Параллельное программирование при решении некоторых задач

Киров 2015

Содержание

1. Введение 3
2. Обзор некоторых технологий параллельного программирования 4
  2.1. C++ и OpenMP 4
  2.2. C++ и MPI 5
  2.3. Java и потоки 6
3. Конфигурация тестирующего оборудования 8
4. Задача нахождения суммы на интервале 10
5. Задача нахождения произведения многочленов 20
6. Задача сортировки массива 32
7. Задача умножения матриц 41
8. Схема Горнера 58
Заключение 61
Литература 62

Введение

Компьютер – это изобретение человечества, использующееся повсеместно. С его помощью решаются как небольшие домашние задачи, так и сложные, требующие огромных вычислений фундаментальные задачи человечества.

Для решения второй группы задач требуется значительное процессорное время, поэтому логичной выглядит мысль о том, что поставленную задачу должны решать несколько компьютеров одновременно.

С этого момент возникают определенные проблемы, так как старые, привычные алгоритмы зачастую не могут быть использованы для параллельных вычислений. Очень часто для этого их требуется модифицировать каким-либо образом, или вовсе отказаться от них в пользу специализированных алгоритмов для параллельных вычислений.

В данной работе будет рассмотрен ряд задач, а также предложены способы их решения при последовательном и параллельном программировании.

Обзор некоторых технологий параллельного программирования

C++ и OpenMP

OpenMP (Open Multi-Processing) - это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с единой памятью на языках C, C++ и Fortran.

Принцип параллельной обработки в OpenMP иллюстрирует рисунок 1. Приложение начинает выполняться в последовательном режиме (поток-мастер), при входе в параллельную область порождаются дополнительные потоки. После порождения каждый поток получает свой уникальный номер, причем поток-мастер всегда имеет номер 0. Все потоки исполняют один и тот же код, соответствующий параллельному региону (параллельной области). При выходе из параллельной области основной поток дожидается завершения остальных, и дальнейшее выполнение программы продолжает только он. Такая схема взаимодействия потоков носит название FORK/JOIN.

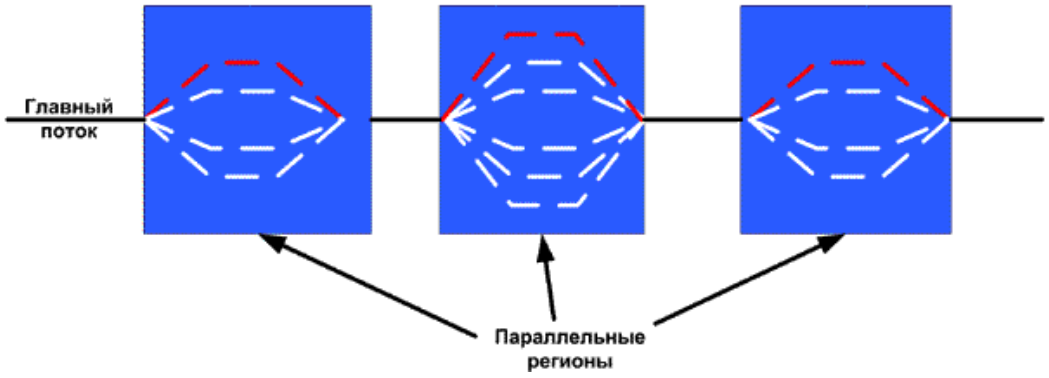

Рис. 1. Параллельные разделы OpenMP

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм. Например, участок кода на языке C++, который должен исполняться несколькими потоками, каждый из которых имеет свою копию переменной N, предваряется следующей директивой: #pragma omp parallel private(N)

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Ключевыми элементами OpenMP являются:

- конструкции для создания потоков (директива parallel),

- конструкции распределения работы между потоками (директивы for и section),
- конструкции для управления работой с данными (выражения shared и private),
- конструкции для синхронизации потоков (директивы critical, atomic и barrier),
- процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num),
- переменные окружения (например, OMP_NUM_THREADS).

К явным преимуществам OpenMP можно отнести:

- Относительная простота написания параллельных программ.
- Код параллельной программы отличается от кода последовательной программы только наличием специальных директив OpenMP, которые, в случае необходимости, могут игнорироваться компилятором. Пользователь получает возможность одновременной работы как с последовательной, так и с параллельной программой.
- Инкрементальный подход в написании параллельной программы: взяв за основу последовательный код, пользователь постепенно добавляет в него специальные директивы.

В качестве недостатков можно отметить:

- Компилятор далеко не всегда может сгенерировать оптимальный код. Приложение, явно использующее многопоточность, как правило, оказывается более эффективным.
- Программу, написанную с использованием OpenMP, невозможно модифицировать для использования в системах с разделенной памятью.

C++ и MPI

MPI (Message Passing Interface) - программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров.

Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя.

MPI поддерживает два подхода к созданию параллельных программ: MIMD и SPMD.

Первый подход подразумевает объедение процессов с различными исходными текстами. Но на практике чаще применятся второй подход - для решения поставленной задачи разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах. Для того, чтобы избежать идентичности вычислений на разных процессорах, на разные процессоры подставляются разные данные.

MPI имеет средства для идентификации процессора, на котором выполняется программа, что предоставляет возможность организовать различия в вычислениях в зависимости от используемого программой процессора.

Интерфейс MPI насчитывает более 120 функций.

Java и потоки

Язык Java разрабатывался с встроенной поддержкой многопоточности. Потоки — один из ключевых элементов технологии Java; они поддерживаются как на языковом (синтаксическом) уровне, так и на уровне виртуальной машины Java и библиотек классов.

Библиотеки классов Java предоставляют класс thread, который поддерживает широкий набор методов для запуска, выполнения и остановки потока, а также проверки его состояния.

Поддержка потоков в Java включает в себя сложный набор примитивов синхронизации на основе мониторов и переменных условия. На уровне языка методы внутри класса или блоки кода, для которых объявляется синхронизация, не выполняются параллельно. Такие методы или блоки выполняются под контролем мониторов, которые помогают убедиться, что данные, доступ к которым осуществляется в этих методах или блоках, остаются в согласованном состоянии. Каждый объект Java имеет собственный монитор, создание экземпляра и активация которого выполняется виртуальной машиной Java в момент первого использования.

Таким образом, главным преимуществом многопоточного программирования в Java является простота использования.

Конфигурация тестирующего оборудования

Тестирование примеров выполнялось на двух конфигурациях:

- Intel Core Duo (1,8 GHz, L2 2Mb), 2 x 1 Gb DDR2 pc5300,
- Intel Atom N450 (1,66 GHz, L2 512Kb), 1 x 1 Gb DDR2 pc5300.

Процессор первой конфигурации является двухъядерным, процессор второй – одноядерный, с поддержкой технологии Hyper-Threading.

Сравнение производительности конфигураций выполнялось с помощью программы PerfomanceTest. Результаты приведены на рисунках 2 и 3.

