ФГБОУ ВПО «Национальный исследовательский университет «МЭИ»

Институт информатики и вычислительной техники

Кафедра прикладной математики и искусственного интеллекта

**Отчет по комплекту лабораторных работ**

по курсу

«Параллельное программирование и параллельные системы»

**Выполнила:**

Кравченко К.С.

Группа: А-05м-21

**Проверил:**

Кутепов В.П.

Москва, 2022

1. **Постановка задачи**

Требуется найти решение системы m линейных уравнений, которая записывается в общем виде как

,

Эту систему уравнений можно записать также в матричном виде:

где , , .

A – матрица системы, – вектор правых частей, – вектор неизвестных.

При известных A и требуется найти такие , при подстановке которых в систему уравнений она превращается в тождество.

1. **Описание метода.**

Для решения поставленной задачи будем использовать метод верхней релаксации.

Приведем систему к виду

Обозначим

В векторно-матричном виде система запишется в виде:

*x=Bx+c*

Для метода верхней релаксации введем параметр ω > 1. Тогда итерационная формула метода верхней релаксации будет иметь следующий вид:

1. **Описание инструментария**

**Лабораторная работа № 1. Императивный язык.**

Для выполнения первой лабораторной работы был выбран язык C#. Для решения поставленной задачи использовались методы Parallel.For и Parallel.ForEach библиотеки System.Threading.Tasks.

**Лабораторная работа № 2. Императивный язык с использованием программного интерфейса MPI.**

MPI – это программный инструментарий для обеспечения связи между ветвями параллельного приложения. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные / многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных). Минимально в состав MPI входят: библиотека программирования и загрузчик приложений.

Базовым механизмом связи между MPI процессами является передача и приём сообщений. Сообщение несёт в себе передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём:

● отправитель — ранг (номер в группе) отправителя сообщения;

● получатель — ранг получателя;

● признак — может использоваться для разделения различных видов сообщений;

● коммуникатор — код группы процессов.

Операции приёма и передачи могут быть блокирующимися и неблокирующимися. Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения операции.

Другим способом связи является удалённый доступ к памяти (RMA), позволяющий читать и изменять область памяти удалённого процесса. Локальный процесс может переносить область памяти удалённого процесса (внутри указанного процессами окна) в свою память и обратно, а также комбинировать данные, передаваемые в удалённый процесс с имеющимися в его памяти данными (например, путём суммирования). Все операции удалённого доступа к памяти не блокирующиеся, однако, до и после их выполнения необходимо вызывать блокирующиеся функции синхронизации.

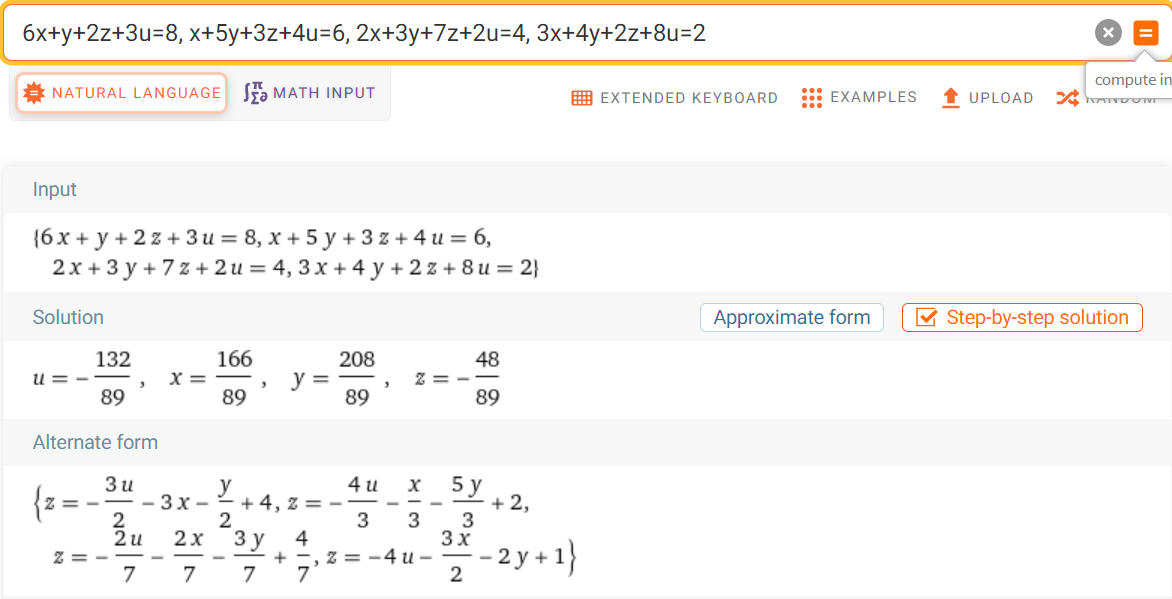
При выполнении второй лабораторной работы была использована реализация MPI для языка Python.

