МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ М. В. ЛОМОНОСОВА
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ
КАФЕДРА ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ

Смирнов Дмитрий Константинович

Оценки времени работы алгоритма циклической пересылки на суперкомпьютере "Ломоносов-2"

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
профессор, д.ф.-м.н.
Черепнёв Михаил Алексеевич

Москва, 2023

Оглавление

1. Введение

Стойкость множества криптографических систем и протоколов основана на сложности факторизации больших целых чисел. Существует множество алгоритмов факторизации целых чисел, но до сих пор не известно полиномиального алгоритма, не использующего квантовые вычисления (как, например, алгоритм Шора). Самые быстрые алгоритмы на сегодняшний день относятся к классу субэкспоненциальных. Одним из таких является, например, метод квадратичного решета. В процессе его работы возникает необходимость решения большой (длина вектора $X \geqslant 2^{20}$ элементов) разреженной системы линейных уравнений над конечным полем.

Одним из наиболее часто упоминаемых в настоящее время методов решения таких систем является метод Ланцоша. Несмотря на то, что использование блочных версий алгоритма Ланцоша улучшает масштабируемость параллельных программ, они не позволяют в полной мере использовать мощности вычислительных систем с производительностью более 100 TFlops. В работе [1] был предложен универсальный блочный метод Ланцоша–Паде, который решает проблему масштабируемости, удобный для реализации на параллельных вычислительных системах. Особую роль в этом методе играет алгоритм циклической пересылки.

Эта работа посвящена вопросу корректной реализации алгоритма циклической пересылки с последующими оценками времени работы этого алгоритма на суперкомпьютере "Ломоносов-2".

2. Цель и значение работы

Ранее алгоритм циклической пересылки уже рассматривался в ряде работ (например, [2, 3]). Он решает следующую задачу (All-reduce): имеется n вычислительных процессов, каждый из них владеет вектором длины m. Необходимо как можно быстрее получить сумму всех векторов так, чтобы в конечном итоге эта сумма оказалась на всех узлах.

Уже существуют алгоритмы, решающую указанную задачу, например, библиотека Open MPI содержит функцию MPI_Allreduce. Цель работы в более глубоком анализе алгоритма циклической пересылки и сравнении времени его работы со стандартным решением в бибилиотеке Open MPI.

3. Алгоритм

3.1. Описание одномерного алгоритма

Рассмотрим вариант алгоритма для одномерной решётки. Пусть число n – количество вычислительных процессов. На каждом процессе хранится некоторый вектор длины m. Каждый вектор делится на n блоков таким образом, что $n-1$ блок имеют длину $\lceil \frac{m}{n} \rceil$ и один блок имеет длину $m - \lceil \frac{m}{n} \rceil \cdot (n-1)$. Обозначим i-тый блок на j-том процессе за B_i^j.

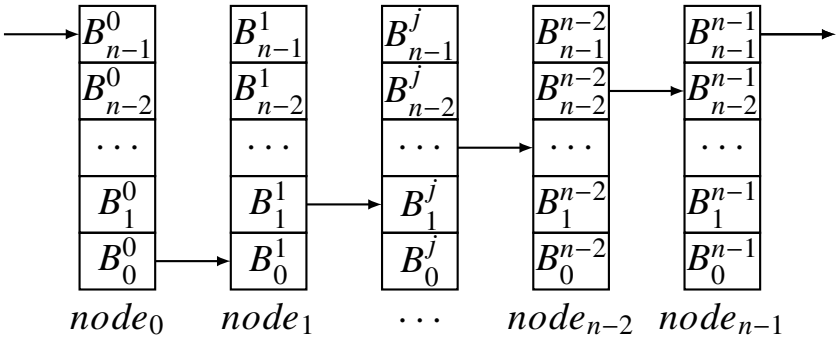

Рис. 1: Схема работы первого шага алгоритма

Алгоритм 3.1: Алгоритм циклической пересылки (Шаг первый, сложение)

```
/* Выполняется одновременно на каждом процессе. Номер
   текущего процесса - j. Номера вычисляются по модулю
   n.                                                  */
```

1 **от** $i \leftarrow 0$ **до** $n-1$ **цикл**
2 отправить B_i^j на узел $j+1$;
3 принять B_{i-1}^{j-1};
4 $B_{i-1}^j \leftarrow B_{i-1}^j + B_{i-1}^{j-1}$;

В результате первого шага алгоритма получаем вектор – сумму n векторов – распределённо хранящийся на процессах, как показано на рисунке 2. После этого осталось переслать этот вектор на все процессы. Для этого применим те же действия, заменив сложение присваиванием и сдвинув пределы итерирования.