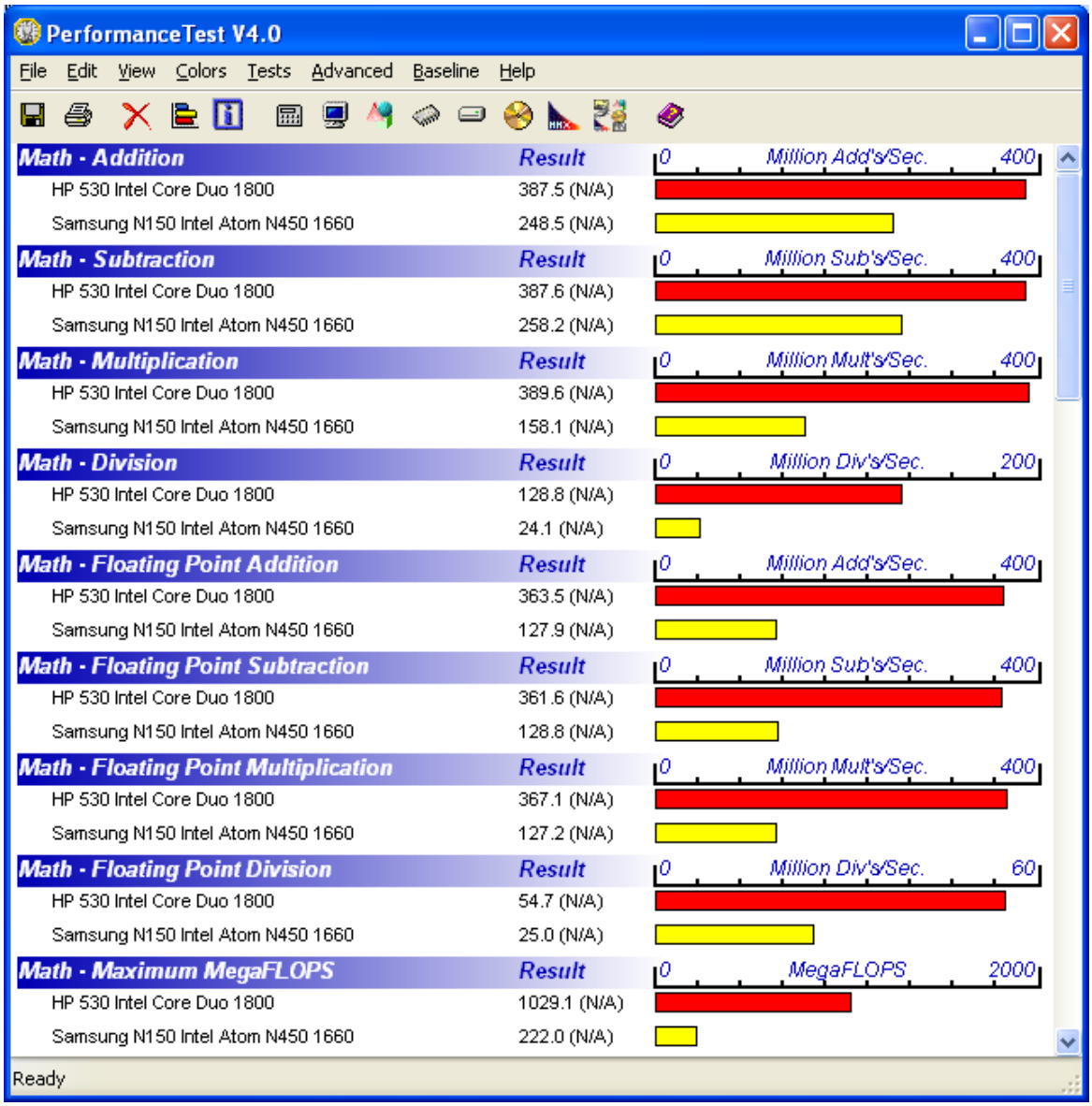

Рис. 2. Сравнение производительности тестирующих конфигураций

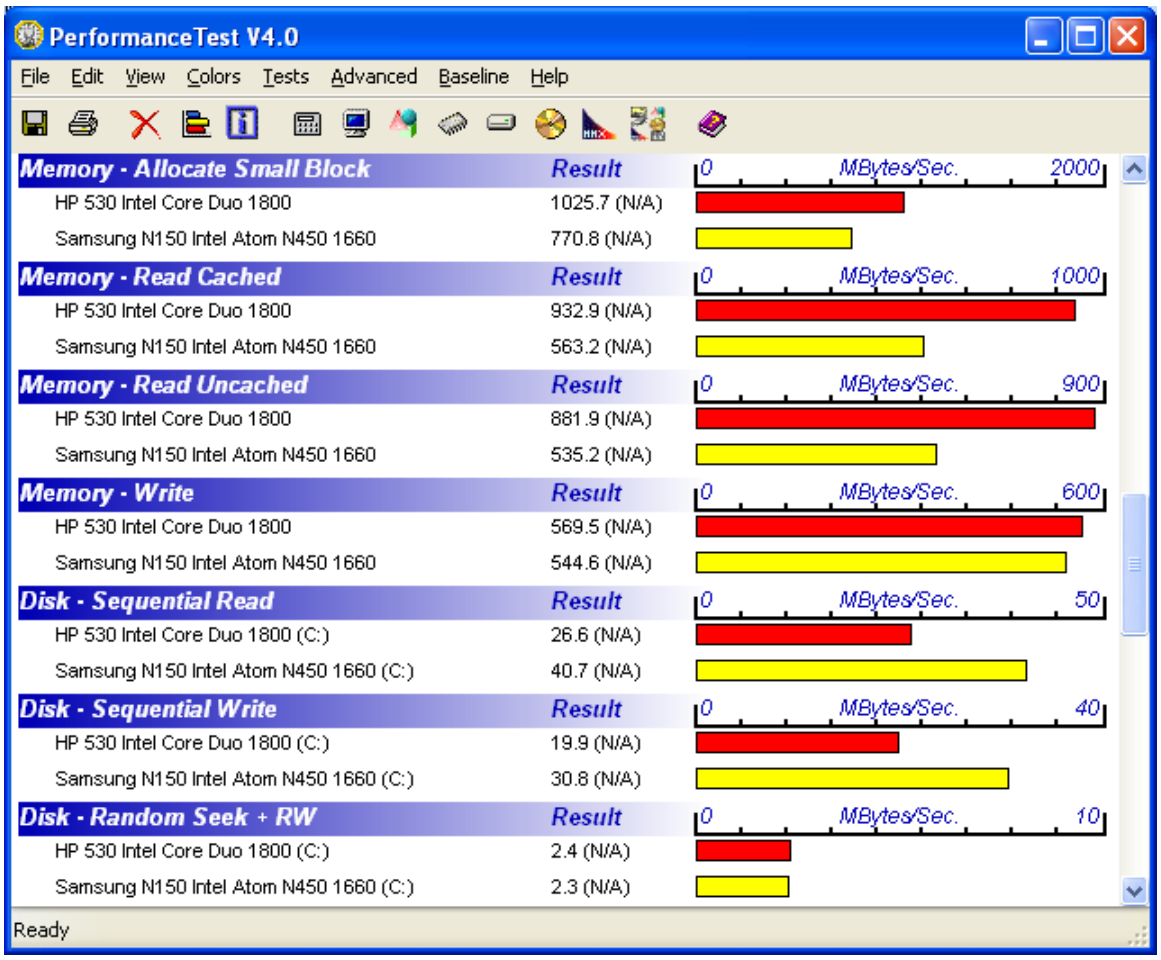

Рис. 3. Сравнение производительности тестирующих конфигураций

Задача нахождения суммы на интервале

Постановка задачи и способы ее решения

Рассмотрим задачу:

По заданному n найти $f(n)$, где $f(n)$ определяется как

$$f(n) = \sum_{i=1}^{n} i \qquad (1)$$

При использовании последовательной схемы вычислений необходимо последовательно выполнить $n-1$ операцию сложения: вычислим сумму первого и второго члена последовательности, к полученной сумме прибавим третий элемент, затем четвертый и так далее.

Для $n=8$ получается следующая последовательность действий:

Дано: $a_1a_2a_3a_4a_5a_6a_7a_8$

1) $(a_1a_2)a_3a_4a_5a_6a_7a_8$
2) $((a_1a_2)a_3)a_4a_5a_6a_7a_8$
3) $(((a_1a_2)a_3)a_4)a_5a_6a_7a_8$
4) $((((a_1a_2)a_3)a_4)a_5)a_6a_7a_8$
5) $(((((a_1a_2)a_3)a_4)a_5)a_6)a_7a_8$
6) $((((((a_1a_2)a_3)a_4)a_5)a_6)a_7)a_8$
7) $(((((((a_1a_2)a_3)a_4)a_5)a_6)a_7)a_8)$

При таком подходе, если вычислительная система имеет более одного процессора, то все они, кроме одного, будут простаивать, так как на каждом шаге алгоритма необходимо выполнить одну операцию сложения и шаг с номером i зависит от всех шагов с меньшими номерами.

Для решения поставленной задачи в многопроцессорной системе можно воспользоваться более эффективной каскадной схемой вычислений. При ее использовании входные данные представляются листьями двоичного дерева, а каждый узел дерева представляет собой результат сложения своих «детей».

При каскадном суммировании для получения результата необходимо также выполнить $n-1$ операцию сложения, но, в предположении, что число входных элементов $n=2^k$, на i-ой итерации параллельно может быть выполнено $\frac{n}{2^i}$ операций сложения.

Таким образом, при условии использования $\frac{n}{2}$ процессоров, полученное ускорение составляет $\frac{n-1}{\log_2 n}$, однако эффективность системы с ростом объема входных данных падает, так как на i-ой итерации простаивает $\frac{n}{2}-\frac{n}{2^i}$ процессоров.

Каскадную схему можно модифицировать следующим образом:

Пусть на первом шаге входные данные разбиваются на $\frac{n}{\log_2 n}$ групп, суммирование элементов которых осуществляется последовательно.

После этого к полученным частичным суммам применяется описанная выше каскадная схема вычислений.

Такая схема обладает меньшим ускорением, но большей эффективностью.

Предположим, что в системе два процессора. Тогда для приведенного выше примера можно использовать более эффективную схему:

Дано: $a_1a_2a_3a_4a_5a_6a_7a_8$

1) $(a_1a_2)a_3a_4(a_5a_6)a_7a_8$

2) $((a_1a_2)a_3)a_4((a_5a_6)a_7)a_8$

3) $(((a_1a_2)a_3)a_4)(((a_5a_6)a_7)a_8)$

4) $((((a_1a_2)a_3)a_4)(((a_5a_6)a_7)a_8))$

Полученная схема вместо 7 шагов, потребовала 4 шага.

Разницу между двумя вышеописанными алгоритмами иллюстрируют рисунки 4 и 5.

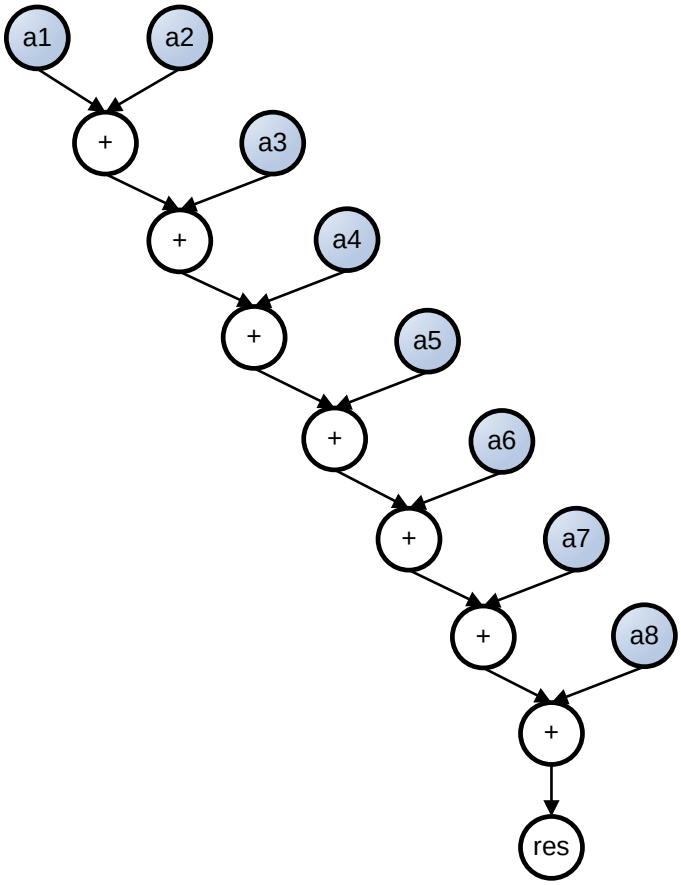

Рис. 4. Последовательный алгоритм решения задачи

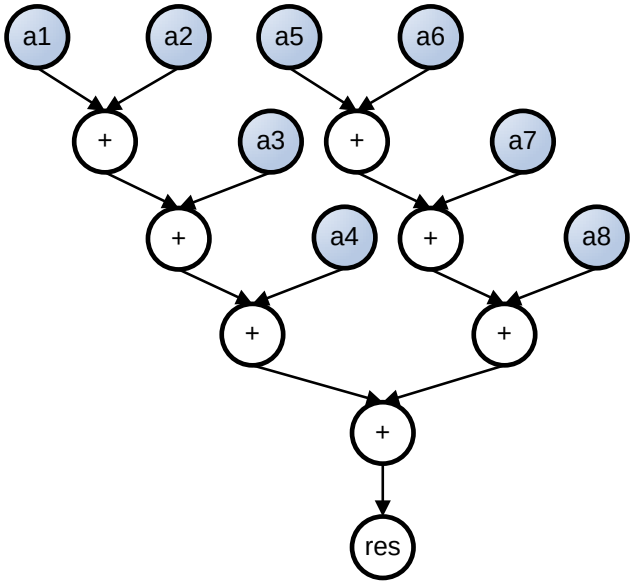

Рис. 5. Алгоритм решения задачи на вычислительной системе с двумя процессорами с использованием модифицированной каскадной схемы

Действительно, в первом случае высота дерева составляет 7 шагов, а во втором – 4 шага.

Реализация алгоритма

Java

Последовательная реализация алгоритма.

```
public class Sum {

	SumResult Calc(int from, int to) {
		long res = 0;
		//Замеряем время старта
		long time = System.currentTimeMillis();
		//Считаем сумму на интервале from - to
		for(int i = from; i <= to; i++)
			res +=i;
		//Возвращаем ответ
		return new SumResult(res, System.currentTimeMillis() - time);
	}

}
```

Класс для параллельной реализации алгоритма. Каждый экземпляр класса запускается в отдельном потоке.

```
//Для того, чтобы класс мог выполняться в отдельном потоке, необходимо, чтобы он реализовывал
//интерфейс Runnable
public class SumTh implements Runnable{

	//Поток, в котором будет запускаться класс
	Thread th;
	//Этот класс будет считать сумму на интервале from - to
	int from, to;
	//Результат будет храниться в поле res
	long res;

	//Конструктор класса
	//На вход получается переменные to и from
	//А также имя потока для его идентификации
	SumTh(int from, int to, String thName) {
		res = 0;
		this.from = from;
		this.to = to;

		//Данный экземпляр класса создается в новом потоке th с именем thName
		//Поток th начинает свою работу (передается управление методу run)
		//после вызова метода start
		th = new Thread(this, thName);
		th.start();
	}

	//Здесь осуществялется основная работа – подсчет результата
	public void run() {
		for(int i = from; i <= to; i++)
			res +=i;
	}
}
```

Класс – мастер, который запускается первым, инициализирует нужное число потоков, а затем подсчитывает итоговый результат.

```
public class SumParallel {

	//Метод Calc получает на вход границы интервала и число потоков, которые надо создать
	SumResult Calc(int from, int to, int thCount) {
		//Замеряем время старта работы
		long time = System.currentTimeMillis();
		//Массив дочерних классов, каждый из которых будет считать значение подинтервала
		SumTh[] th = new SumTh[thCount];
		//В этом блоке считается длина интервала, которую будет передана в качестве подзадачи
		//дочерним потокам
		int next = from, d;
		if((to - from + 1) % thCount != 0)
			d = (to - from + 1) / thCount + 1;
		else
			d = (to - from + 1)/ thCount;

		//Создаются дочерние потоки
		for(int i = 0; i < thCount; i++) {
			th[i] = new SumTh(next, Math.min(next + d - 1, to), String.valueOf(i));
			next += d;
		}

		//В этом блоке поток-мастер ждет завершения работы дочерних потоков
		try {
			//Специальный метод join усыпляет опросивший поток до тех пор, пока
			//опрошенный поток не завершит работу
			for(int i = 0; i < thCount; i++)
				th[i].th.join();
		} catch (InterruptedException e) {
			//В случае непредвиденной ошибки на консоль будет выведено сообщение об
			//ошибке
			System.err.print("Ошибка выполнения");
			System.err.flush();
			return new SumResult(0, 0);
		}

		//Подсчитаем итоговый результат с помощью опроса дочерних потоков
		long res = 0;
		for(int i = 0; i < thCount; i++)
			res += th[i].res;

		//Программисту не нужно заботиться об уничтожении дочерних потоков, так как за него
		//это сделает автоматический сборщик мусора

		//Вернем итоговый результат и время работы
		return new SumResult(res, System.currentTimeMillis() - time);
	}

}
```