**Лабораторная работа №3. Функциональный язык**

Для выполнения третьей лабораторной работы был выбран функциональный язык Haskell.

1. **Подтверждение корректности работы**

Проверим корректность работы разработанных программ на матрице размерности 4х4 с помощью WolframAlpha.



*Рис. 4.1. Решение системы с помощью WolframAlpha*

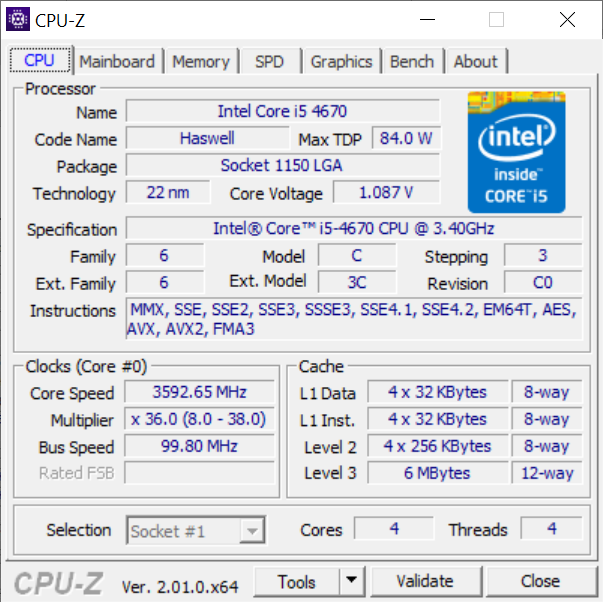
После выполнения реализаций всех программ, получаем следующее решение:

[1.8651684058282514,2.337078187182826,-0.5393257765249797,-1.4831458319527844]

Таким образом, можно сделать вывод о корректной работе всех реализаций.