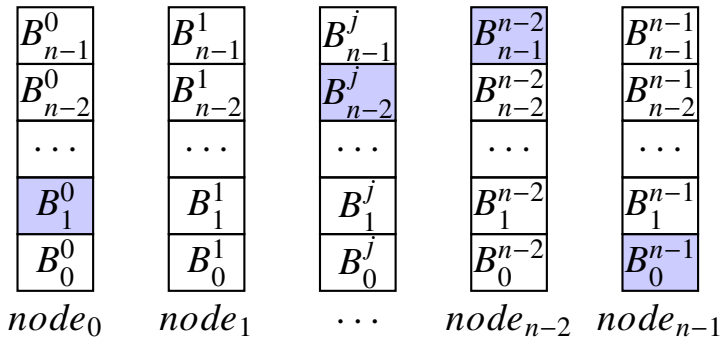

Рис. 2: Результат первого шага алгоритма

Алгоритм 3.2: Алгоритм циклической пересылки (Шаг второй, распределение)

```
/* Выполняется одновременно на каждом процессе. Номер
   текущего процесса - j. Номера вычисляются по модулю
   n.                                                  */
```

1 **от** $i \leftarrow 1$ **до** n **цикл**
2 отправить B_i^j на узел $j+1$;
3 принять B_{i-1}^{j-1};
4 $B_{i-1}^j \leftarrow B_{i-1}^{j-1}$;

Теперь на всех процессах хранится сумма входных векторов.

3.2. Построение многомерного алгоритма

Рассмотрим теперь работу алгоритма для d-мерной решётки. Пусть r_i — длина i-той размерности, $N = r_1 r_2 \cdots r_d$ — количество всех процессов. Эти процессы можно расположить на решётке $r_1 \times r_2 \times \ldots \times r_d$. Пусть у каждого из них есть набор координат на указанной решётке вида $(c_1, c_2, \ldots, c_d)$.

Построение многомерного алгоритма сводится к выполнению d одномерных алгоритмов следующим образом:

- Фиксируем все координаты, кроме первой. Проводим одномерный алгоритм, используя нумерацию процессов по первой координате. При этом каждый процесс отправляет свою часть процессу с координатами $((c_1 + 1)\%r_1, c_2, \ldots, c_d)$, а получает от процесса с координатами $((r_1 + c_1 - 1)\%r_1, c_2, \ldots, c_d)$.

- Далее в пересылке будут участвовать только части исходного вектора, содержащие сумму, они имеют длину $\frac{m}{r_1}$. Фиксируем все координаты, кроме второй. Делим на новые блоки указанную часть, эти блоки отправляются на процессы с координатами $(c_1, (c_2 + 1)\%r_2, \ldots, c_d)$, получаются от процессов с координатами $(c_1, (r_2 + c_2 - 1)\%r_2, \ldots, c_d)$.

- Указанные действия повторяются d раз. В результате у каждого процесса есть часть искомой суммы длины $\frac{m}{N}$. Для распределения проведём второй шаг одномерного алгоритма, проходя по координатам в обратном порядке.

3.3. Анализ сложности алгоритма

Введём некоторые обозначения. α — латентность сети, β — затраты на передачу одного элемента вектора, γ — затраты на сложение одного элемента вектора.

Латентность является временем, необходимым для того, чтобы начать передачу информации по сети. Значит, время на передачу блока информации равно $\alpha + \beta x$, где x — размер этого блока.