Класс результата. С его помощью будет представляться ответ и время, которое потребовалось для его вычисления.

```
public class SumResult {

	//ответ
	private long res;
	//время
	private long time;

	SumResult(long result, long time) {
		this.res = result;
		this.time = time;
	}

	public long getResult() {
		return res;
	}

	public long getTime() {
		return time;
	}

	public boolean equal(SumResult other) {
		return this.res == other.res;
	}
}
```

Класс, который выполняет запуск параллельной и последовательной программы.

```
public class Run {

	//Метод main запускается JVM автоматически
	public static void main(String[] args) {
		//Установим границы интервала и число потоков для параллельной программы
		int down = 1, up = 1000000000, thCount = 2;

		//Последовательная программа. Вывод на консоль времени работы
		SumResult res1 = new Sum().Calc(down, up);
		System.out.println(res1.getTime());

		//Параллельная программа. Вывод на консоль времени работы
		SumResult res2 = new SumParallel().Calc(down, up, thCount);
		System.out.print(res2.getTime());

		System.out.flush();
	}

}
```

OpenMP

```
#define _CRT_SECURE_NO_DEPRECATE
#include <iostream>
//Подключаем OpenMP
#include <omp.h>
#include <cmath>
#include <algorithm>
#include <time.h>
#pragma comment (linker,"/STACK:64000000")
using namespace std;

int main()
{
	freopen("input.txt","r",stdin);
	freopen("output.txt","w",stdout);

	//Устанавливаем границу интервала
	long long i, up = 1000000000;
	long long res = 0;

	//Замеряем время старта
	clock_t start = clock();
	//Пробуем посчитать сумму на интервале последовательно
	for(i = 1; i <= up; i++)
		res += i;

	//Выводим время работы и полученный результат
	cout << clock() - start << "\n";
	cout << res <<"\n";

	//Обнуляем результат и замеряем время снова
	res = 0;
	start = clock();

	//Данная директива указывает компилятору, что блок должен выполняться параллельно
	#pragma omp parallel
	{
		//Данная директива указывает, что цикл for, следующий за ней необходимо выполнять
		//параллельно. При этом переменная i должна быть собственной в каждом потоке, а
		//все значения res необходимо просуммировать
		#pragma omp for private(i) reduction(+: res)
		for(i = 1; i <= up; i++)
			res += i;
	}

	//Выводим время работы и полученный результат
	cout << clock() - start << "\n";
	cout << res;

	return 0;
}
```

MPI

```
//Подключаем MPI
#include "mpi.h"
#include <stdio.h>
//Определяем размерность задачи
#define N 1000000000
int main(int argc, char *argv[])
{
	int numtasks, taskid, numworkers, source, dest, tsk, aveN, extra, offset, k, rc;
	long long res = 0, ans = 0;
	 //Переменная для времени работы
	double time;
	MPI_Status status;
	rc = MPI_Init(&argc,&argv);
	rc |= MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
	rc |= MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
	//Проверяем на корреткность
	if (rc != MPI_SUCCESS)
		printf ("Error");
	numworkers = numtasks - 1;
	//Если мастер
	if (taskid == 0)
	{
		//Делим на подзадачи
		aveN = N/numworkers;
		extra = N%numworkers;
		offset = 1;
		//Замеряем время
		time = MPI_Wtime();
		//Раздаем подзадачи
		for (dest=1; dest<=numworkers; dest++)
		{
			if(dest <= extra)
				tsk = aveN + 1;
			else
				tsk = aveN;
			MPI_Send(&offset, 1, MPI_INT, dest, FROM_0, MPI_COMM_WORLD);
			MPI_Send(&tsk, 1, MPI_INT, dest, FROM_0, MPI_COMM_WORLD);
			offset += tsk - 1;
		}
		//Принимаем решения
		for (source = 1; source <= numworkers; source++)
		{
			MPI_Recv(&res, 2, MPI_UNSIGNED_LONG, source, 2, MPI_COMM_WORLD,
&status);
			ans += res;
		}
		//Выводим время работы
		printf ("%.3lf\n", MPI_Wtime() - time);
	}
	//Если не мастер
	if (taskid > 0)
	{
		//Получаем задачу
		MPI_Recv(&offset, 1, MPI_INT, 0, FROM_0, MPI_COMM_WORLD, &status);
		MPI_Recv(&tsk, 1, MPI_INT, 0, FROM_0, MPI_COMM_WORLD, &status);
		//Решаем ее
		for (k = offset; k < offset + tsk - 1; k++)
			res += k;
		//Возвращаем ответ
		MPI_Send(&res, 2, MPI_UNSIGNED_LONG, 0, 2, MPI_COMM_WORLD);
```

```
        }
        //Конец программы
        MPI_Finalize();
}
```

Сравнение быстродействия

Intel Core Duo

Java			
n	Один поток	Два потока	Коэффициент
1000000	16	16	1,0000
10000000	70	31	2,2581
100000000	507	282	1,7979
1000000000	5031	2687	1,8723

C++					
	Один поток	OpenMP		MPI	
n			Коэффициент		Коэффициент
1000000	16	15	1,0667	31	0,5161
10000000	31	16	1,9375	16	1,9375
100000000	297	192	1,5469	210	1,4143
1000000000	3079	1750	1,7594	2094	1,4704

Intel Atom

Java			
n	Один поток	Два потока	Коэффициент
1000000	15	15	1,0000
10000000	78	47	1,6596
100000000	711	453	1,5695
1000000000	7078	4469	1,5838

C++					
	Один поток	OpenMP		MPI	
n			Коэффициент		Коэффициент
1000000	0	0	1,0000	16	1,0000
10000000	31	40	0,7750	31	1,0000
100000000	488	395	1,2354	476	1,0252
1000000000	5188	3737	1,3883	4513	1,1496

Задача нахождения произведения многочленов

Постановка задачи и способы ее решения

Пусть заданы два многочлена $A=\sum_{i=0}^{n} a_i x^i$ и $B=\sum_{i=0}^{m} b_i x^i$. Тогда их произведением назовем многочлен C, такой что

$$C = AB, \tag{2}$$

или

$$C=\sum_{i=0}^{n}\sum_{j=0}^{m} a_i b_j x^{i+j} = \sum_{i=0}^{m+n} x^i \sum_{j=0}^{i} a_j b_{i-j}\ . \tag{3}$$

По заданным многочленам A и B требуется найти многочлен C.

Данная задача может быть решена по определению. Сложность такого алгоритма составит $O(nm)$.

При этом такой алгоритм обладает высокой степенью параллелизма. Каждый поток может обрабатывать первый многочлен и часть второго многочлена. После этого частичные результаты необходимо сложить.

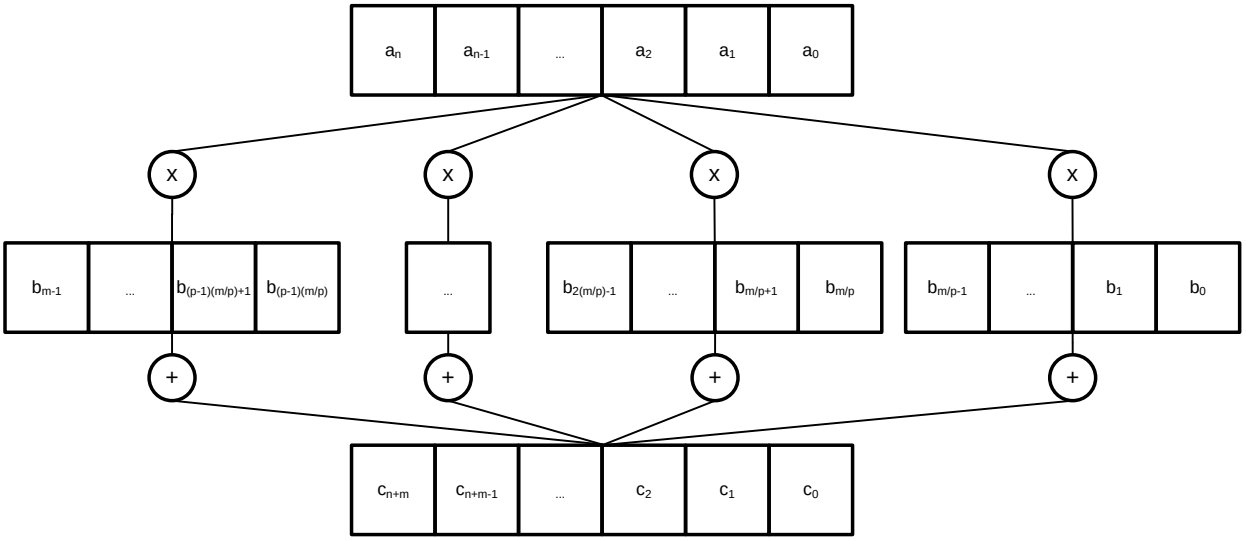

Рис. 6. Алгоритм решения задачи «по определению» на вычислительной системе p процессорами

В случае параллельной реализации рассмотренного алгоритма потребуется выполнить $\frac{m}{p}n$ операций умножения в параллельном режиме на p процессорах, после этого необходимо сложить p полученных частичных результата размерностью $\frac{m}{p}+n-1$.

Таким образом, при реализации на вычислительной системе с p процессорами потребуется $\frac{m}{p}n+p(\frac{m}{p}+n-1)$ операций.

Рассмотрим функцию

$$f(p) = \frac{m}{p}n + p(\frac{m}{p} + n - 1) \quad (4)$$

Найдем экстремум функции. Исходя из

$$f'(p) = -\frac{m}{p^2}n + n = 0 \quad (5)$$

получаем, что наилучшее ускорение достигается при использовании $\sqrt{m}$ процессоров и достигает $\frac{\sqrt{m}n}{2n+1}$.

Для решения поставленной задачи можно также воспользоваться одним из алгоритмов быстрого умножения: например, быстрым преобразованием Фурье или алгоритмом Карацубы.

Рассмотрим алгоритм Карацубы детальнее.

Без потери общности допустим, что требуется перемножить два многочлена степени $n = 2k$.

Представим исходные многочлены как

$$A = A_1 + x^k A_2,\ B = B_1 + x^k B_2. \quad (6)$$

Воспользуемся преобразованием

$$(a + bx)(c + dx) = ac + ((a + b)(c + d) - ac - bd)x + bdx^2. \quad (7)$$

Тогда

$$AB = A_1B_1 + x^k((A_1 + A_2)(B_1 + B_2) - (A_1B_1 + A_2B_2)) + x^n A_2B_2. \quad (8)$$

При таком подходе потребуется выполнить три умножения многочленов степени в два раза меньше исходной и конечное число сложений и вычитаний со сложностью $O(n)$. Очевидно, что каждое из трех умножений может быть выполнено на отдельном процессоре.

Представленный алгоритм несложно обобщить для использования схемы «разделяй и властвуй».

Для оценки сложности описанного алгоритма необходимо решить следующее рекуррентное соотношение:

$$\varphi(n) = 3\varphi(\frac{n}{2}) + cn. \quad (9)$$

По основной теореме получаем, что

$$O(cn) = O(n^{\log_2 3 - \varepsilon}), \text{ при } \varepsilon = 1 \quad (10)$$

Таким образом, сложность описанного алгоритма составляет $O(n^{\log_2 3})$, где n - степень большего из многочленов.