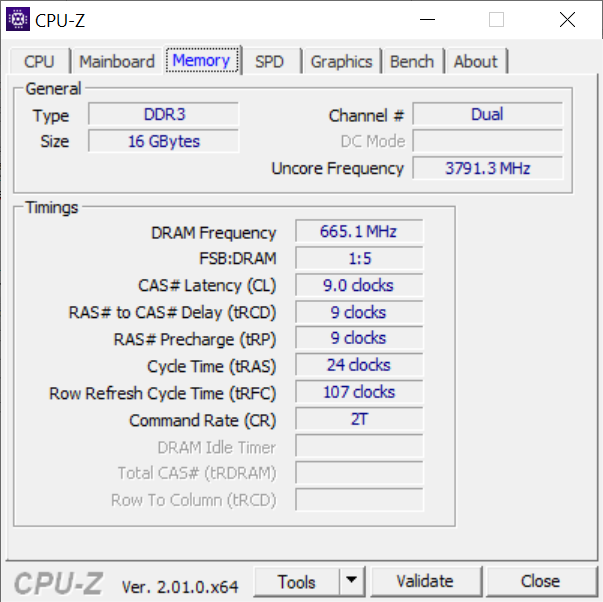
1. **Характеристики компьютера**

**Процессор:**



*Рис.5.1. Сведения о процессоре*

**Оперативная память:**



*Рис.5.2. Сведения об оперативной памяти*

**Операционная система:**

Название: Windows 10 Pro

Разрядность: 64

Версия: 21H2

Сборка: 19044.1466

1. **Описание набора экспериментальных данных.**

С целью формирования симметричной положительно определенной матрицы элементы ниже главной диагонали генерировались в диапазоне от 0 до 1, значения элементов выше главной диагонали получались симметрией элементов ниже главной диагонали, а элементы на главной диагонали генерировались в диапазоне от n до 2n, где n – размер матрицы.

В качестве набора данных использовались:

* СЛАУ с 500 неизвестными и 500 уравнениями.
* СЛАУ с 5000 неизвестными и 5000 уравнениями.

1. **Экспериментальное исследование.**

* **Зависимость времени выполнения Т от степени зернистости N**

Результаты экспериментов на малом наборе данных.

*Таблица 9.1.* *Зависимость T от N на малых данных*

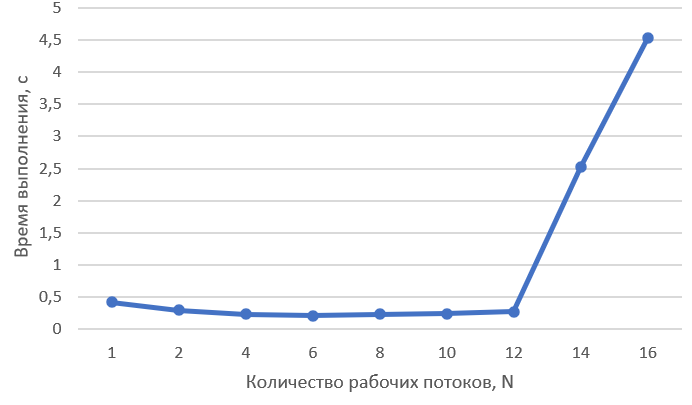
|  |  |  |  |
| --- | --- | --- | --- |
|  | T, сек | | |
| N | C# | Haskell | MPI |
| 1 | 0,462 | 1,998 | 0,417 |
| 2 | 2,01 | 1,953 | 0,293 |
| 4 | 5,413 | 2,041 | 0,232 |
| 6 | 3,418 | 1,894 | 0,209 |
| 8 | 4,777 | 1,323 | 0,235 |
| 10 | 2,008 | 1,456 | 0,237 |
| 12 | 1,966 | 1,242 | 0,271 |
| 14 | 2,937 | 1,725 | 2,521 |
| 16 | 2,371 | 1,628 | 4,528 |