Рассмотрим вариант алгоритма для одномерной решётки. Векторы разбиваются на n блоков. Далее блоки каждого ряда пересылаются на соседний узел, складываются и пересылаются дальше. Это происходит $n - 1$ раз. На втором шаге повторяются те же действия в обратном порядке, но уже без сложения. Следовательно, получим:

$$T_1(n, m) = (2\alpha + (2\beta + \gamma)\frac{m}{n})(n - 1) = 2\alpha(n - 1) + (2\beta + \gamma)(1 - \frac{1}{n})m.$$

При увеличении количества процессов сложность будет асимптотически равна прямой с наклоном $k = 2\alpha$ и сдвигом $b = (2\beta + \gamma)m - 2\alpha$:

$$k = \lim_{n\to\infty} \frac{T_1(n, m)}{n} = \lim_{n\to\infty} \left(2\alpha + \frac{(2\beta + \gamma)m - 2\alpha}{n} - \frac{(2\beta + \gamma)m}{n^2}\right) = 2\alpha$$

$$b = \lim_{n\to\infty} (T_1(n, m) - kn) = \lim_{n\to\infty} \left(-2\alpha + (2\beta + \gamma)(1 - \frac{1}{n})m\right) = (2\beta + \gamma)m - 2\alpha$$

Поскольку на суперкомпьютере "Ломоносов-2" в качестве основной сети установлен FDR Infiniband [4], латентность которого составляет, в среднем,

2 мкс [5], наклон прямой существенной проблемы не представляет. Гораздо большее влияние на время работы алгоритма оказывает b.

А при увеличении размера вектора сложность будет асимптотически равна прямой с наклоном $k = (2\beta+\gamma)(1-\frac{1}{n})$ и сдвигом $b = 2\alpha(n-1)$. Отсюда можно сделать вывод, что выгоднее будет брать как можно меньшее число коммутирующих узлов.

Далее, для удобства переобозначим $\tilde{\alpha} = 2\alpha$, $\tilde{\beta} = 2\beta + \gamma$. Будем считать эти параметры разными для разных измерений решётки.

Используя описанный в разделе 3.2 метод, получим выражение для многомерной решётки:

$$T_d(r_1,\ldots,r_d,m) = T_1(r_1,m) + T_1(r_2,\frac{m}{r_1}) + \ldots + T_1(r_d,\frac{m}{r_1\ldots r_{d-1}}) = \\ = \sum_{i=1}^{d}(\tilde{\alpha}_i(r_i-1)) + mB_d(r_1,\ldots,r_{d-1}),$$

$$B_d(r_1,\ldots,r_{d-1}) = \sum_{i=1}^{d}\left(\tilde{\beta}_i\left(1-\frac{1}{\prod_{k=0}^{i-1}r_k}\right)\frac{1}{r_i}\right) = \\ = \tilde{\beta}_1 + \frac{\tilde{\beta}_2-\tilde{\beta}_1}{r_1} + \frac{\tilde{\beta}_3-\tilde{\beta}_2}{r_1r_2} + \ldots + \frac{\tilde{\beta}_d-\tilde{\beta}_{d-1}}{r_1r_2\ldots r_{d-1}} - \frac{\tilde{\beta}_d}{n}$$

Ранее был получен частный случай этой формулы в предположении $\tilde{\alpha} = \tilde{\alpha}_i, \tilde{\beta} = \tilde{\beta}_i, i = \overline{1,d}$ ([2], [3]):

$$T_d(r_1,\ldots,r_d,m) = \tilde{\alpha}\left(\sum_{i=1}^{d}r_i - d\right) + \tilde{\beta}(1-\frac{1}{n})m$$

Но на практике скорость передачи данных непостоянна. Обмен данными между двумя ядрами на одном процессоре происходит во много раз быстрее, чем между двумя узлами по сети. Таким образом, в условиях работы на одном кластере можно свести задачу к оптимизации алгоритма на двумерной решётке, где β_1, γ_1 — скорость передачи и сложения одного элемента вектора между ядрами одного узла, β_2 и γ_2 — между узлами в одной сети. При этом может возникнуть и обмен данными между несколькими кластерами, тогда следует применять алгоритм на трёхмерной решётке.