Ниже будет проведено сравнение времени работы некоторых методов решения поставленной задачи: алгоритма решения задачи по определению,

алгоритма решения задачи по определению с использованием параллельной обработки, алгоритма умножения Карацубы, алгоритма умножения Карацубы с использованием параллельной обработки подзадач, алгоритма умножения Карацубы с обобщением по схеме «разделяй и властвуй» (вместо непосредственного решения подзадачи по определению она вновь будет решаться алгоритмом Карацубы) и алгоритма Карацубы с обобщением по схеме «разделяй и властвуй» с параллельной обработкой подзадач.

При этом в последовательной версии умножения Карацубы в качестве значения «точки пересечения» (величина подзадачи, начиная с которой вместо умножения Карацубы выполняется обычный метод умножение) используется 32, а в параллельной версии – 512.

Реализация алгоритма

Java

Класс, реализующий «наивный» алгоритм нахождения произведения многочленов по определению.

```
public class Polynom {

	//Метод получает на вход два исходных многочлена
	PolynomResult Calc(int[] arg0, int[] arg1) {
		//Замеряем время работы
		long time = System.currentTimeMillis();
		//Выделяем память для ответа
		long[] res = new long[arg0.length + arg1.length - 1];

		//Вычисление произведения по определению
		for(int i = 0; i < arg0.length; i++)
			for(int j = 0; j < arg1.length; j++)
				res[i + j] += arg0[i] * arg1[j];

		//Метод возвращает ответ и приближенное время работы программы
		return new PolynomResult(res, System.currentTimeMillis() - time);
	}

}
```

Класс для параллельной реализации «наивного» алгоритма. Каждый экземпляр запускается в отдельном потоке.

```
public class PolynomTh implements Runnable{

	Thread th;
	int[] x, y;
	long[] res;

	//Конструктор получает исходные массивы
	PolynomTh(int[] x, int[] y, String thName) {
		res = new long[x.length + y.length - 1];
		this.x = x;
		this.y = y;

		th = new Thread(this, thName);
		th.start();
	}

	//Здесь выполняется вычисление произведения многочленов
	public void run() {
		for(int i = 0; i < x.length; i++)
			for(int j = 0; j < y.length; j++)
				res[i + j] += x[i] * y[j];
	}
}
```

Класс для параллельного вычисления произведения многочленов.

```
import java.util.Arrays;

public class PolynomParallel {

	//Метод принимает исходные многочлены и число потоков, которые нужно создать
	PolynomResult Calc(int[] x, int[] y, int thCount) {
		long time = System.currentTimeMillis();
		PolynomTh[] th = new PolynomTh[thCount];
		int next = 0, d;
		//Вычисление размера подзадачи
		if(x.length % thCount != 0)
			d = x.length / thCount + 1;
		else
			d = x.length / thCount;

		//Создание дочерних процессов
		for(int i = 0; i < thCount; i++) {
			th[i] = new PolynomTh(Arrays.copyOfRange(x, next, Math.min(next + d, x.length)), y,
String.valueOf(i));
			next += d;
		}

		//Ожидание завершения дочерних процессов
		try {
			for(int i = 0; i < thCount; i++)
				th[i].th.join();
		} catch (InterruptedException e) {
			System.err.print("Ошибка выполнения");
			System.err.flush();
			return new PolynomResult(null, 0);
		}

		long[] res = new long[x.length + y.length - 1];
		for(int i = 0; i < thCount; i++)
			for(int j = 0; j < th[i].res.length; j++)
				res[i * d + j] += th[i].res[j];

		return new PolynomResult(res, System.currentTimeMillis() - time);
	}

}
```

Класс, реализующий умножение Карацубы с параллельным решением подзадачи.

```
import java.util.Arrays;
import polynom.PolynomResult;

public class KaratsubaPolynomParallel {

	//Метод получает исходные массивы
	PolynomResult Calc(int[] arg0, int[] arg1) {
		long time = System.currentTimeMillis();
		int n = Math.max(arg0.length, arg1.length), i;

		//Выделение памяти для декомпозиции исходных массивов
		if(n % 2 == 1)
			n++;
		int N = n / 2;
		long[] res = new long[2 * n];
		int[] a1 = new int[n / 2];
		int[] a2 = new int[n / 2];
		int[] b1 = new int[n / 2];
		int[] b2 = new int[n / 2];
		long[] a, b, c;

		//Декомпозиция исходных массивов
		for(i = 0; i < N; i++) {
			if(i < arg0.length)
				a1[i] = arg0[i];
			if(i < arg1.length)
				b1[i] = arg1[i];
			if(N + i < arg0.length)
				a2[i] = arg0[N + i];
			if(N + i < arg1.length)
				b2[i] = arg1[N + i];
		}

		//Вычисление частичных результатов с помощью параллельной программы
		a = new PolynomParallel().Calc(a1, b1, 2).getResult();
		b = new PolynomParallel().Calc(a2, b2, 2).getResult();

		for(i = 0; i < N; i++) {
			a1[i] += a2[i];
			b1[i] += b2[i];
		}

		c = new PolynomParallel().Calc(a1, b1, 2).getResult();

		//Получение конечного результата
		for(i = 0; i < n; i++) {
			if(i < a.length) {
				res[i + N] -= a[i];
				res[i] += a[i];
			}
			if(i < b.length) {
				res[i + N] -= b[i];
				res[i + n] +=b[i];
			}
			if(i < c.length)
				res[i + N] += c[i];
		}
```

```
		return new PolynomResult(Arrays.copyOfRange(res, 0, arg0.length + arg1.length - 1), System.currentTimeMillis() - time);
	}

}
```

Класс, реализующий алгоритм Карацубы с использованием подхода «разделяй и властвуй» и параллельным решением подзадач.

```
import java.util.Arrays;
import polynom.PolynomResult;

public class KaratsubaPolynomRecParallelTh implements Runnable {

	Thread th;
	PolynomResult result;
	int[] arg0, arg1;

	//Конструктор принимает исходыне массивы
	KaratsubaPolynomRecParallelTh(int[] arg0, int[] arg1, String thName) {
		this.arg0 = arg0;
		this.arg1 = arg1;

		//И запускает вычисления в отдельном потоке
		th = new Thread(this, thName);
		th.start();
	}

	public void run() {
		//Замеряем время начала выполнения программы
		long time = System.currentTimeMillis();
		int n = Math.max(arg0.length, arg1.length), i, j;
		if(n % 2 == 1)
			n++;
		long[] res = new long[2 * n];

		//Если размерность подзадачи менее «точки пересечения», то результат считается
		//по определению
		if(n <= 512) {
			for(i = 0; i < arg0.length; i++)
				for(j = 0; j < arg1.length; j++)
					res[i + j] += (long)arg0[i] * arg1[j];
		}
		else {
			//В противном случае продолжаем решать задачу с помощью алгоритма Карацубы
			int N = n / 2;
			int[] a1 = new int[n / 2];
			int[] a2 = new int[n / 2];
			int[] b1 = new int[n / 2];
			int[] b2 = new int[n / 2];

			//Выполняем декомпозицию исходных массивов
			for(i = 0; i < N; i++) {
				if(i < arg0.length)
					a1[i] = arg0[i];
				if(i < arg1.length)
					b1[i] = arg1[i];
				if(N + i < arg0.length)
					a2[i] = arg0[N + i];
				if(N + i < arg1.length)
					b2[i] = arg1[N + i];
			}
```

```
                        //Считаем частичные результаты в отдельном потоке параллелльно
                        KaratsubaPolynomRecParallelTh x = new KaratsubaPolynomRecParallelTh(a1, b1, "");
                        KaratsubaPolynomRecParallelTh y = new KaratsubaPolynomRecParallelTh(a2, b2, "");

                        for(i = 0; i < N; i++) {
                                a1[i] += a2[i];
                                b1[i] += b2[i];
                        }

                        KaratsubaPolynomRecParallelTh z = new KaratsubaPolynomRecParallelTh(a1, b1, "");

                        //Ждем завершения дочерних потоков
                        try {
                                x.th.join();
                                y.th.join();
                                z.th.join();
                        } catch (InterruptedException e) {
                                System.err.print("Ошибка выполнения");
                                System.err.flush();
                                result = new PolynomResult(null, 0);
                        }

                        //Получаем частичные результаты
                        long[] a = x.result.getResult();
                        long[] b = y.result.getResult();
                        long[] c = z.result.getResult();

                        //Получаем конечный ответ
                        for(i = 0; i < n; i++) {
                                if(i < a.length) {
                                        res[i + N] -= a[i];
                                        res[i] += a[i];
                                }
                                if(i < b.length) {
                                        res[i + N] -= b[i];
                                        res[i + n] +=b[i];
                                }
                                if(i < c.length)
                                        res[i + N] += c[i];
                        }
                }

                //Возвращаем результат и время работы
                result = new PolynomResult(Arrays.copyOfRange(res, 0, arg0.length + arg1.length - 1),
System.currentTimeMillis() - time);
        }

}
```

OpenMP

Реализация алгоритма по определению в последовательном и параллельном режимах. В параллельном режиме выполняется разделение второго многочлена между потоками.

```
#define _CRT_SECURE_NO_DEPRECATE
#include <iostream>
//Подключаем OpenMP
#include <omp.h>
#include <cmath>
#include <algorithm>
#include <time.h>
//Константы для явного указания размерности многочленов
#define N 10000
#define M 100000
#pragma comment (linker,"/STACK:64000000")
using namespace std;
int x[N] = {0};
int y[M] = {0};
long long res1[N + M - 1] = {0};
long long res2[N + M - 1] = {0};
int main()
{
	freopen("input.txt","r",stdin);
	freopen("output.txt","w",stdout);

	int i, j;
	long long time1, time2;

	//Инициализация многочленов
	for(i = 0; i < N; i++)
		x[i] = i;
	for(i = 0; i < M; i++)
		y[i] = i;

	//Замеряем время начала выполнения умножения
	//Выполняем умножение в последовательном режиме
	clock_t start = clock();
	for(i = 0; i < N; i++)
		for(j = 0; j < M; j++)
			res1[i + j] += (long long)x[i] * y[j];

	//Запоминаем время работы
	time1 = clock() - start;

	//Замерям время начала умножения
	start = clock();

	//Данная директив указывает на то, что участок кода должен выполняться с использованием
	//OpenMP. Причем массив res2 должен быть общим для всех потоков,
	//а переменные i и j должны быть уникальны для каждого потока
	#pragma omp parallel shared(res2) private(i, j)
	{
		for(i = 0; i < N; i++)
			//Цикл for следует выполнять в многопоточном режиме
			#pragma omp for
			for(j = 0; j < M; j++)
				res2[i + j] += (long long)x[i] * y[j];
	}
```

```
	//Запоминаем время работы
	time2 = clock() - start;

	//Проверяем результат на корректность
	int flag = 0;
	for(i = 0; i < N + M - 1; i++)
		if (res1[i] != res2[i])
		{
			flag = 1;
			break;
		}

	//Если результат некорректен, то будет выведено сообщение,
	//в противном случае – время работы
	if (flag)
		cout << "Bad answer";
	else
		cout << time1 << "\n" << time2;

	return 0;
}
```

Сравнение быстродействия

Алгоритм вычисления произведения многочленов по определению

Intel Core Duo

Java				
n	m	Один поток	Два потока	Коэффициент
1000	1000	16	15	1,0667
1000	10000	156	81	1,9259
10000	1000	148	78	1,8974
10000	10000	1369	706	1,9391
10000	100000	13664	6976	1,9587
100000	10000	13570	6859	1,9784
100000	100000	174922	96273	1,8169

C++				
		Один поток	OpenMP	
n	m			Коэффициент
1000	1000	15	47	0,3191
1000	10000	86	70	1,2286
10000	1000	79	328	0,2409
10000	10000	813	671	1,2116
10000	100000	8188	6247	1,3107
100000	10000	8172	6719	1,2163
100000	100000	81969	65485	1,2517