*Рис. 9.1. Зависимость T от N для C# на малых данных*



*Рис. 9.2. Зависимость T от N для Haskell на малых данных*

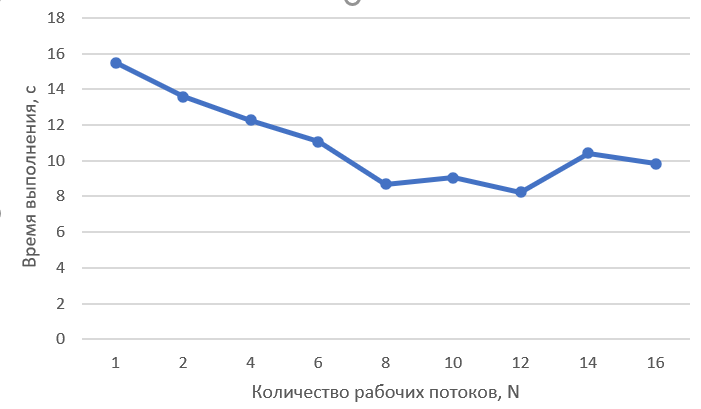


*Рис. 9.3. Зависимость T от N для MPI на малых данных*

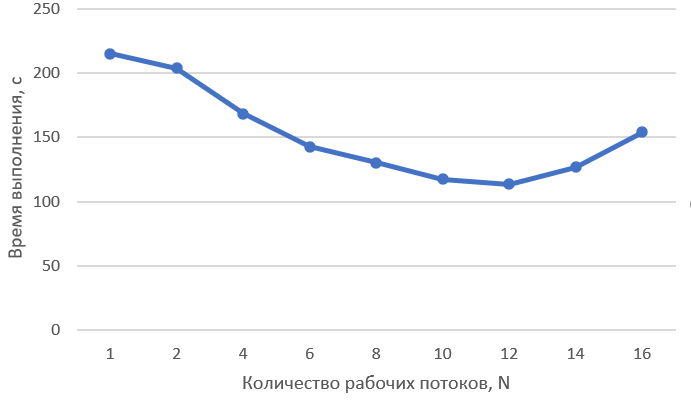
Результаты экспериментов на большом наборе данных.

*Таблица 9.2.* *Зависимость T от N на больших данных*

|  |  |  |  |
| --- | --- | --- | --- |
|  | T, сек | | |
| N | C# | Haskell | MPI |
| 1 | 15,49 | 215,21 | 41,76 |
| 2 | 13,58 | 203,859 | 29,15 |
| 4 | 12,26 | 168,495 | 19,419 |
| 6 | 11,07 | 142,938 | 18,293 |
| 8 | 8,69 | 130,204 | 19,594 |
| 10 | 9,05 | 117,351 | 20,284 |
| 12 | 8,241 | 113,429 | 24,592 |
| 14 | 10,43 | 126,748 | 32,313 |
| 16 | 9,84 | 154,192 | 42,752 |



*Рис. 9.4. Зависимость T от N для C# на больших данных*



*Рис. 9.5. Зависимость T от N для Haskell на больших данных*



*Рис. 9.6. Зависимость T от N для MPI на больших данных*

**Выводы**

Из полученных данных видно, что:

* На C# на малых данных параллельная реализация не дала выигрыша во времени. Это произошло, т.к. накладные расходы на создание, синхронизацию и передачу задачи данных в потоки вызывают большие потери производительности, чем выигрыш от распараллеливания малого количества простых операций (в методе верхней релаксации мы можем распараллелить только вычисление сумм).

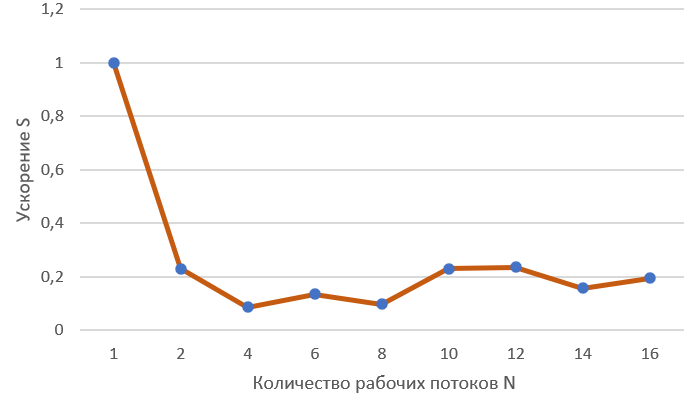
На больших данных наименьшее время достигается при 12 потоках.