3.4. Выбор размеров решётки

Будем работать в двумерной решётке:

$$T_2(n, m) = \tilde{\alpha}\,(r_1 + r_2 - 2) + mB_2(r_1).$$

Нам требуется минимизировать второе слагаемое с функцией:

$$B_2(r_1) = \tilde{\beta}_1 + \frac{\tilde{\beta}_2 - \tilde{\beta}_1}{r_1} - \frac{\tilde{\beta}_2}{n},$$

при этом будем считать, что $\tilde{\beta}_i$ – скорость обмена данными по размерности длины r_i.

Найдём производную:

$$B_2{}'(r_1) = \frac{\tilde{\beta}_1 - \tilde{\beta}_2}{r_1^2}.$$

Если $\tilde{\beta}_1 > \tilde{\beta}_2$, то $B_2(r_1)$ – возрастающая функция, и тогда r_1 должно быть минимально возможным, иначе – максимально возможным.

В условиях работы на ”Ломоносов-2”, r_1 максимально может быть равным 14 – таково количество ядер на каждом узле [4].

4. Вариант без асинхронного сложения

4.1. Описание программы

Первый шаг программы реализован следующим образом. В цикле по номеру измерения определяются соседи в этом измерении решётки. Далее в цикле по длине измерения вычисляются начало и длина отрезков на отправку и получение (”сегменты”). Затем используется MPI_Sendrecv, позволяющий одновременно посылать одну часть вектора и принимать другую. Это блокирующая операция, что означает, программа не будет работать дальше, пока не завершатся все посылки. После этого производится сложение полученного блока вектора с соответствующим ему блоком вектора процесса. Вычисляется индекс блока, содержащего сумму по текущему измерению и запоминаются его начало и длина. Код описываемого фрагмента программы можно найти в приложении.

4.2. Результаты запусков на "Ломоносове-2"

На рисунке 4 пунктирными линиями красного и зелёного цветов представлена производительность одномерного и двумерного алгоритма циклической пересылки соотвественно. Синяя линия – производительность библиотечной функции MPI_Allreduce. Заметно, что она действительно хорошо оптимизирована для векторов размера до 1 Мб. Если использовать для хранения тип $uint32\_t$, требующий 4 байта на элемент, то это менее 2^{18} элементов. После этого скорость обработки выходит на плато. На бóльших векторах одномерный алгоритм циклической пересылки становится более эффективным, чем MPI_Allreduce.

Двумерный алгоритм с фиксированной длиной первого измерения решётки, равной 14, оказался самым медленным для больших векторов, хотя на маленьких показывает аналогичный одномерному результат. Возможная причина — оказалось $\tilde{\beta}_1 > \tilde{\beta}_2$ и нужно было взять r_1 меньше, чем 14. Это вполне могло получиться из-за более эффективного использования кэш–памяти в случае деления вектора на блоки меньшего размера. Установленные на "Ломоносове-2" процессоры имеют объём L3 кэш–памяти 35 Мб, это около 2^25 байт. Двумерный вариант алгоритма начинает проигрывать по скорости

двум другим даже чуть раньше – кэшируются не только данные, но и код, поэтому для рабочих блоков остаётся ещё меньше места.

5. Вариант с асинхронным сложением

Предыдущий вариант прост в реализации и является решением ”в лоб”, таким, какое приходит в голову первым. Однако при его анализе можно заметить два существенных недостатка:

- Сложение происходит отдельно от посылок. Поскольку скорость коммутации намного ниже скорости сложения, можно выполнять сложение параллельно с посылками. Тем самым можно добиться $\tilde{\beta} = 2\beta$ вместо $\tilde{\beta} = 2\beta + \gamma$.
- Сложение большими блоками приводит к снижению эффективности использования кэш-памяти. Это было показано, например, У.Дреппером в [6]. Поэтому лучше разбить эти блоки на более мелкие части и работать с ними.

5.1. Описание программы

Новый вариант программы предполагает разбиение блоков на ”пакеты”, размер которых можно варьировать (см. рисунок 3). Первый пакет принимается блокирующей операцией MPI_Recv, затем идёт запрос на получение следующего пакета асинхронной операцией MPI_Irecv и сложение пакета, полученного на предыдущем шаге. До получения идёт асинхронная отправка пакетов MPI_Isend, чтобы не произошёл т.н. *deadlock* — состояние, в котором каждый из процессов ждёт ресурсов, занятых друг другом, и ни один из них не может продолжать своё выполнение. Начиная со второго шага цикла, процесс начинает ожидать завершения передачи пакета, начатой на прошлом шаге, блокирующей операцией MPI_Wait. Код описываемого фрагмента программы можно найти в приложении.