Intel Atom

Java				
n	m	Один поток	Два потока	Коэффициент
1000	1000	47	31	1,5161
1000	10000	297	196	1,5153
10000	1000	298	188	1,5851
10000	10000	2942	1848	1,5920
10000	100000	29856	18766	1,5910
100000	10000	29150	18468	1,5784
100000	100000	296983	189012	1,5712

C++				
		Один поток	OpenMP	
n	m			Коэффициент
1000	1000	15	31	0,4839
1000	10000	176	197	0,8934
10000	1000	125	301	0,4153
10000	10000	1983	1524	1,3012
10000	100000	18997	13609	1,3959
100000	10000	19047	14422	1,3207
100000	100000	197234	138969	1,4193

Алгоритм вычисления произведения многочленов с помощью умножения Карацубы

Intel Core Duo

Java					
n	m	Алгоритм Карацубы	Алгоритм Карацубы с параллельной обработкой	Алгоритм Карацубы "разделяй и властвуй"	Алгоритм Карацубы "разделяй и властвуй" с параллельной обработкой
10000	10000	937	469	219	219
20000	20000	3734	1860	625	456
100000	100000	113187	66781	10219	7625

Intel Atom

Java					
n	m	Алгоритм Карацубы	Алгоритм Карацубы с параллельной обработкой	Алгоритм Карацубы "разделяй и властвуй"	Алгоритм Карацубы "разделяй и властвуй" с параллельной обработкой
10000	10000	2031	1422	422	407
20000	20000	8125	5734	1141	984
100000	100000	197531	145094	14250	12891

Наилучший результат показал алгоритм Карацубы, обобщенный по схеме «разделяй и властвуй» с использованием параллельной обработки подзадач. С его помощью удалось сократить требуемое время для решении задачи размерностью 10^{10} с ~175 до ~7,5 секунд в случае использования платформы Intel Core Duo (примерно в ~22,94 раза быстрее) и с ~297 до ~13 секунд в случае использования платформы Intel Core Duo (примерно в ~23,04 раза быстрее).

Задача сортировки массива

Постановка задачи и способы ее решения

Задача сортировки формулируется следующим образом: задано множество $S=\{a_1,a_2...a_n\}$, требуется переупорядочить элементы этого множества таким образом, чтобы элементы полученного множества $S'=\{a'_1,a'_2...a'_n\}$ удовлетворяли свойству $a'_1\le a'_2\le...\le a'_i\le a'_{i+1}\le...\le a'_n$ при всех $i=1...n$.

Для решения поставленной задачи существует множество последовательных алгоритмов: «пузырьковая» сортировка, сортировка вставками, сортировка выборкой, сортировка Шелла, быстрая сортировка, пирамидальная сортировка, сортировка слиянием, сортировка подсчетом, цифровая сортировка и т. д.

Асимптотическая сложность описанных алгоритмов варьируется от $O(n^2)$ до $O(n)$.

Некоторые из этих алгоритмов могут быть обобщены для выполнения на вычислительных системах с общей памятью.

Рассмотрим подробнее быструю сортировку.

Данный алгоритм базируется на схеме «разделяй и властвуй».

1. Пусть на определенном шаге требуется упорядочить подинтервал $[l,r]$ исходного массива. Назовем элемент с номером r опорным.

2. Разделим все элементы множества $[l,r-1]$ на две группы: элементы, меньшие опорного, и элементы, большие опорного. Пусть в первой группе x элементов. Будем выполнять разбиение таким образом, чтобы все эти элементы в итоге оказались на позициях $[l,x-1]$.

3. Поменяем опорный элемент с элементом на позиции x. После выполнения данной процедуры исходный подинтервал имеет вид $[(l,x-1),x,(x+1,r)]$, где $(l,x-1)$ - элементы, меньшие опорного, x- опорный элемент, $(x+1,r)$ - элементы, большие опорного.

4. Очевидно, что теперь опорный элемент находится на позиции, на которой и должен находиться в отсортированном массиве. Выполним описанную процедуру рекурсивно для подинтервалов $[l,x-1]$ и $[x+1,r]$.

Предположим, что в каждом случае сортируемый подинтервал разбивался на две подзадачи одинакового размера. В таком случае сложность описанного алгоритма составит $O(n\log_2 n)$.

В худшем случае подзадача размерности Z может разбиваться на две подзадачи размерности $Z-1$ и 0. В таком случае сложность описанного алгоритма $O(n^2)$.

На практике время работы алгоритма сильно зависит от входных данных и стратегии выбора опорного элемента. Например, выбирая опорный элемент случайно (выбрать случайный элемент из интервала $[l, r-1]$ и поменять его с элементом на позиции r) можно гарантировать средне-ожидаемое время работы $O(n \log_2 n)$.

Данный алгоритм может быть распараллелен для систем с общей памятью без дополнительных накладных расходов. В случае использования числа процессоров, пропорционального объему входных данных, ускорение составит $\frac{\log_2 n}{2}$. Однако на практике, как правило, достигается гораздо меньшее ускорение, что связано с ограниченным числом процессоров в системе.

Очевидным образом, время выполнения алгоритма на вычислительной системе с разделенной памятью и p процессорами составит порядка $O(\frac{n}{p}\log_2 p + \frac{n}{p}\log_2 \frac{n}{p})$ операций без учета межпроцессорных взаимодействий.

Оценка сложности коммуникационных операций является сложной задачей, решение которой зависит от многочисленных факторов: топологии сети, латентности, пропускной способности и т.д., - и не рассматривается в данной работе. Ограничимся оценкой в $O(\alpha \log_2^2 p + \frac{\beta n}{2p}\log_2 p + \gamma \log_2 p)$ коммуникационных операций при условии выбора оптимальной топологии сети, где α, β, γ - некоторые параметры сети.

Для систем с разделенной памятью часто применяются специализированные алгоритмы сортировки: чет-нечетная сортировка, сортировка «сдваиванием» и т.д.

В качестве примера рассмотрим модификацию сортировки «пузырьком» (чет-нечетная сортировка или обменная сортировка Бэтчера).

Пусть есть массив размерности n и p процессоров. Разобьем массив на p равных частей, передадим каждому процессору для обработки по части. После выполнения сортировки каждой из частей, необходимо объединить полученные отсортированные участки. Для этого объединим две «соседние» части A_i и A_{i+1} в одну с помощью последовательного слияния. Теперь передадим первую половину («меньшую») процессору, который обрабатывал часть A_i, а вторую («большую») – другому.

Таким образом, «пузырьками» при взаимодействии процессоров выступают подзадачи. При этом на каждой итерации необходимо обрабатывать либо четные, либо нечетные группы элементов.

Для наглядности рассмотри пример:

Задан массив {8, 7, 6, 5, 4, 3, 2, 1}, в системе четыре процессора.

Первый процессор получает для обработки первые два элемента, второй – третий и четвертый, третий – пятый и шестой, четвертый – седьмой и восьмой.

После «внутренней» сортировки получаем массив {7, 8, 5, 6, 3, 4, 1, 2}.

На первом шаге слияние и обмен выполняется между первым и вторым процессорами и между третьим и четвертым процессорами. В результате получается массив {5, 6, 7, 8, 1, 2, 3, 4}.

На втором шаге слияние и обмен выполняется между вторым и третьим процессорами. В результате получается массив {5, 6, 1, 2, 7, 8, 3, 4}.

На третьем шаге снова взаимодействуют первый – второй процессоры и третий – четвертый процессоры. {1, 2, 5, 6, 3, 4, 7, 8}.

После четвертого шага получается упорядоченный массив {1, 2, 3, 4, 5, 6, 7, 8}.

Первая часть алгоритма (сортировка каждым процессором своей части исходного массива) в общем случае может быть выполнена за $O(\frac{n}{p}\log_2\frac{n}{p})$ операций при использовании одного из методов быстрой сортировки. В частном случае эта часть может быть выполнена за $O(\frac{n}{p})$ операций.

Во второй части алгоритма необходимо выполнить порядка p итерация на каждой из которых выполняется слияние двух массивов размерности $\frac{n}{p}$. При использовании метода «двух указателей» эта часть может быть выполнена за $\frac{2n}{p}$ операций. Таким образом, сложность второй части составляется $O(2n)$ операций, а общая сложность алгоритма без учета коммуникационных взаимодействий составляет $O(\frac{n}{p}\log_2\frac{n}{p}+2n)$ операций. При оптимальном выборе топологии сети сложность коммуникационных взаимодействий составит $O(ap+\beta n)$ операций.

Предложенный алгоритм может быть модифицирован таким образом, чтобы сократить ожидаемое количество операций, требуемых для сортировки исходного массива. Модифицированный алгоритм носит название сортировки Шелла и, хотя имеет такую же асимптотическую оценку, на практике показывает примерно в два раза лучший результат.

Идея сортировки «сдваиванием» заключается в применении каскадной схемы слияния к упорядоченным последовательностям.

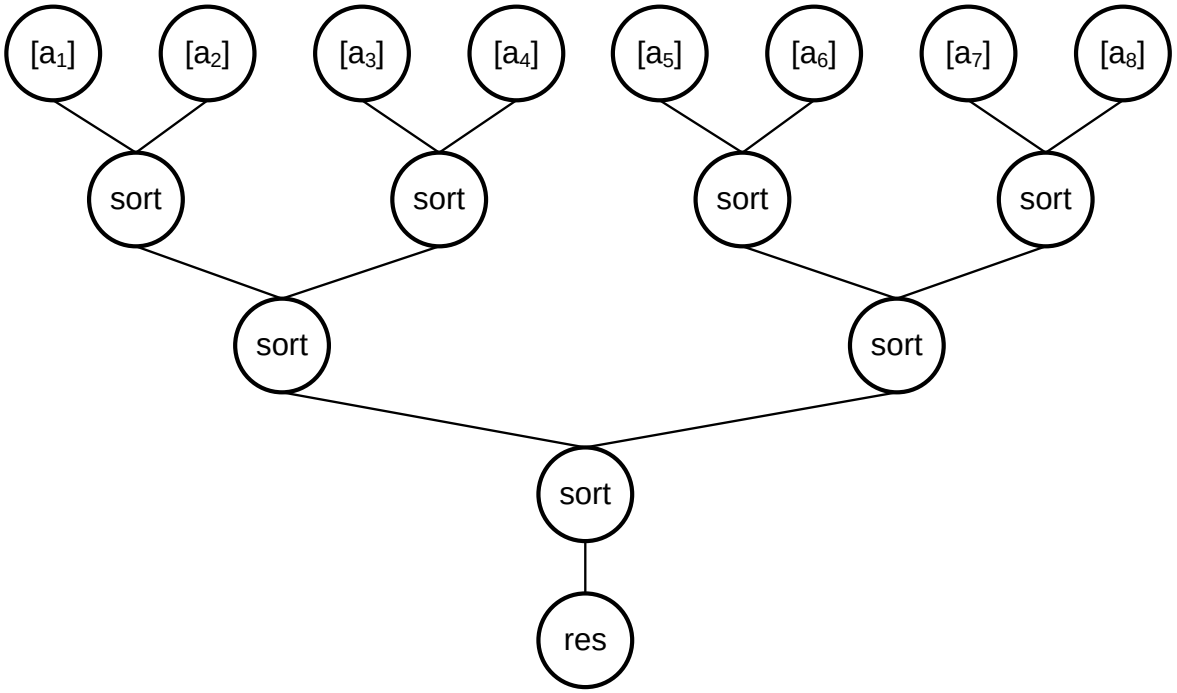

Рис. 7. Каскадная схема сортировки массива на 8-ми процессорах (сортировка «сдваиванием»)

К ее недостаткам можно отнести меньшую эффективность и больший объем пересылаемых данных на поздних этапах сортировки, однако в некоторых ситуациях она может оказаться более эффективна, чем чет-нечетная сортировка.