* На Haskell наименьшее время как на малых, так и на больших данных достигается при 12 потоках.
* На Python (MPI) наименьшее время как на малых, так и на больших данных достигается при 6 потоках.
* При увеличении размерности задачи время выполнения всех реализаций, как и ожидалось, сильно увеличивается.
* **Зависимость ускорения S в зависимости от количества рабочих потоков N**

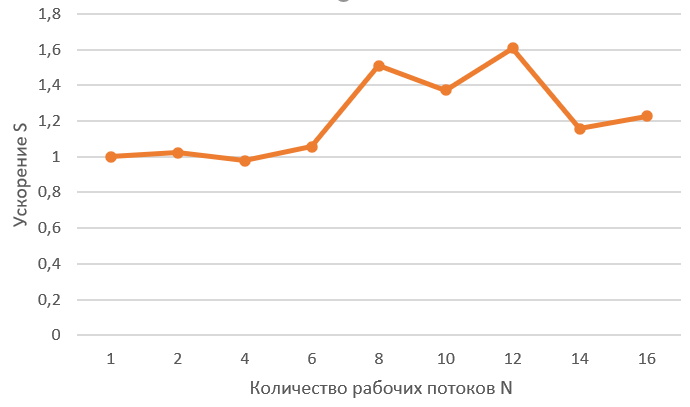
Результаты экспериментов на малом наборе данных.

*Таблица 9.3.* *Зависимость S от N на малых данных*

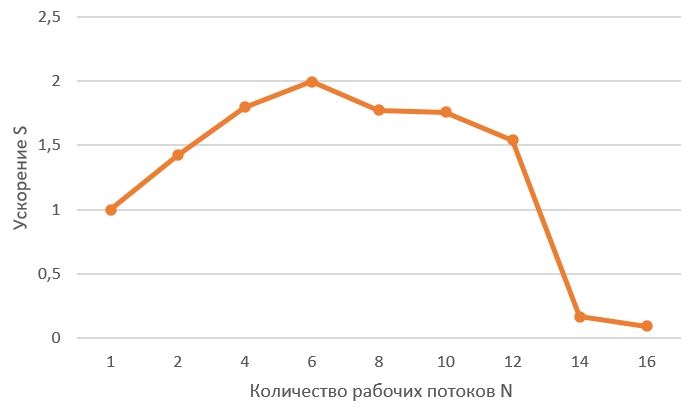
|  |  |  |  |
| --- | --- | --- | --- |
|  | S | | |
| N | C# | Haskell | MPI |
| 1 | 1 | 1 | 1 |
| 2 | 0,229850746 | 1,023041 | 1,423208191 |
| 4 | 0,085350083 | 0,978932 | 1,797413793 |
| 6 | 0,135166764 | 1,05491 | 1,995215311 |
| 8 | 0,096713418 | 1,510204 | 1,774468085 |
| 10 | 0,230079681 | 1,372253 | 1,759493671 |
| 12 | 0,234994914 | 1,608696 | 1,538745387 |
| 14 | 0,157303371 | 1,158261 | 0,165410551 |
| 16 | 0,194854492 | 1,227273 | 0,09209364 |