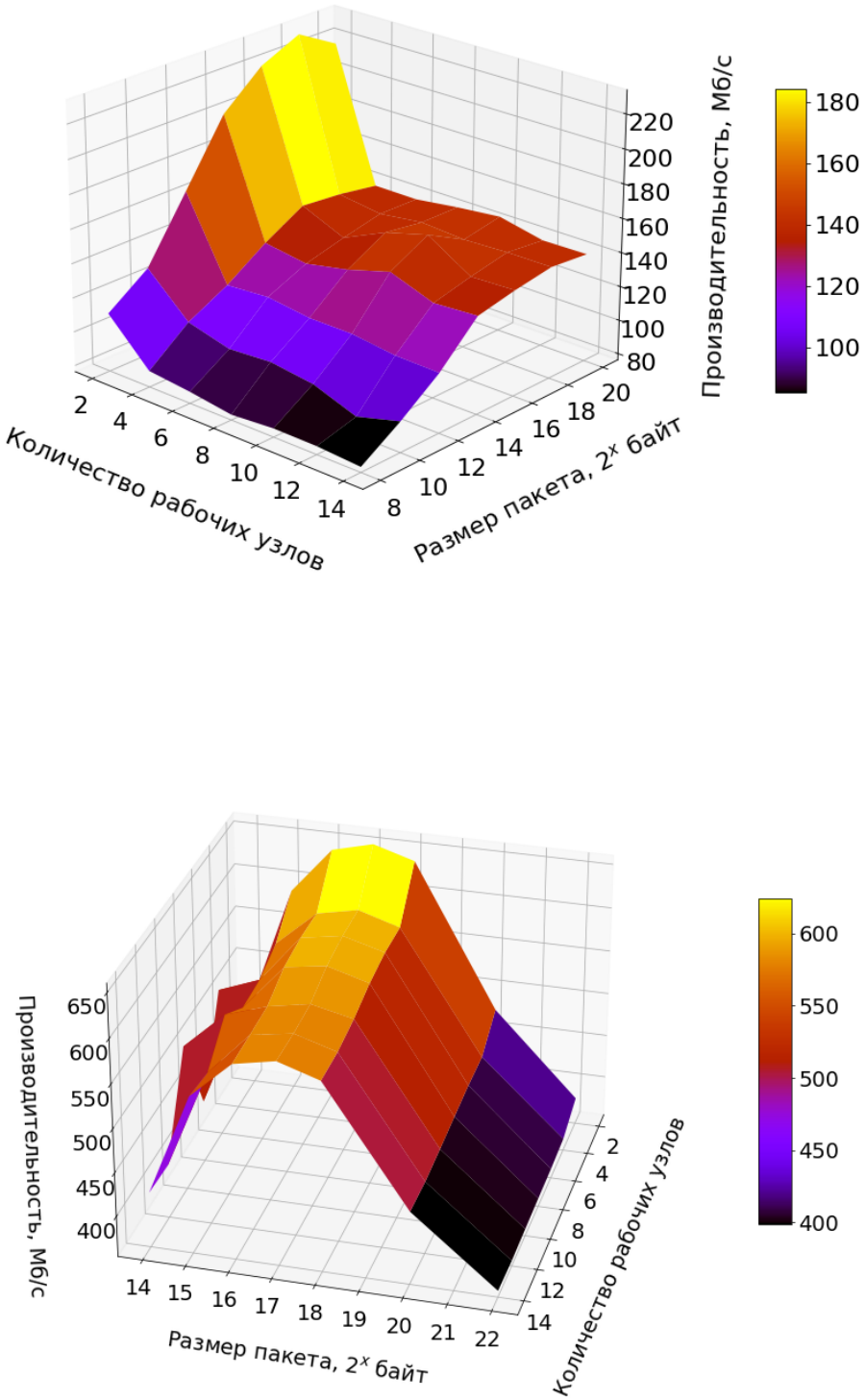

Рис. 3: Зависимость производительности от размера пакета. Верхний график с вектором размера 2^{20} байт, нижний – 2^{28} байт. Двумерный алгоритм.