Таким образом, выбор определенного алгоритма сортировки тесно связан с особенностями вычислительной системы.

Реализация алгоритма

Java

Класс, реализующий быструю сортировку.

```
import java.util.Random;

public class QSort {

	private int[] res;
	private Random rnd;

	//Метод для сравнения чисел
	private boolean cmp(int x, int y) {
		if(x > y)
			return false;
		if(x < y)
			return true;
		return rnd.nextBoolean();
	}

	//Метод для разбиения элементов текущей подзадачи на две группы
	private int partition(int l, int r) {
		int i, q = l - 1, tmp;

		//Обмен опорного элемента со случайным
		i = rnd.nextInt(r - l + 1) + l;
		tmp = res[i];
		res[i] = res[r];
		res[r] = tmp;

		//Разбиение элементов на две группы
		for(i = l; i < r; i++)
			if(cmp(res[i], res[r])) {
				tmp = res[++q];
				res[q] = res[i];
				res[i] = tmp;
			}

		//Обмен опорного элемента
		tmp = res[++q];
		res[q] = res[r];
		res[r] = tmp;

		//Позиция опорного элемента
		return q;
	}

	//Сортировка
	private void qsort(int l, int r) {
		if(l >= r)
			return;
		int q = partition(l, r);
		//Рекурсивный вызов для подзадач
		qsort(l, q - 1);
		qsort(q + 1, r);
	}

	//Метод получает массив для сортировки
	SortResult Calc(int[] arg0) {
```

```
		//Замер времени
		long time = System.currentTimeMillis();

		//Сортировка
		res = arg0;
		rnd = new Random();
		qsort(0, res.length - 1);

		return new SortResult(res, System.currentTimeMillis() - time);
	}
}
```

Класс для параллельной реализации алгоритма. Каждый экземпляр запускается в отдельном потоке.

```
import java.util.Random;

public class QSortTh implements Runnable{

	Thread th;
	private int left, right, recursionLevel;
	private QSortParallel data;
	private Random rnd;

	//Конструктор принимает ссылку на данные, размер массива и требуемый уровень рекурсии
	QSortTh(QSortParallel data, String thName, int recursionLevel, int left, int right) {
		this.data = data;
		this.left = left;
		this.right = right;
		this.recursionLevel = recursionLevel;
		rnd = new Random();

		//Вычисления выполняются в отдельном потоке
		th = new Thread(this, thName);
		th.start();
	}

	//Метод для сравнения чисел
	private boolean cmp(int x, int y) {
		if(x > y)
			return false;
		if(x < y)
			return true;
		return rnd.nextBoolean();
	}

	//Метод для разбиения элементов текущей подзадачи на две группы
	private int partition(int left, int right) {
		int i, q = left - 1, tmp;

		//Обмен опорного элемента со случайным
		i = rnd.nextInt(right - left + 1) + left;
		tmp = data.x[i];
		data.x[i] = data.x[right];
		data.x[right] = tmp;

		//Разбиение элементов на две группы
		for(i = left; i < right; i++)
			if(cmp(data.x[i], data.x[right])) {
				tmp = data.x[++q];
				data.x[q] = data.x[i];
				data.x[i] = tmp;
```

```
                }

                //Обмен опорного элемента
                tmp = data.x[++q];
                data.x[q] = data.x[right];
                data.x[right] = tmp;

                //Позиция опорного элемента
                return q;
        }

        //Сортировка
        private void qSort(int left, int right) {
                if(left < right) {
                        int q = partition(left, right);
                        //Рекурсивный вызов для подзадачи либо в новом потоке, либо в текущем
                        if(recursionLevel > 0) {
                                QSortTh a = new QSortTh(data, String.valueOf(recursionLevel) + "0",
recursionLevel - 1, left, q - 1);
                                QSortTh b = new QSortTh(data, String.valueOf(recursionLevel) + "1",
recursionLevel - 1, q + 1, right);
                                try {
                                        a.th.join();
                                        b.th.join();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                        else {
                                qSort(left, q - 1);
                                qSort(q + 1, right);
                        }
                }
        }

        //Запуск сортировки
        public void run() {
                qSort(left, right);
        }

}
```

Класс, реализующий параллельную сортировку.

```
public class QSortParallel {

        int[] x;

        //Метод получает массив и требуемый уровень рекурсии
        SortResult Calc(int[] arg0, int recursionLevel) {
                //Замер времени
                long time = System.currentTimeMillis();
                x = arg0;

                //Вызов сортировки
                QSortTh a = new QSortTh(this, "master", recursionLevel, 0, x.length - 1);

                try {
                        //Ожидание завершения сортировки
                        a.th.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
```

```
                }

                //Результат и время работы
                return new SortResult(x, System.currentTimeMillis() - time);
            }
        }
```

Сравнение быстродействия

Алгоритм быстрой сортировки

Intel Core Duo

Java			
n	Один поток	Два потока	Коэффициент
10000	15	15	1,0000
100000	31	32	0,9688
1000000	405	297	1,3636
10000000	4127	3106	1,3287
30000000	17589	12628	1,3929

Intel Atom

Java			
n	Один поток	Два потока	Коэффициент
10000	31	15	2,0667
100000	62	63	0,9841
1000000	641	531	1,2072
10000000	6906	6031	1,1451
30000000	21250	18109	1,1734

Алгоритм модифицированной «пузырьковой» сортировки

Intel Core Duo

C++			
	Один поток	MPI	
n			Коэффициент
10000	16	16	1,0000
100000	16	31	0,5161
1000000	211	237	0,8903
10000000	2591	2148	1,2062
30000000	8131	6927	1,1738

Intel Atom

C++			
	Один поток	MPI	
n			Коэффициент
10000	16	31	0,5161
100000	16	16	1,0000
1000000	315	289	1,0900
10000000	4078	3641	1,1200
30000000	14870	12956	1,1477

Задача умножения матриц

Постановка задачи и способы ее решения

Умножение матриц - операция вычисления матрицы

$$AB = C, \tag{11}$$

элементы которой равны сумме произведений элементов в соответствующей строке первого множителя и столбце второго.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{12}$$

В первом множителе должно быть столько же столбцов, сколько строк во втором, иначе операция умножения не определена.

Если матрица A имеет размерность $n \times m$, B - $m \times k$, то размерность их произведения C есть $n \times k$.

Перемножение двух матрицы можно выполнять по определению. Каждый элемент результирующей матрицы будет вычисляться по формуле (12).

Очевидно, что сложность такого алгоритма $O(nmk)$, так как нам необходимо получить nk элементов, для получения каждого из которых необходимо выполнить m умножений и $m-1$ сложений.

В частном случае будем рассматривать умножение квадратных матриц. Сложность такого алгоритма составляет $O(n^3)$.

Перемножение двух прямоугольных матриц можно свести к перемножению двух квадратных матриц, добавив вместо недостающих элементов нули.

Предложенный алгоритм легко распараллеливается на системах с общей памятью – каждый поток выполняет вычисление какого-либо одного или группы элементов. При этом не происходит изменение элементов исходных матриц, а запись ответа всегда осуществляется в разные области памяти, поэтому не требуется синхронизация по доступу к данным.

В случае использования описанного алгоритма в системах с разделенной памятью возникают трудности – избыточность пересылок данных (так как для вычисления элемента процессору необходимо иметь соответствующие строку и столбец исходных матриц). В таком случае возможна ситуация, при которой время, необходимое на пересылку данных, может превысить время, необходимое на вычисления.

Данную проблему устраняет схема ленточного умножения, при которой процессор, получая группу строк и группу столбцов, вычисляет частичный результат – прямоугольную подматрицу результирующей матрицы.

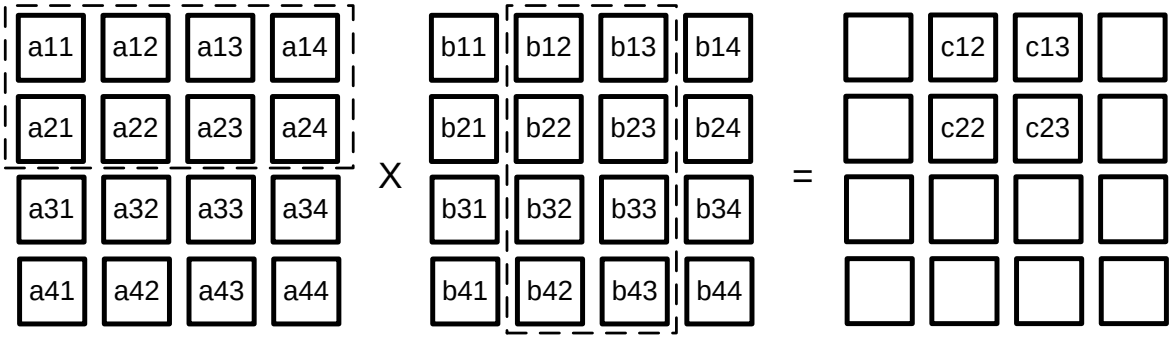

Рис. 8. Схема ленточного умножения

Данная схема может быть модифицирована таким образом, что на каждом процессоре хранится строка первой матрицы и столбец второй, по завершении вычисления каждым процессором очередного элемента выполняется циклический обмен столбцов между процессорами.

Такой подход предпочтительнее в случае, когда исходные матрицы имеют большой размер и не помещаются в память целиком.

Для умножения двух матриц при использовании ленточной схемы необходимо число операций порядка $\frac{n^3}{p}$, поэтому ускорение алгоритма составит p. Коммуникационные затраты составляют порядка $\alpha p + \beta n^2$, где α, β некоторые параметры сети.

Широкое распространение получил блочный алгоритм умножения матриц, который заключается в декомпозиции исходных матриц на четыре части и получения результирующей матрицы с помощью вычисления произведения декомпозиций. Данный метод будет рассмотрен ниже как основа для вывода более эффективного алгоритма.

Также в системах с разделенной памятью часто применяются алгоритмы Фокса и Кэннона.

Оба этих алгоритма являются модификацией блочного алгоритма умножения матриц и, также как и ленточный алгоритм умножения, обладают ускорением p, однако отличаются схемой межпроцессорных коммуникаций.

Задачу перемножения матриц можно решать с помощью рекурсивного алгоритма, разработанного Штрассеном. Время работы этого алгоритма составляет $O(n^{\log_2 7})$.

Идея алгоритма базируется на методе "разделяй и властвуй".

Предположим, что необходимо перемножить две матрицы размером $n \times n$ каждая. Считая, что n является точной степенью двойки, можно поделить матрицы на четыре части.

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}. \tag{13}$$

Каждое уравнение требует двух умножений и одного сложения матриц размером $n/2 \times n/2$.

$$r = ae + bg \tag{14}$$
$$s = af + bh \tag{15}$$
$$t = ce + dg \tag{16}$$
$$u = cf + dh \tag{17}$$

Используя эти уравнения как определение простейшего рекурсивного алгоритма, получается следующее рекуррентное соотношение для оценки времени работы полученного алгоритма.

$$\varphi(n) = 8\varphi(\frac{n}{2}) + cn^2. \tag{18}$$

К сожалению, рекуррентное соотношение (18) имеет решение $O(n^3)$, так что данный метод не быстрее обычного умножения матриц.

Штрассен разработал рекурсивный алгоритм, который требует семи умножений матриц размера $n/2 \times n/2$ и конечное число скалярных сложений и умножений, что приводит к рекуррентному соотношению

$$\varphi(n) = 7\varphi(\frac{n}{2}) + cn^2, \tag{19}$$

которое имеет решение $O(n^{\log_2 7})$.

Метод Штрассена состоит из четырех шагов.

1. Разделяем подматрицы A и B на подматрицы размером $n/2 \times n/2$.

2. Используя конечное число скалярных сложений и умножений, вычисляем четырнадцать матриц $A_1, B_1, A_2, B_2, ..., A_7, B_7$, каждая из которых имеет размер $n/2 \times n/2$.