*Рис.9.7. Зависимость S от N для C# на малых данных*



*Рис.9.8. Зависимость S от N для Haskell на малых данных*

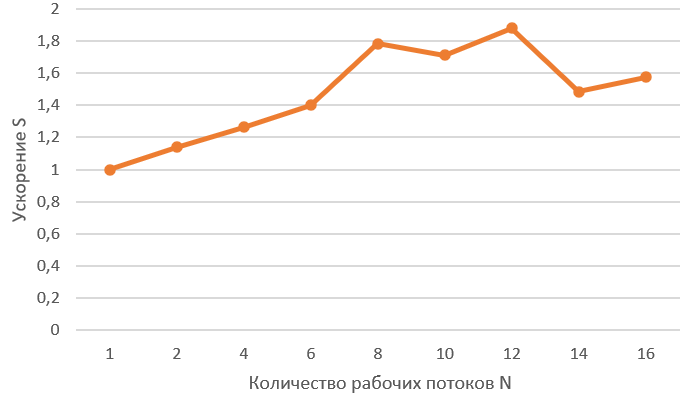


*Рис.9.9. Зависимость S от N для MPI на малых данных*

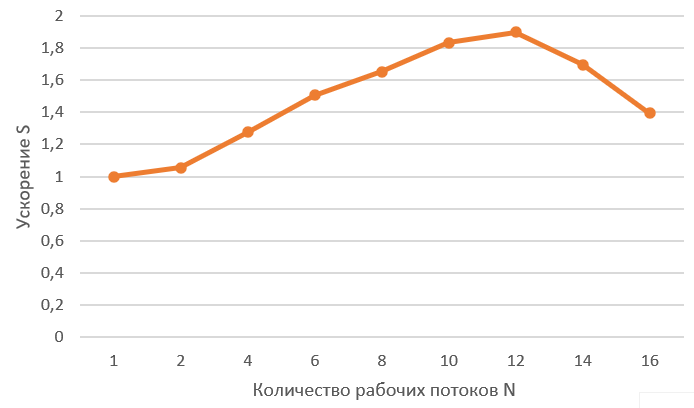
Результаты экспериментов на большом наборе данных.

*Таблица 9.4.* *Зависимость S от N на больших данных*

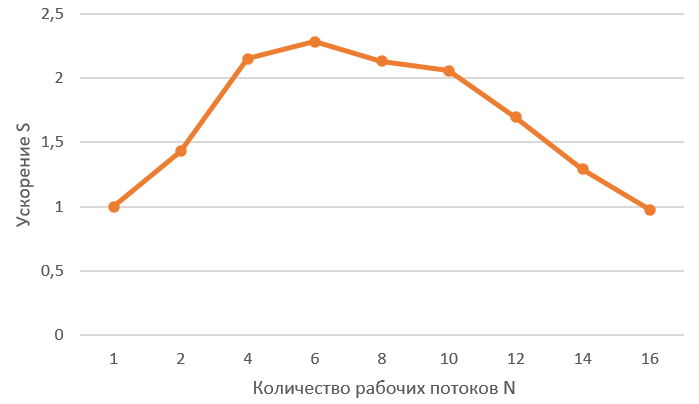
|  |  |  |  |
| --- | --- | --- | --- |
|  | S | | |
| N | C# | Haskell | MPI |
| 1 | 1 | 1 | 1 |
| 2 | 1,140648 | 1,055681 | 1,43259 |
| 4 | 1,263458 | 1,277249 | 2,150471 |
| 6 | 1,399277 | 1,505618 | 2,28284 |
| 8 | 1,782509 | 1,652868 | 2,131265 |
| 10 | 1,711602 | 1,8339 | 2,058766 |
| 12 | 1,879626 | 1,89731 | 1,698113 |
| 14 | 1,485139 | 1,697936 | 1,292359 |
| 16 | 1,574187 | 1,395727 | 0,976796 |



*Рис.9.10. Зависимость S от N для C# на больших данных*



*Рис.9.11. Зависимость S от N для Haskell на больших данных*



*Рис.9.12. Зависимость S от N для MPI на больших данных*

**Выводы**

Из полученных данных видно, что:

* На C# на малых данных параллельная реализация не дала никакого ускорения. Это произошло, т.к. накладные расходы на создание, синхронизацию и передачу задачи данных в потоки вызывают большие потери производительности, чем выигрыш от распараллеливания малого количества простых операций (в методе верхней релаксации мы можем распараллелить только вычисление сумм).

На больших данных наибольшее ускорение было достигнуто при 12 потоках.

* На Haskell наибольшее ускорение как на малых, так и на больших данных достигается при 12 потоках.
* На Python (MPI) наибольшее ускорение как на малых, так и на больших данных достигается при 6 потоках.
* Для всех реализаций с увеличением количества рабочих потоков сначала наблюдается увеличение ускорения, но затем всегда следует спад.
* Для всех реализаций на больших данных было получено большее ускорение, чем на малых данных.

1. **Общие выводы**

В ходе выполнения лабораторных работ были разработаны 3 программы решения СЛАУ методом релаксации: на C#, на Haskell и на Python с использованием MPI.

После проведения экспериментального исследования и сравнительного анализа реализаций можно сделать вывод, что лучшей является реализация на MPI, так как почти всегда она демонстрирует наибольшее ускорение при наименьших ресурсах.