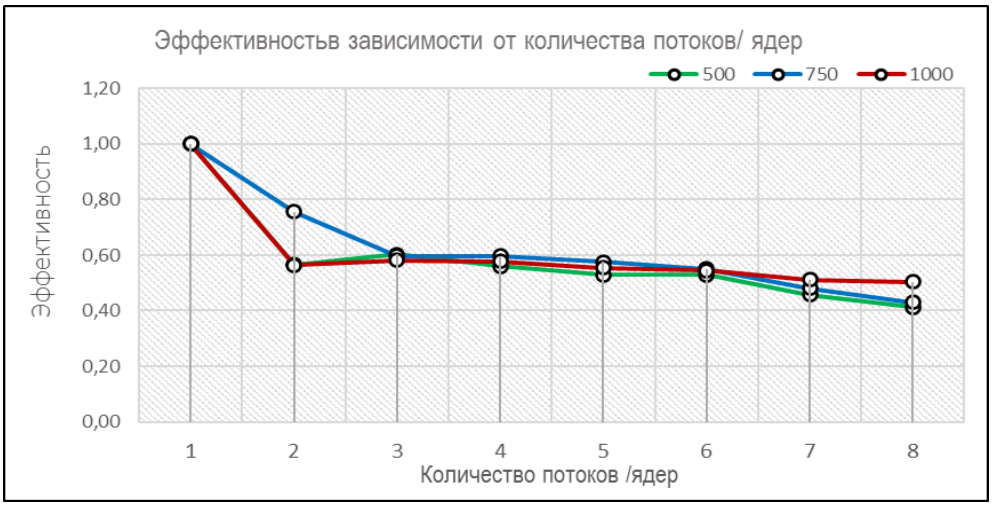

Рис. 5. – Показывает эффективность в зависимости от количества потоков

При реализации параллельного алгоритма Дейкстры с использованием модели разделяемой памяти и OpenMP и проведении экспериментов с использованием больших графов. Было измерено время выполнения последовательной и параллельной версий алгоритма и проведено сравнение их производительности.

Результаты эксперимента, приведенные в таблице №1, показали, что параллельная версия алгоритма, использующая общую память и OpenMP, превзошла последовательную версию для больших графов. Сравнивая результаты, мы обнаруживаем, что время параллельного выполнения двух потоков на двух ядрах сокращается примерно на (22%, 44%, 22%) от времени последовательного выполнения когда количество узлов равно (500, 750,

1000) соответственно и на (70%, 72%, 76%) при выполнении восемью потоками на восьми ядрах когда количество узлов равно (500, 750, 1000) соответственно, как показано на рис. 3.

Результаты, приведенные в таблице № 2, показали достижение повышенной производительности процессора в целом, но во всех случаях в разных пропорциях, как показано на рис. 4, и это считается хорошим показателем полной производительности процессора.

В таблице № 3 и на рис. 5, результаты показывают эффективность алгоритма для всех случаев, и можно наблюдать изменение эффективности алгоритма по мере увеличения ускорения процессора. Достижение наивысшей эффективности алгоритма и высочайшей производительности процессора может быть достигнуто при использовании принципа параллельных вычислений только в идеальных случаях. Согласно полученным результатам, эксперимент соответствует требованиям к времени выполнения алгоритма и эффективности, а также приемлемому уровню производительности процессора. Количество процессорных ядер напрямую влияет на то, насколько быстро выполняется алгоритм. Тип ядер, количество ядер, другие запущенные приложения, фоновые процессы и т.д. - все это может повлиять на скорость выполнения кода.

Заключение

В заключение, использование модели разделяемой памяти с библиотекой OpenMP может значительно повысить эффективность алгоритма Дейкстры для поиска кратчайшего пути в больших графах. Несколько исследований продемонстрировали эффективность OpenMP при распараллеливании алгоритма Дейкстры, и было показано, что повышение производительности зависит от размера графика и количества используемых ядер. Модель общей памяти OpenMP позволяет нескольким потокам получать доступ к общему пространству памяти, что снижает накладные

расходы на обмен данными между потоками и повышает производительность алгоритма. Кроме того, OpenMP предоставляет простой и гибкий программный интерфейс, который позволяет легко реализовать распараллеленный алгоритм Дейкстры. В целом, использование OpenMP может сделать алгоритм Дейкстры более эффективным и практичным для реальных приложений с большими графами.

Литература (References)

1. Jasika N., Alispahic N., Elma A., Ilvana K., Elma L., and Nosovic N., Dijkstra's shortest path algorithm serial and parallel execution performance analysis, in 2012 proceedings of the 35th international convention MIPRO, IEEE, 2012, pp. 1811-1815.

2. Gunawan R. D., Napianto R., Borman R. I., and Hanifah I., Implementation Of Dijkstra's Algorithm In Determining The Shortest Path (Case Study: Specialist Doctor Search In Bandar Lampung), Int. J. Inf. Syst. Comput. Sci, vol. 3, no. 3, pp. 98-106, 2019.

3. Wayahdi M. R., Ginting S. H. N., and Syahputra D., Greedy, A-Star, and Dijkstra's algorithms in finding shortest path, International Journal of Advances in Data and Information Systems, vol. 2, vol. 1, pp. 45-52, 2021.

4. Anzt H., Huckle T. K., Bräckle J., Dongarra and J., Incomplete sparse approximate inverses for parallel preconditioning, Parallel Comput, vol. 71, pp. 1-22, 2018.

5. Eichstadt J., Vymazal M., Moxey D., Peyroux and J., A comparison of the shared-memory parallel programming models OpenMP, OpenACC and Kokkos in the context of implicit solvers for high-order FEM, Comput Phys Commun, 2020, vol. 255, c. 107245.

6. Al-Saeedi A. A. K., Yurievna S. O., and Khairi T. W., Improving the efficiency of sparse matrix class processing by using the SPM-CSR parallel algorithm and OpenMP technology, in 2022 4th International Youth Conference on

Radio Electronics, Electrical and Power Engineering (REEPE), IEEE, 2022, pp. 1-4.

7. Martineau M., McIntosh-Smith S., and Gaudin W., Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model, in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2016, pp. 338-347.

8. Murni, A., Bustamamam Ernastuti, Handhika T., and Kerami D., Hypergraph partitioning implementation for parallelizing matrix-vector multiplication using CUDA GPU-based parallel computing, in AIP Conference Proceedings, AIP Publishing LLC, 2017, pp. 030153.

9. Janssen D. M., Pullan W., and Liew A. W., Graphics processing unit acceleration of the island model genetic algorithm using the CUDA programming platform, Concurr Comput, 2022, vol. 34, no 2, pp. e6286.

10. Buzachis A., Celesti A., Galletta A., Wan J., and Fazio M., Evaluating an Application Aware Distributed Dijkstra Shortest Path Algorithm in Hybrid Cloud/Edge Environments, IEEE Transactions on Sustainable Computing, 2022, vol. 7, issue 2, pp. 289-298, doi: 10.1109/TSUSC.2021.3071476.

11. L. Huang, Wang L., Shao J., Liu X., Hao Q., Xing L., Zheng Y., and Xiao Y. Parallel processing transport model MT3DMS by using OpenMP, Int J Environ Res Public Health, 2018, vol. 15, issue 6, p. 1063, 2018.

12. Awari R., Parallelization of shortest path algorithm using OpenMP and MPI, in 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2017, pp. 304-309. doi: 10.1109/I-SMAC.2017.8058360.