3. Рекурсивно вычисляем семь матричных произведений $P_i = A_i B_i$ для $i = 1, 2, ..., 7$.

4. Вычисляем матрицы r, s, t и u результирующей матрицы C путем сложения и/или вычитания различных комбинаций матриц P_i с использованием конечного числа скалярных умножений и вычитаний.

Предположим, что каждое матричное произведение P_i можно записать в виде

$$P_i = A_i B_i = (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h), \qquad (20)$$

где коэффициенты α_{ij}, β_{ij} принимают значения из множества $\{-1, 0, 1\}$. Т.е. каждое произведение вычисляется при помощи сложения или вычитания некоторых из подматриц A, сложения или вычитания некоторых из подматриц B, и перемножения полученных результатов.

Описанный вид произведения можно использовать без оглядки на некоммутативность умножения матриц, так как в каждом произведении все подматрицы A оказываются слева, а все подматрицы B - справа.

Для удобства представления линейных комбинаций произведений подматриц можно воспользоваться матрицами размера 4×4. Например, можно записать уравнение (14) как

$$r = ae + bg = \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} + & . & . & . \\ . & . & + & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}. \qquad (21)$$

В последнем выражении используется сокращенная запись, где «+» представляет «+1», «.» представляет «0», а «-» - «-1».

Используя данные обозначения, можно получить уравнения для остальных подматриц, составляющих результирующую матрицу C.

$$s = af + bh = \begin{pmatrix} . & + & . & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, \qquad (22)$$

$$t = ce + dg = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & . & + & . \end{pmatrix}, \qquad (23)$$

$$u = cf + dh = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & + & . & . \\ . & . & . & + \end{pmatrix}, \qquad (24)$$

Подматрицу s можно вычислить как $s = P_1 + P_2$, где матрицы P_1 и P_2 вычисляются с использованием одного матричного произведения каждая:

$$P_1 = A_1 B_1 = a(f - h) = af - ah = \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \quad (25)$$

$$P_2 = A_2 B_2 = (a + b)h = ah + bh = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad (26)$$

Матрица t может быть вычислена как $t = P_3 + P_4$, где

$$P_3 = A_3 B_3 = (c + d)e = ce + de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix} \quad (27)$$

и

$$P_4 = A_4 B_4 = d(g - e) = dg - de = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix} \quad (28)$$

$$P_5 = A_5 B_5 = (a + d) \cdot (e + h) = ae + ah + de + dh = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} \quad (29)$$

$$P_5 + P_4 - P_2 = ae + dh + dg - bh = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix} \quad (30)$$

$$P_6 = A_6 B_6 = (b - d) \cdot (g + h) = bg + bh - dg - dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix} \quad (31)$$

$$r = P_5 + P_4 - P_2 + P_6 = ae + bg = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad (32)$$

$$P_5 + P_1 - P_3 = ae + af - ce + dh = \begin{pmatrix} + & + & . & . \\ . & . & . & . \\ - & . & . & . \\ . & . & . & + \end{pmatrix} \quad (33)$$

$$P_7 = A_7 B_7 = (a - c) \cdot (e + f) = ae + af - ce - cf = \begin{pmatrix} + & + & . & . \\ . & . & . & . \\ - & - & . & . \\ . & . & . & . \end{pmatrix} \quad (34)$$

$$u = P_5 + P_1 - P_3 - P_7 = cf + dh = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & + & . & . \\ . & . & . & + \end{pmatrix} \quad (35)$$

На практике реализации быстрого умножения плотных матриц с использованием алгоритма Штрассена имеют «точку пересечения» с обычным алгоритмом умножения, и переключаются на использование простого алгоритма при перемножении матриц с размером, меньшим точки пересечения.

Отдельно стоит отметить, что на настоящий момент существуют асимптотически более эффективные методы перемножения матриц, одним из которых является, например, алгоритм Винограда. Однако в данной работе эти методы не рассматриваются.

Ниже будет приведена реализации стандартного алгоритма умножения матриц по определению, параллельная реализация стандартного алгоритма для систем с общей памятью, ленточного умножения матриц с использованием MPI, реализация алгоритма Штрассена для систем с общей памятью.

Реализация алгоритма

OpenMP

Реализация алгоритма по определению в последовательном и параллельном режимах.

```
#define _CRT_SECURE_NO_DEPRECATE
#include <iostream>
//Подключаем OpenMP
#include <omp.h>
#include <cmath>
#include <algorithm>
#include <time.h>
//Определяем размерность матриц
#define N 2000
#pragma comment (linker,"/STACK:64000000")
using namespace std;
//Определяем массивы. Пусть будет переполнение – нас интересует только время работы
int x[N][N] = {0};
int y[N][N] = {0};
int res1[N][N] = {0};
int res2[N][N] = {0};
int main()
{
	freopen("input.txt","r",stdin);
	freopen("output.txt","w",stdout);

	int i, j, k;
	long long time1, time2, tmp;

	//Инициализируем массивы
	for(i = 0; i < N; i++)
		for(j = 0; j < N; j++)
		{
			x[i][j] = i * N + j;
			y[i][j] = j * N + i;
		}

	//Замеряем время старта
	clock_t start = clock();
	//Выполняем последовательное умножение
	for(i = 0; i < N; i++)
		for(j = 0; j < N; j++)
		{
			tmp = 0;
			for(k = 0; k < N; k++)
				tmp += x[i][k] * y[k][j];
			res1[i][j] = tmp;
		}

	//Замеряем время работы
	time1 = clock() - start;

	start = clock();

	//Параллельный алгоритм
	#pragma omp parallel shared(res2, x, y) private(i, j, k, tmp)
	{
		for(i = 0; i < N; i++)
```

```
                        #pragma omp for
                        for(j = 0; j < N; j++)
                        {
                                tmp = 0;
                                for(k = 0; k < N; k++)
                                        tmp += x[i][k] * y[k][j];
                                res2[i][j] = tmp;
                        }
        }

        //Замеряем время работы
        time2 = clock() - start;

        //Проверяем, чтобы результаты совпали
        int flag = 0;
        for(i = 0; i < N; i++)
        {
                for(j = 0; j < N; j++)
                        if(res1[i][j] != res2[i][j])
                        {
                                flag = 1;
                                break;
                        }
                if(flag)
                        break;
        }

        //Выводим либо предупреждение, либо время работы
        if (flag)
                cout << "Bad answer";
        else
                cout << time1 << "\n" << time2;

        return 0;
}
```

MPI

```
//Подключаем MPI
#include "mpi.h"
#include <stdio.h>
//Определяем размерности матриц
#define N 1000
#define M 1000
#define K 1000
int main(int argc, char *argv[])
{
	int numtasks, taskid, numworkers, source, dest, rows, averow, extra, offset, i, j, k, rc;
	//Объявление матриц
	int a[N][M], b[M][K], c[N][K];
	//Переменные для времени работы
	double time1;
	MPI_Status status;
	rc = MPI_Init(&argc,&argv);
	rc |= MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
	rc |= MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
	//Проверяем на корреткность
	if (rc != MPI_SUCCESS)
		printf ("Error");
	numworkers = numtasks - 1;
	//Если мастер
	if (taskid == 0)
	{
		//Инициализируем массивы
		for (i=0; i<N; i++)
			for (j=0; j<M; j++)
				a[i][j]= i + j;
		for (i=0; i<M; i++)
			for (j=0; j<K; j++)
				b[i][j] = i * j;
		//Делим на подзадачи
		averow = N/numworkers;
		extra = N%numworkers;
		offset = 0;
		//Замеряем время
		time1 = MPI_Wtime();
		//Раздаем подзадачи
		for (dest=1; dest<=numworkers; dest++)
		{
			if(dest <= extra)
				rows = averow + 1;
			else
				rows = averow;
			rows = (dest <= extra) ? averow + 1 : averow;
			MPI_Send(&offset, 1, MPI_INT, dest, FROM_0, MPI_COMM_WORLD);
			MPI_Send(&rows, 1, MPI_INT, dest, FROM_0, MPI_COMM_WORLD);
			MPI_Send(&a[offset][0], rows*M, MPI_DOUBLE, dest, FROM_0,
MPI_COMM_WORLD);
			MPI_Send(&b, M*K, MPI_DOUBLE, dest, FROM_0, MPI_COMM_WORLD);
			offset += rows;
		}
		//Принимаем решения
		for (source = 1; source <= numworkers; source++)
		{
			MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
			MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
			MPI_Recv(&c[offset][0], rows*K, MPI_DOUBLE, source, 2, MPI_COMM_WORLD,
&status);
```

```
                }
                //Выводим время работы
                printf ("%.3lf\n", MPI_Wtime() - time1);
        }
        //Если не мастер
        if (taskid > 0)
        {
                //Получаем задачу
                MPI_Recv(&offset, 1, MPI_INT, 0, FROM_0, MPI_COMM_WORLD, &status);
                MPI_Recv(&rows, 1, MPI_INT, 0, FROM_0, MPI_COMM_WORLD, &status);
                MPI_Recv(&a, rows*M, MPI_DOUBLE, 0, FROM_0, MPI_COMM_WORLD, &status);
                MPI_Recv(&b, M*K, MPI_DOUBLE, 0, FROM_0, MPI_COMM_WORLD, &status);
                //Решаем ее
                for (k = 0; k < K; k++)
                        for (i = 0; i < rows; i++)
                        {
                                c[i][k] = 0;
                                for (j = 0; j < M; j++)
                                        c[i][k] += a[i][j] * b[j][k];
                        }
                //Возвращаем ответ
                MPI_Send(&a_offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
                MPI_Send(&a_rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
                MPI_Send(&c, a_rows*K, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
        }
        //Конец программы
        MPI_Finalize();
}
```

Java

Класс, реализующий последовательный алгоритм умножения матриц.

```
public class Matrix {

	//На вход метод получает два матрицы, произведение которых необходимо найти
	MatrixResult Calc(int[][] arg0, int[][] arg1) {
		long[][] res = new long[arg0.length][arg1[0].length];
		//Замеряем время работы
		long time = System.currentTimeMillis();

		//Выполняем унможение по пределению
		for(int i = 0; i < arg0.length; i++)
			for(int j = 0; j < arg1[0].length; j++)
				for(int k = 0; k < arg0[0].length; k++)
					res[i][j] += arg0[i][k] * arg1[k][j];

		//Возвращаем ответ
		return new MatrixResult(res, System.currentTimeMillis() - time);
	}

}
```

Класс, для параллельного перемножения матрицы. Каждый экземпляр запускается в отдельном потоке.

```
public class MatrixTh implements Runnable {

	Thread th;
	int[][] x, y;
	//Матрица частичного результата
	long[][] res;

	//Конструктор принимает исходные матрицы
	MatrixTh(int[][] x, int[][] y, String thName) {
		res = new long[x.length][y[0].length];
		this.x = x;
		this.y = y;

		//Каждый экземпляр запускается в отдельном потоке
		th = new Thread(this, thName);
		th.start();
	}

	//Вычисление результата
	public void run() {
		for(int i = 0; i < x.length; i++)
			for(int j = 0; j < y[0].length; j++)
				for(int k = 0; k < x[0].length; k++)
					res[i][j] += x[i][k] * y[k][j];
	}

}
```