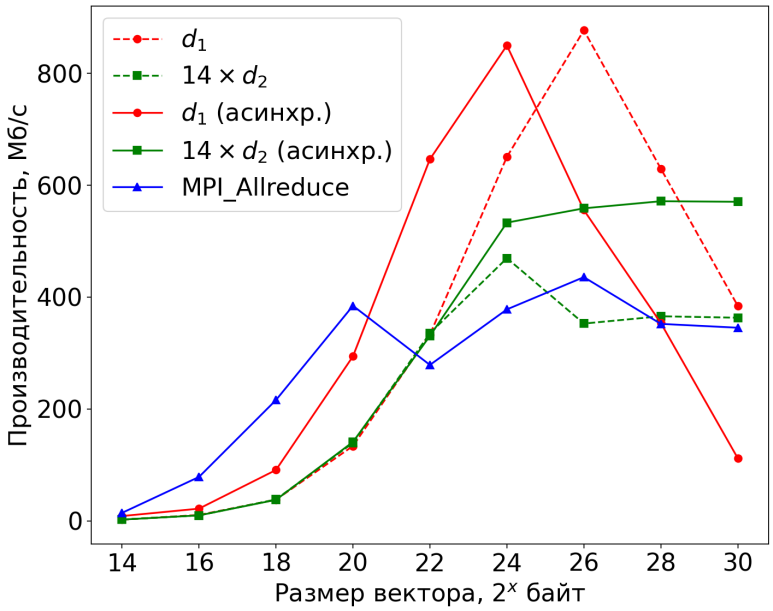

Рис. 4: Результаты запусков на 210 процессах (15 узлов). В эксперименте с асинхронным сложением использовались пакеты размера 256 Кб.

5.2. Результаты запусков на "Ломоносове-2"

На основе представленных графиков можно сделать несколько выводов. Во-первых, низкий уровень наклона прямой, обсуждавшейся в 3.3, соответствует действительности. Во-вторых, асинхронное сложение позволило достичь большей производительности на меньших размерах вектора у одномерного алгоритма, а у двумерного – на больших. В-третьих, для размера пакета лучше всего подходят значения в районе 2^{16}-2^{18} байт (64-256 Кб). Это неслучайно: у процессоров на ”Ломоносове-2” размеры кэшей L1 и L2 – 64 и 256 килобайт соответственно.

6. Метод исследования

Исходный код компилировался с использованием флагов $-O3$ и $-mavx2$. Алгоритм запускался в цикле 10 раз, и использовалось арифметическое среднее от полученных 10 значений времени работы.

В работе использовался компилятор mpicc версии 19.0.5.281 (совместимый с gcc 9.1.0) и библиотека Open MPI версии 4.0.1.

7. Заключение

Мною был изучен ряд статей, содержащих описания и анализ представленного алгоритма. Написана программа в двух вариантах: без асинхронного сложения и с ним, произведено несколько запусков на реальном оборудовании — на ”Ломоносове-2”. Замеры времени показали, что алгоритм циклической пересылки с асинхронным сложением работает лучше, чем него. Одномерный его вариант превосходит над функцией MPI_Allreduce из библиотеки Open MPI на векторах, больших 2^{20} байт (1 Мб), а двумерный — их обоих с 2^{26} байт (64 Мб). Изучено поведение алгоритма на разных размерах пересылаемого пакета, сделан вывод, что лучше всего подходят пакеты 64-256 Кб.

1. Приложение

1.1. Первая версия программы (отрывок)

```
312: void step1(segment *segments, int *coordinates, int *vector, int *buffer) {
313:   int prev, next; // Ранги соседей, куда отправлять отрезки
314:   MPI_Status st;
315:   segment send, recv;
316:
317:   for (int dim = 0; dim < d; dim++) {
318:     get_neighbours(&prev, &next, coordinates, dim);
319:
320:     for (int i = dimensions[dim]; i > 1; i--) {
321:       get_segments(&send, &recv, segments, coordinates, i, dim);
322:
323:       MPI_Sendrecv(&vector[send.start], send.len, MPI_INT, next, i, buffer,
324:                    recv.len, MPI_INT, prev, i, GlobalComm, &st);
325:       add_buff(vector, buffer, recv);
326:     }
327:
328:     int sum_idx = comm_shift(coordinates, 1, dim);
329:     segments[dim + 1] = new_segment(segments, sum_idx, dim);
330:   }
331: }
```

1.2. Вторая версия программы (отрывок)

```
201: // v1 -- адрес вектора, куда складывать
202: // v2 -- адрес вектора, откуда отправлять
203: // src -- ранк узла для получения,
204: // dst -- ранк узла для отправки
205: // size1 -- размер вектора на получение,
206: // size2 -- размер вектора на отправку
207: // comm -- коммутатор
208: // operation -- либо функция сложения, либо функция копирования
```

```
209: void SendRecvSum( int * v1, int * v2, int size1, int size2,
210:  int src, int dst, MPI_Comm comm, operation_t operation) {
211:   int *buffer1 = (int *)malloc(PACKET_SIZE * sizeof(int));
212:   int *buffer2 = (int *)malloc(PACKET_SIZE * sizeof(int));
213:   int *cb = buffer1; // Текущий буфер, в него будем загружать
214:   int *pb = buffer2; // Предыдущий буфер, с ним будем складывать
215:   int psize; // Размер предыдущего пакета
216:   int csize; // Размер текущего пакета
217:   MPI_Request send, crecv, precv;
218:   MPI_Status stat;
219:   int tag = 0;
220:
221:   do {
222:     if (likely(size2 > 0)) {
223:       MPI_Isend(v2, (size2 > PACKET_SIZE) ? PACKET_SIZE : size2, MPI_INT, dst,
tag, comm, &send);
224:       size2 -= PACKET_SIZE;
225:       v2 += PACKET_SIZE;
226:     }
227:     if (unlikely(tag == 0 && size1 > 0)) {
228:       psize = (size1 > PACKET_SIZE) ? PACKET_SIZE : size1;
229:       MPI_Recv(pb, psize, MPI_INT, src, tag, comm, &stat);
230:       size1 -= PACKET_SIZE;
231:     }
232:     if (likely(size1 > 0)) {
233:       csize = (size1 > PACKET_SIZE) ? PACKET_SIZE : size1;
234:       MPI_Irecv(cb, csize, MPI_INT, src, tag + 1, comm, &crecv);
235:       size1 -= PACKET_SIZE;
236:     } else csize = 0;
237:     if (likely(tag > 0 && psize > 0)) MPI_Wait(&precv, &stat);
238:     if (likely(psize > 0)) {
239:       operation(v1, pb, psize);
240:       v1 += psize;
241:     }
242:     psize = csize; precv = crecv; pb = cb;
```

```
243:       cb = (cb == buffer1) ? buffer2 : buffer1;
244:       tag++;
245:     } while (size1 > 0 || size2 > 0);
246:
247:     if (likely(psize > 0)) {
248:       MPI_Wait(&precv, &stat);
249:       operation(v1, pb, psize);
250:     }
251:
252:     free(buffer1);
253:     free(buffer2);
254: }
```

Список литературы

1. *Черепнёв М. А.*, *Замарашкин Н. Л.* Универсальный блочный метод Ланцоша – Паде для систем линейных уравнений над большими простыми полями // Фундаментальная и прикладная математика. — 2014. — Т. 19, № 6. — С. 223—247.
2. Global combine on mesh architectures with wormhole routing / M. Barnett [и др.] // [1993] Proceedings Seventh International Parallel Processing Symposium. — 1993. — С. 156—162.
3. Collective Communication: Theory, Practice, and Experience / E. Chan [и др.] // Concurrency and Computation: Practice and Experience. — 2007. — Сент. — Т. 19. — С. 1749—1783. — DOI: 10.1002/cpe.v19:13.
4. Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community / V. V. Voevodin [и др.] // Supercomput. Front. Innov. — 2019. — Т. 6. — С. 4—11.
5. Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems / J. Vienne [и др.] //. — 08.2012. — DOI: 10.1109/HOTI.2012.19.
6. *Drepper U.* What Every Programmer Should Know About Memory //. — 2007.