Класс для параллельного решения задачи.

```
import java.util.Arrays;

public class MatrixParallel {

	//Метод принимает исходные матрицы и требуемое число потоков
	MatrixResult Calc(int[][] x, int[][] y, int thCount) {
		//Замер времени старта
		long time = System.currentTimeMillis();
		//Выделение памяти под дочерние потоки
		MatrixTh[] th = new MatrixTh[thCount];

		//Вычисление отпимального размера подзадачи
		int next = 0, d;
		if(x.length % thCount != 0)
			d = x.length / thCount + 1;
		else
			d = x.length / thCount;

		//Разбинение на подзадачи
		for(int i = 0; i < thCount; i++) {
			th[i] = new MatrixTh(Arrays.copyOfRange(x, next, Math.min(next + d, x.length)), y,
String.valueOf(i));
			next += d;
		}

		//Ожидание завершения дочерних потоков
		try {
			for(int i = 0; i < thCount; i++)
				th[i].th.join();
		} catch (InterruptedException e) {
			System.err.print("Ошибка выполнения");
			System.err.flush();
			return new MatrixResult(null, 0);
		}

		//Получение окончательного результата
		long[][] res = new long[x.length][y[0].length];
		next = 0;
		for(int i = 0; i < thCount; i++) {
			for(int j = 0; j < th[i].res.length; j++)
				for(int k = 0; k < th[i].res[j].length; k++)
					res[next + j][k] = th[i].res[j][k];
			next += d;
		}

		return new MatrixResult(res, System.currentTimeMillis() - time);
	}
}
```

Класс, реализующий последовательную версию алгоритма Штрассена.

```
public class MatrixStrassen {

	//Вычитание матриц
	private int[][] sub(int[][] m1, int[][] m2) {
		int n = m1.length, i, j;
		int[][] res = new int[n][n];

		for(i = 0; i < n; i++)
			for(j = 0; j < n; j++)
```

```
                        res[i][j] = m1[i][j] - m2[i][j];

        return res;
}

//Сложение матриц
private int[][] add(int[][] m1, int[][] m2) {
        int n = m1.length, i, j;
        int[][] res = new int[n][n];

        for(i = 0; i < n; i++)
                for(j = 0; j < n; j++)
                        res[i][j] = m1[i][j] + m2[i][j];

        return res;
}

//Вычитание матриц
private long[][] sub(long[][] m1, long[][] m2) {
        int n = m1.length, i, j;
        long[][] res = new long[n][n];

        for(i = 0; i < n; i++)
                for(j = 0; j < n; j++)
                        res[i][j] = m1[i][j] - m2[i][j];

        return res;
}

//Сложение матриц
private long[][] add(long[][] m1, long[][] m2) {
        int n = m1.length, i, j;
        long[][] res = new long[n][n];

        for(i = 0; i < n; i++)
                for(j = 0 ; j < n; j++)
                        res[i][j] = m1[i][j] + m2[i][j];

        return res;
}

//Умножение матриц
private long[][] mult(int[][] x, int[][] y) {
        int n = x.length, i, j, k;
        long[][] res = new long[n][n];

        //Точка пересечения
        if(n <= 64) {
                for(i = 0; i < n; i++)
                        for(j = 0; j < n; j++)
                                for(k = 0; k < n; k++)
                                        res[i][j] += x[i][k] * y[k][j];
        }
        else {
        //Вычисление элементов согласно алгоритму
                int m = n/2;

                int[][] b = new int[m][m];
                for(i = 0; i < m; i++)
                        for(j = 0; j < m; j++)
                                b[i][j] = x[i][m + j];

                int[][] c = new int[m][m];
```

```
            for(i = 0; i < m; i++)
                        for(j = 0;j < m; j++)
                                    c[i][j] = x[m + i][j];

            int[][] f = new int[m][m];
            for(i = 0; i < m; i++)
                        for(j = 0; j < m; j++)
                                    f[i][j] = y[i][m + j];

int[][] g = new int[m][m];
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        g[i][j] = y[m + i][j];

int[][] A1 = new int[m][m];
for(i = 0; i < m; i++)
            for(j = 0;j < m; j++)
                        A1[i][j] = x[i][j];

int[][] A4 = new int[m][m];
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        A4[i][j] = x[m + i][m + j];

int[][] B3 = new int[m][m];
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        B3[i][j] = y[i][j];

int[][] B2 = new int[m][m];
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        B2[i][j] = y[m + i][m + j];

int[][] A2 = add(A1, b);
int[][] A3 = add(c, A4);
int[][] A5 = add(A1, A4);
int[][] A6 = sub(b, A4);
int[][] A7 = sub(A1, c);
int[][] B1 = sub(f, B2);
int[][] B4 = sub(g, B3);
int[][] B5 = add(B3, B2);
int[][] B6 = add(g, B2);
int[][] B7 = add(B3, f);
long[][] P1 = mult(A1, B1);
long[][] P2 = mult(A2, B2);
long[][] P3 = mult(A3, B3);
long[][] P4 = mult(A4, B4);
long[][] P5 = mult(A5, B5);
long[][] P6 = mult(A6, B6);
long[][] P7 = mult(A7, B7);
long[][] r = add(sub(add(P5, P4), P2), P6);
long[][] s = add(P1, P2);
long[][] t = add(P3, P4);
long[][] u = sub(sub(add(P1, P5), P3), P7);

//Получение результата
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        res[i][j] = r[i][j];
for(i = 0; i < m; i++)
            for(j = 0; j < m; j++)
                        res[i][m + j] = s[i][j];
```

```
                for(i = 0; i < m; i++)
                        for(j = 0; j < m; j++)
                                res[m + i][j] = t[i][j];
                for(i = 0; i < m; i++)
                        for(j = 0; j < m; j++)
                                res[m + i][m + j] = u[i][j];
                }
                return res;
        }

        MatrixResult Calc(int[][] x, int[][] y) {
                long time = System.currentTimeMillis();

                long[][] res = mult(x, y);

                return new MatrixResult(res, System.currentTimeMillis() - time);
        }

}
```

Сравнение быстродействия

Алгоритм умножения матриц по определению

Intel Core Duo

Java			
n	Один поток	Два потока	Коэффициент
100	31	15	2,0667
200	187	94	1,9894
500	3015	1547	1,9489
1000	27922	14357	1,9448
2000	257563	154797	1,6639

C++					
	Один поток	OpenMP		MPI	
n			Коэффициент		Коэффициент
100	0	16	1,0000	16	1,0000
200	32	15	2,1333	31	1,0323
500	704	312	2,2564	607	1,1598
1000	7344	4031	1,8219	5814	1,2632
2000	60453	34047	1,7756	47615	1,2696

Intel Atom

Java			
n	Один поток	Два потока	Коэффициент
100	47	46	1,0217
200	453	296	1,5304
500	8063	5016	1,6075
1000	72625	50172	1,4475
2000	1962062	1196938	1,6392

C++					
	Один поток	OpenMP		MPI	
n			Коэффициент		Коэффициент
100	16	15	1,0667	31	0,5161
200	156	157	0,9936	286	0,5455
500	4484	2735	1,6395	4011	1,1179
1000	43219	24578	1,7584	38168	1,1323
2000	364907	244513	1,4924	339357	1,0753

Алгоритм Штрассена для умножения матриц

Intel Core Duo

Java		
n	Алгоритм Штрассена	Параллельный алгоритм Штрассена
256	250	261
512	1578	1387
1024	10750	8916
2048	75062	59831

Intel Atom

Java		
n	Алгоритм Штрассена	Параллельный алгоритм Штрассена
256	531	483
512	3281	2671
1024	21953	16312
2048	152641	112319

Схема Горнера

Постановка задачи и способы ее решения

Схема Горнера – алгоритм вычисления значения многочлена, записанного в виде суммы мономов, при заданном значении переменной.

Метод Горнера позволяет найти корни многочлена, а также вычислить производные полинома в заданной точке.

Схема Горнера также является простым алгоритмом для деления многочлена на бином вида $x-c$.

Пусть задан многочлен

$$P(x)=a_0+a_1x+a_2x^2+a_3x^3+...+a_nx^n, a_i\in R. \quad (36)$$

Перепишем его в следующем виде

$$P(x)=a_0+x(a_1+x(a_2+...x(a_{n-1}+a_nx)...)). \quad (37)$$

Тогда значение многочлена в точке может быть легко вычислено путем выполнения операций, начиная с внутренних скобочек.

Описанный метод называется схемой Горнера.

Для реализации описанного алгоритма в параллельном режиме можно использовать следующую декомпозицию:

Пусть имеется p процессоров.

«Разобьем» исходный многочлен (36) на p многочленов вида

$$\begin{aligned} P_1(x)&=a_0+a_px^p+a_{2p}x^{2p}+a_{3p}x^{3p}... \\ P_2(x)&=a_1+a_{p+1}x^p+a_{2p+1}x^{2p}+a_{3p+1}x^{3p}... \\ P_3(x)&=a_2+a_{p+2}x^p+a_{2p+2}x^{2p}+a_{3p+2}x^{3p}... \\ &..... \\ P_p(x)&=a_{p-1}+a_{2p-1}x^p+a_{3p-1}x^{2p}+a_{4p-1}x^{3p}... \end{aligned} \quad (38)$$

Вычислим значение многочленов (38) в заданной точке по описанной выше схеме.

Очевидно, что значение исходного многочлена в этой же точке может быть получено как

$$P(x)=P_1(x)+xP_2(x)+x^2P_3(x)+...+x^{p-1}P_p(x) \quad (39)$$

При декомпозиции задачи на p процессоров необходимо выполнить $\frac{n}{p}$ параллельных операций для вычисления значений многочленов (38) и p последовательных операций для объединения результата.

Рассмотрим функцию

$$f(p) = \frac{n}{p} + p \tag{40}$$

Найдем экстремум функции. Исходя из

$$f'(p) = -\frac{n}{p^2} + 1 = 0 \tag{41}$$

минимум достигается в точке $p = \sqrt{n}$.

Таким образом, наибольший эффект достигается при $p = \sqrt{n}$. В таком случае для вычисления значения многочлена в точке понадобится порядка $2\sqrt{n}$ операций, а коммуникационные затраты составят порядка $\alpha n + \beta\sqrt{n}$, где α, β - некоторые параметры сети.

Описанный алгоритм может быть обобщен для использования каскадной или модифицированной каскадной схем.

В общем случае значительного эффекта от параллельного вычисления значения многочлена в точке можно добиться при работе с большими числами.

Сравнение быстродействия

Intel Core Duo

Java			
n	Один поток	Два потока	Коэффициент
1000000	31	15	2,0667
10000000	31	32	0,9688
100000000	135	107	1,2617
1000000000	1210	815	1,4847
10000000000	11954	7648	1,5630

C++			
	Один поток	MPI	
n			Коэффициент
1000000	15	16	0,9375
10000000	16	16	1,0000
100000000	64	65	0,9846
1000000000	684	459	1,4902
10000000000	6321	3987	1,5854

Intel Atom

Java			
n	Один поток	Два потока	Коэффициент
1000000	31	15	2,0667
10000000	31	32	0,9688
100000000	174	121	1,4380
1000000000	1513	1076	1,4061
10000000000	16579	10895	1,5217

C++			
	Один поток	MPI	
n			Коэффициент
1000000	16	31	0,5161
10000000	16	16	1,0000
100000000	98	176	0,5568
1000000000	764	653	1,1700
10000000000	7944	6418	1,2378

Заключение

В результате работы были рассмотрены несколько типичных задач, а также проанализированы способы их решения в последовательном и параллельном режимах.

Разработка параллельного алгоритма даже на основе существующего последовательного является сложной задачей, которая требует от программиста знаний как средств и технологий программирования, так и архитектуры вычислительных систем и математики.

Для достижения наибольшей эффективности алгоритм решения задачи всегда должен разрабатываться с учетом особенностей системы, на которой он будет выполняться.

Литература

1. Кормен Т. Х., Лейзерсон Ч. И., Риверст Р. Л., Штайн К. «Алгоритмы: построение и анализ» - Вильямс, 2005 г., 1296 с.
2. Воеводин В.В., Воеводин В. В. «Параллельные вычисления» - СПб.: БХВ – Петербург, 2002 г., 608 с.
3. Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д. «Структуры данных и алгоритмы» - Вильямс, 2000 г., 384 с.
4. Баканов В. М., Осипов Д. В. «Параллельное программирование в стандарте MPI» - Москва, 2006 г., 78 с.
5. ru.wikipedia.org